
White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/246769/

Version: Published Version

Proceedings Paper:
Maton, M., Kapfhammer, G.M. and McMinn, P. (2026) multiplex: a modular LLM-based mutation framework. In: Winter, S., Nejati, S. and Petke, J., (eds.) ASE '26: Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering. ASE '26: 41st IEEE/ACM International Conference on Automated Software Engineering, 12-16 Oct 2026, Munich, Germany. ACM, pp. 3468-3472. ISBN: 9798400728822. ISSN: 1938-4300. EISSN: 9174-8585.

https://doi.org/10.1145/3832783.3834617

© 2026 Copyright held by the owner/author(s). This work is licensed under a Creative Commons Attribution International 4.0 License.
(https://creativecommons.org/licenses/by/4.0/)

Reuse
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:
https://creativecommons.org/licenses/

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.





multiplex: A Modular LLM-Based Mutation Framework

Megan Maton
University of Sheffield
Sheffield, United Kingdom
m.e.maton@sheffield.ac.uk

Gregory M. Kapfhammer
Allegheny College
Meadville, USA
gkapfham@allegheny.edu

Phil McMinn
University of Sheffield
Sheffield, United Kingdom
p.mcminn@sheffield.ac.uk

Abstract

Mutation testing introduces synthetic defects into a program under test to assess test suite adequacy. While recent advances in large language models (LLMs) have led to more realistic and hard-to-kill mutants, the field is currently dominated by single-purpose research prototypes that tightly couple the mutant generation technique with the LLM provider and build environment, thereby requiring mutation testing researchers to re-implement techniques for empirical comparison. To address this challenge, this paper introduces *multiplex*, a modular LLM-based mutation framework. *multiplex* enables researchers to implement their own LLM-based mutant generation (MG) modules to prototype or compare techniques, as well as configure and create language, LLM or framework-specific modules for extended study. In addition to providing an overview of the system design and implementation, this paper demonstrates how researchers can use *multiplex* to prototype and compare techniques, along with sharing experiments conducted with *multiplex* and outlining future extensions. The tool is available at https://github.com/LLM-Mutation/multiplex, and a video demonstration is available at https://youtu.be/l-hj7Pmn3oo.

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. multiplex: A Modular LLM-Based Mutation Framework. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834617

1 Introduction

Mutation testing is a state-of-the-art method for evaluating test suite adequacy through inserting synthetic faults into the code-under-test and asserting whether the test suite detects them. Recent advances in large language models (LLMs) have enabled mutation testing researchers to create tools that generate more diverse, context-specific mutants that resemble real code defects [15].

Current LLM-based mutant generation (MG) approaches are embedded into environment-specific tools (often research prototypes) that dictate the language, build system (e.g., Maven) or LLM provider that a researcher must use. However, the MG techniques themselves, being prompt-based, are often platform agnostic. As such, mutation testing researchers are having to re-implement MG techniques with a common execution mechanism and LLM provider in order to evaluate them in empirical studies [12, 15]. Not only does this create additional work, it also introduces a threat to validity regarding the implementation accuracy. This means that there is an urgent need for a unified, but modular, LLM-based mutant generation framework that enables researchers to share and compare MG techniques, while also using them in their chosen environment.

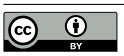
To address this challenge, this paper presents *multiplex* (*mu*tation analysis with *l*arge language models for *t*esting, *i*n a *p*rompt-*l*evel *ex*perimentation framework), a modular framework designed for rapid prototyping and empirical comparison of LLM-based mutation techniques. *multiplex* enables users to construct and share modules for language parsing, mutant generation, LLM interfacing (i.e., cloud and local) as well as mutant execution and evaluation. Along with describing *multiplex*'s design and implementation, this paper shows how mutation testing researchers can use and extend it. It also overviews a comparison of five LLM-based MG techniques using multiple LLMs against 279 Defects4J bugs using *multiplex* as well as transferring these MG techniques to Python. We offer these methods as existing modules with the *multiplex* framework.

2 Related Mutant Generation Tools

Traditional rule-based tools, such as Major [8] and PIT [3] for Java, apply small syntactic code changes based on predefined transformation patterns. Large language models (LLMs) have enabled the generation of context-aware mutants that mimic real developer-introduced faults [12, 15]. Generally, techniques fall under two categories: Mask-and-Predict and Guided Re-implementation.

Mask-and-Predict techniques "hide" a segment of code from the language model and prompt it to infill the segment with possible alternatives. The pre-LLM tool, μBERT, masks tokens within Java expressions and asks the pre-trained language model, CodeBERT, to predict them [4]. μBERT then replaces masked tokens with predicted tokens to create mutants. LLMorpheus uses a similar approach, but instead prompts LLMs to generate the replacements for a pre-set list of masked code patterns [14]. LLMorpheus can switch LLMs, but is tied to StrykerJS's mutant execution mechanism. Both tools were initially evaluated only against traditional rule-based baselines, as no other LLM-based approaches existed for comparison at the time. However, the field now demands direct comparisons between competing LLM-based strategies which can be facilitated by the *multiplex* tool, as demonstrated in Section 3.3.

Guided Re-implementation provides the LLM with a code segment and requests differing implementations given specific guidance. The BugFarm tool creates hard-to-detect and hard-to-repair



This work is licensed under a Creative Commons Attribution 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834617

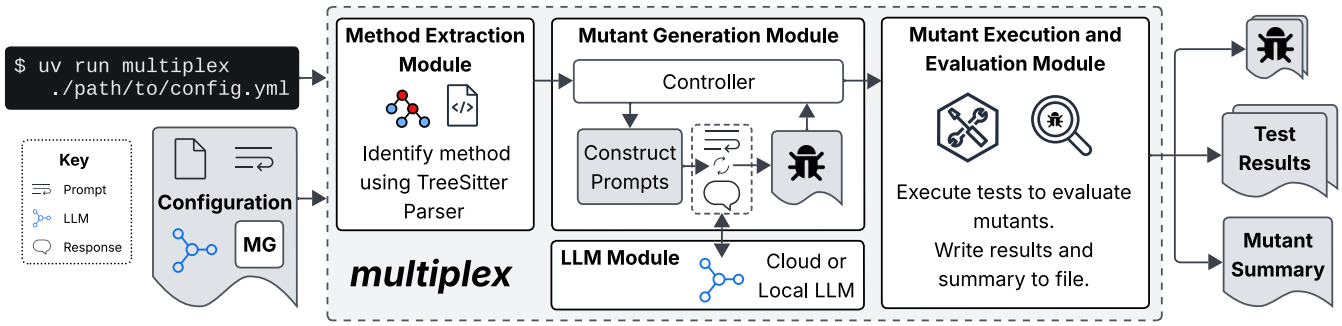


Figure 1: System overview of *multiplex*. Researchers can design and adapt each module to facilitate empirical studies.

mutants by identifying multiple individual lines that are not "well-attended" by the test suite and prompting an LLM to mutate them, before combining them as a mutant [7]. The open-source tool, MUTAHUNTER, is both language- and build-tool-agnostic [2]. Although not documented, MUTAHUNTER supports either an OpenAI cloud LLM or an Ollama-run local LLM. MUTAHUNTER is one of the most popular open-source LLM-based mutation tools by GitHub star count, highlighting a demand for this type of approach.

The LLMUT tool was created to explore the use of a few-shot prompting technique across different LLM configurations [15]. Wang et al. performed a comprehensive study comparing LLMUT with both rule- and LLM-based mutant generation strategies, including BUGFARM and LLMORPHEUS [15]. To perform their study, Wang et al. re-implemented the LLMORPHEUS tool in Java to enable a comparison with the Java prompts and tools. The SEMANTIC MUTATION WITH ADAPTIVE RETRIEVAL AND TUNING (SMART) approach applies retrieval-augmented generation (RAG) using a vectorized dataset of real bugs, code chunking and LLM fine-tuning, aiming to improve mutant generation with LLMs [16]. While this study compares SMART with LLMORPHEUS and LLMUT, it seems to also have used its own implementation of LLMORPHEUS, as the original tool is not available for Java. This study also compares multiple LLMs for SMART and LLMUT, but uses only a single LLM for LLMORPHEUS. Using *multiplex* would enable this study to be performed in multiple languages and by using multiple LLMs without the need for re-implementation, instead only needing approach configuration.

Intent-based mutation testing uses mutated code intents to prompt the LLM to implement an incorrect code intent [5]. The Round-Trip Mutation Testing (RTM) approach uses LLMs to extract the code intent from a piece of code, mutate the intent and ask the LLM to implement the mutated intent [6]. This approach is similar in concept to the existing *multiplex* implemented approaches and therefore can be implemented in *multiplex* for comparison.

All of these LLM-based mutant generation methods are conceptually platform-agnostic in that they primarily consist of text-based prompts submitting to a LLM. However, they are embedded in tools that are often coupled with a specific programming language, build environment or LLM provider [12, 15], thus making them difficult to clearly compare in experiments. The popularity of tools like MUTAHUNTER reveals an underlying desire for platform-independent techniques. To overcome this challenge, *multiplex* offers a unified framework for building and sharing mutant generation techniques, enabling researchers to conduct standardized comparisons regardless of language or test framework, thereby removing the time-consuming need for a researcher to re-implement each one.

3 *multiplex* Background and Existing Use

We created *multiplex*, shown in Figure 1, to implement LLM-based mutant generation (MG) techniques guided by hazard analysis [12]. This section overviews hazard analysis and explains why using it to guide LLM-based MG necessitated the creation of *multiplex*.

3.1 Hazard Analysis Techniques

Safety-critical industries employ hazard analysis methods to evaluate processes for potential risks, enabling engineers to identify the possible causes and consequences of things going wrong. The HAZOP (HAZard and OPerability study) method identifies hazards as deviations from the original intended design using guide words, such as *More* or *Part Of*, to rephrase the intended design to cause deviations. For instance, given a train, applying *More* to the parameter *Speed* may cause it to operate beyond safe limits. The STPA (System-Theoretic Process Analysis) method defines hazards as "unsafe control actions" (UCAs) issued by a controller—human or software—through inappropriate, missing or untimely commands. For instance, if a train driver were to issue an open-door command *Too Early*—before the train stops—this would constitute a UCA despite the individual components functioning correctly. To apply these methods to LLM-based mutant generation, three common LLM-supported steps were required: first, describe the code under test—for HAZOP, this required natural-language descriptions; for STPA, a control flow diagram. Second, perturb these descriptions using each method's prescribed guide words to create HAZOP deviations and STPA UCAs for the code-under-test. Finally, re-implement the code using the hazardous descriptions to create mutants. Exploring these techniques required a modular framework for prototyping and comparing mutant-generation techniques.

3.2 Existing Mutant Generation Modules

By creating *multiplex* modules to support HAZOP- and STPA-guided mutant generation (i.e., *m*-HAZOP and *m*-STPA, respectively), we enable direct comparison between the techniques using only a single tool version. To further compare these techniques with the existing ones, LLMORPHEUS and MUTAHUNTER, we implemented

them as *m*-LLMorpheus and *m*-MutaHunter, respectively, to produce comparable outputs and focus on how the techniques compare rather than the platforms in which they are implemented. LLMorpheus, for example, although using language-agnostic LLM prompts, is integrated into the JavaScript mutation tool StrykerJS and therefore required integration into a Java execution system for inclusion in this study. *multiplex* also includes a further LLM-based "unguided" baseline module, called *m*-unguided, that simply asks the LLM to generate mutants with no additional guidance (e.g., hazard analysis like *m*-HAZOP and *m*-STPA, or size and location like *m*-LLMorpheus and *m*-MutaHunter). Each module is included in the *multiplex* repository for researchers to use [11].

3.3 Evidence of Utility

This section overviews experiments performed using *multiplex* to generate and evaluate mutants for both Java and Python projects.

3.3.1 Hazard-Guided Java Mutants in Defects4J. This section summarizes an empirical comparison conducted with *multiplex* across five LLM-based techniques and Major, a rule-based tool, over 279 Defects4J bugs [12]. The comparison introduced two hazard analysis-guided techniques, *m*-HAZOP and *m*-STPA, as mutant generation approaches. Results showed that these approaches—when evaluated against *m*-unguided, *m*-LLMorpheus, *m*-MutaHunter and Major—generated large, unique method-level mutants, and complement existing techniques in both subsumption and mutant behavior using both cloud (GPT-4.1) and local (gpt-oss:20B) LLMs.

Below is the `println` method in Defects4J bug `Csv` version 5. To show the diversity in LLM-based approaches, the following paragraph describes the mutants created by each approach.

```
1 public void println() throws IOException {
2   final String recordSeparator = format.
      getRecordSeparator();
3   if (recordSeparator != null) {
4     out.append(recordSeparator);
5   }
6   newRecord = true;
7 }
```

m-STPA creates control flow-based mutants, and therefore, one of its mutants *Does not provide* the `null` check, and therefore deletes Lines 3 and 5. *m*-HAZOP uses the *More* guide word to add a `for` loop around Line 4 to repeat it three times. *m*-unguided deletes the entire method body and replaces it with "`newRecord = false;`". *m*-LLMorpheus restricts mutation to specified nodes and changes the condition on Line 3 to "`format != null`" by changing the variable that dictates the condition. Lastly, *m*-MutaHunter mutates Line 4 to "`out.append(recordSeparator + "CORRUPTION");`" to simulate a corrupted string. While targeting a simple method, each of these techniques produces diverse mutants specific to its design. This highlights the value of different LLM-based mutant generation strategies and the need to compare them with a modular tool.

3.3.2 Hazard-Guided Python Mutants in the PySnooper project. This section summarizes an end-to-end execution of *multiplex* on 10 functions from the PySnooper Python project using gpt-oss:20B using the *m*-HAZOP, *m*-STPA and *m*-unguided approaches [13]. This resulted in 434 mutants with mutation scores of 0.72, 0.53 and 0.80, from each aforementioned approach respectively, across the 215 competent mutants (i.e., those that did not cause errors). Similar to those obtained in the prior Java study, these results demonstrate how *multiplex* enables multi-language studies of such approaches.

4 *multiplex* Design and Implementation

multiplex serves as a modular LLM-based mutation testing framework to enable rapid prototyping and empirical comparisons of LLM-based mutant generation approaches. Unlike existing tools, *multiplex* provides the utility of interchangeable modules for method extraction, MG approach, LLM interfacing and mutant execution and evaluation. This design supports custom study configurations and baseline controls through specifying the mutation testing approach and setup in the input configuration file.

4.1 Inputs and Configuration

Built with Python, *multiplex* uses the `uv` project manager, enabling researchers to execute the tool using the following CLI command:

```
> uv run multiplex ./path/to/config.yml
```

To support reproducibility, *multiplex* takes a single `yaml` configuration file as input to define the mutation testing setup. This ensures that researchers can share their setup by specifying the project under test, mutant generation approach, LLM and system prompts.

4.1.1 Project under test. The first section of the `yaml` configuration file details the project language and the locations of the file and method under test. It also specifies the test "`runtool`" which, in the following configuration example using Java and Defects4J, is `d4j`.

```
project:
  language: "java"
  projectroot: "./project/csv_4_fixed"
  filename: "./path/to/CSVParser.java"
  method: "getHeaderMap"
  line: "287"
  runtool: "d4j"
```

multiplex can also mutate Python projects by changing the `language` and `runtool` configurations to `python` and `pytest`, respectively.

4.1.2 Mutant Generation (MG) Approach. This section specifies the MG module the user wishes to use to generate mutants.

```
mutation:
  approach: "basic"
```

In this example, `basic` refers to the *m*-unguided module described in Section 3.2. Instructions for extending *multiplex* are provided in Section 4.6, with additional information in the tool's repository [11].

4.1.3 Large Language Model (LLM). *multiplex* can use both local and cloud-based LLMs. The configuration file requires the model name, endpoint and the name of the environment variable used to store the LLM API token for cloud LLMs.

```
llm:
  model: "ollama/gpt-oss: 20b"
  endpoint: "http: //127.0.0.1: 11434"
  token_env_var: "null"
```

By requiring the environment variable containing the API key rather than the token itself, *multiplex* prevents accidental credential exposure and enables researchers to confidently share configurations for reproducibility. In this example, the user sets the token variable to `null` because it configures a locally deployed LLM.

4.1.4 System Prompts. Finally, the configuration file contains the system prompts for the mutant generation modules, enabling them to have multiple steps and prompts that are uniquely identifiable.

```
system_prompts:
  basic_generate_mutants:  |
    "...
    When given a method, you are required to re-
    implement it to introduce an **incorrect
    behavior or defect**.
    ..."
```

The aforementioned example presents an excerpt of the system prompt for the *m*-UNGUIDED approach module. The user prompts are constructed within the MG modules themselves, enabling each approach to customize them for a given input. Users can find working examples with full configuration file examples for both Python and Java in the `examples/` directory in the tool repository [11].

4.2 Method Extraction

multiplex is written in Python and currently targets mutation testing for Java and Python programs. To extract the method- or function-under-test, the user must specify its location in the configuration, as shown in the example in Section 4.1.1. *multiplex* utilizes the TREESITTER parser to iterate through the file concrete syntax tree (CST), searching for the method/constructor/function node matching the name and line number given in the configuration [1]. The TREESITTER system provides parsers for multiple languages, enabling users to set up method extraction modules for their chosen language, as described in Section 4.6.1. After extracting the method, *multiplex* saves it to the `original_method.java` file for mutation.

4.3 Mutant Generation Approach

While the MG approaches are customizable, we provide some examples with the *multiplex* tool to help users get started. Each approach module includes a `controller.py` file that accepts the LLM model, the output directory and the prompts from the configuration as inputs. This design flexibly supports multi-step approaches, such as *m*-HAZOP and *m*-STPA. As described in Section 3, these modules first prompt the LLM to represent the code via natural language (*m*-HAZOP) or control-flow diagrams (*m*-STPA), then perturb these representations to generate source code mutants. To optimize performance, *multiplex* then uses the TREESITTER parser to pre-screen mutants for syntax errors. At this stage, *multiplex* records whether the mutant *should* be compilable in the `mutant_summary.csv` file, thereby avoiding redundant compilation attempts. These multi-step approaches place all source-code mutants and intermediate outputs in the output directory for execution and evaluation.

4.4 Mutant Execution and Evaluation

Mutant execution and evaluation come as a pair, as the module must be able to parse passing and failing tests from the execution

```
output
├── stpa-mutants
│   ├── mutant_0.java
│   └── mutant_summary.csv
├── stpa-test
│   └── mutant_0_test.txt
├── control_diagram.txt
├── original_method.java
└── ucas.csv
```

Figure 2: Output directory file structure example for *m*-STPA.

tool output. To support users in running and evaluating Java and Python mutants in empirical studies, *multiplex* comes with a test runner for MAVEN, DEFECTS4J framework and PYTEST. The module uses the project's working directory and executes the tests specified by the provided test execution command. *multiplex* then parses the test outputs to determine whether the mutant was first compilable, whether it is surviving or killed, and if so, which tests killed it. Finally, *multiplex* adds the test outcomes for each mutant to the output directory and updates `mutant_summary.csv` to reflect them.

4.5 Tool Outputs

The design of *multiplex* prioritizes transparency by logging every stage of the MG process. Figure 2 shows an example output directory file structure from a *m*-STPA MG approach execution on a Java method, with other included approaches following a similar structure. The `original_method.java` file contains the source code for the method under test for the approaches to use. For *m*-STPA, the `control_diagram.txt` file contains the LLM-generated control flow diagram, and the `ucas.csv` contains the "unsafe control actions" (UCAs) perturbed from it. Archiving these intermediate steps is essential for interpreting the sequential, non-deterministic outputs of LLMs. The `*-mutants/` dir then contains each mutated version of `original_method.java`, which under *m*-STPA implements a UCA. After executing the tests, the `*-test/` dir contains the test outcomes, including the reasons for build failures (if there is a compilation issue) and which tests fail when executed on the mutant and therefore kill it. These comprehensive output directories assist both manual analysis and systematic fine-tuning of mutation modules.

4.5.1 Outputting a Marv-Compatible Summary. MARV is a mutant visualization tool that enables users to view mutant source code diffs and review them [17]. *multiplex* provides a `–marv` flag option that enables users to output the mutant data to a MARV-compatible `JSON` file. Full instructions for setting up and using MARV with *multiplex* can be found in the *multiplex* repository [11].

4.6 Adding *multiplex* Languages and Modules

multiplex is modular by design; therefore, we encourage users to add languages and modules to facilitate their research. Full details for extending *multiplex* can be found in the tool's documentation [11].

4.6.1 Adding a Language. To add compatibility with a programming language to *multiplex*, users must create a language specification in the `_REGISTRY` list in `multiplex/languages/__init__.py`. The language must have a TreeSitter parsing library as this is used

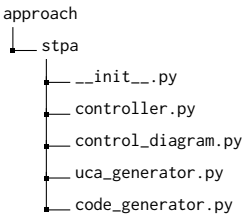


Figure 3: Mutant generation approach directory file structure example for *m*-STPA.

for code extraction and mutant insertion [1]. The user must specify information such as the language file extension, the TreeSitter language library, nodes to be mutated as well as some approach-specific strings to be inserted into the prompts sent to the LLM.

4.6.2 Adding a LLM Module. *multiplex* uses LiteLLM to interface with LLM models (both cloud- and locally-hosted), enabling most LLMs to be configured via a configuration file. However, when some providers require additional information, users can update the `multiplex/model.py` file to support specific endpoints.

4.6.3 Adding a Mutant Generation (MG) Module. To extend *multiplex*, a user must add an MG package to the `multiplex/approach` directory containing a `controller.py` file that directs their approach, as shown in the example for *m*-STPA in Figure 3. For instance, *m*-STPA uses a three-step process in which it prompts the LLM at each step: firstly, to create a control flow diagram; secondly, to derive "unsafe control actions" (UCAs) from the diagram; and finally, to implement these UCAs as mutants. In Figure 3, each of these steps has its own class and is called by the controller. As shown in Figure 2, each output is saved to the `outputs/` directory.

4.6.4 Adding an Execution and Evaluation Module. Users can add a mutant execution and evaluation class to `multiplex/execute/` that inserts the mutant into the source code file, runs the project's tests with the appropriate build and execution tools, and parses the results to identify which mutants are killed by which tests.

5 Conclusions and Future Work

This paper introduces *multiplex*, a modular LLM-based mutation framework to assist researchers in prototyping and empirically comparing LLM-based mutant generation approaches. *multiplex*'s modular nature ensures that it can be adapted and extended to facilitate multi-language and approach comparisons, such as the Java and Python examples in Section 3.3. As well as overviewing *multiplex*'s design and implementation, this paper describes how users can extend and use the tool for their research. We envision *multiplex* as a workbench for LLM-based mutation and, therefore, future work should continue to assist users in prototyping and comparing LLM-based MG approaches. A core component of this will be a plugin and template system that enables researchers to create modules without forking the *multiplex* repository. As such, the configuration options will enable users to create their own modules independently. Subsequently, we will create an open-source record of MG modules, along with including configuration options for the code structures under test, as working at the method or function level may not be appropriate for all paradigms. Leveraging *multiplex* and a completed version of the proposed future work, researchers can refine and evaluate LLM-based mutant generation approaches and readily share their experimental setup and results.

Acknowledgments

Megan Maton and Phil McMinn are supported by the EPSRC grant "Test FLARE" (EP/X024539/1).

Data Availability Statement

The *multiplex* version used in this paper is archived on Zenodo and the latest version in development is available on GitHub [11]. The replication packages referenced in Section 3.3 are available in archived GitHub repositories [9, 10].

References

[1] M. Brunsfeld, A. Qureshi, A. Hlynskyi, ObserverOfTime, W. Lillis, J. Vera, dundargoc, P. Turnbull, T. Clem, D. Creager, A. Helwer, R. Rix, D. Kavolis, H. van Antwerpen, M. Davis, C. Clason, R. Bruins, Ika, T-A. Nguyễn, A. Ya, S. Brunk, M. Massicotte, bfredl, N. Hasabnis, M. Dong, S. Moelius, S. Kalt, S. Finer, and Kolja. 2025. TreeSitter. doi:10.5281/zenodo.15383273

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

[3] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*. doi:10.1145/2931037.2948707

[4] R. Degiovanni and M. Papadakis. 2022. μBERT: Mutation Testing using Pre-Trained Language Models. In *Proceedings of the International Workshop on Mutation Testing*. doi:10.1109/ICSTW55395.2022.00039

[5] A. Hamidi, A. Khanfir, and M. Papadakis. 2025. Intent-Based Mutation Testing: From Naturally Written Programming Intents to Mutants. In *Proceedings of the International Workshop on Mutation Testing*. 347–357. doi:10.1109/ICSTW64639.2025.10962508

[6] A. Hamidi, C. Richter, A. Khanfir, and M. Papadakis. 2026. Round-Trip Mutation Testing: Translating Code to Natural Language Intent and Back. In *Proceedings of the International Workshop on Mutation Testing*. 259–264. doi:10.1109/ICSTW72326.2026.00050

[7] 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*. doi:10.1109/SCAM67354.2025.00021

[8] R. Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of International Symposium on Software Testing and Analysis*. doi:10.1145/2610384.2628053

[9] M. Maton. 2026. Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches Replication Package. https://github.com/LLM-Mutation/ease-2026-replication-package

[10] M. Maton. 2026. Multiplex Study Replication. https://github.com/LLM-Mutation/ase-2026-replication-package

[11] M. Maton. 2026. multiplex (https://github.com/LLM-Mutation/multiplex). doi:10.5281/zenodo.21775953

[12] M. Maton, G. M. Kapfhammer, and P. McMinn. 2026. Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*. doi:10.1145/3816483.3816524

[13] R. Rachum. 2026. PySnooper. https://github.com/cool-RR/pysnooper

[14] F. Tip, J. Bell, and M. Schaefer. 2025. LLMorpheus: Mutation Testing using Large Language Models. *IEEE Transactions on Software Engineering* 51 (2025). doi:10.1109/TSE.2025.3562025

[15] B. Wang, M. Chen, M. Deng, Y. Lin, M. Harman, M. Papadakis, and J. M. Zhang. 2026. A Comprehensive Study on Large Language Models for Mutation Testing. *ACM Transactions on Software Engineering and Methodology* (2026). doi:10.1145/3805038

[16] B. Wang, M. Deng, M. Chen, C. Yang, Y. Lin, M. Harman, M. Papadakis, and J. M. Zhang. 2026. Boosting LLMs for Mutation Generation. *Proceedings of the ACM on Software Engineering* (2026). doi:10.1145/3808156

[17] D. Wells, Z. Lévai, and P. McMinn. 2026. Marv: An Interactive Tool for Visualising and Reviewing the Results of Mutation Analysis. In *Proceedings of the International Conference on Automated Software Engineering, Tools and Datasets Track*. doi:10.1145/3832783.3834643

Received 2026-05-12; accepted 2026-06-19