



Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/246767/>

Version: Published Version

Proceedings Paper:

Wells, D., Lévai, Z. and McMinn, P. (2026) Marv: An interactive tool for visualising and reviewing the results of mutation analysis. In: ASE '26: Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering. ASE '26: Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering, 12-16 Oct 2026, Munich, Germany. ACM, pp. 3596-3600. ISBN: 9798400728822.

<https://doi.org/10.1145/3832783.3834643>

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.

MARV: An Interactive Tool for Visualising and Reviewing the Results of Mutation Analysis

Daniel Wells[✉]

University of Sheffield
Sheffield, UK
dwells1@sheffield.ac.uk

Zalán Lévai[✉]

University of Sheffield
Sheffield, UK
zblevai1@sheffield.ac.uk

Phil McMinn[✉]

University of Sheffield
Sheffield, UK
p.mcminn@sheffield.ac.uk

Abstract

Mutation analysis is a useful technique for both practitioners and researchers in examining the strength of a test suite. If the change introduced by a mutant cannot be detected, it may represent a “blind spot” that the test suite failed to consider. However, there are currently no publicly available tools that developers and researchers can use to examine mutants produced by the plethora of existing mutation analysis tools that currently exist for a variety of programming languages (e.g., PIT for Java and Stryker for JavaScript). This is increasingly important given the fact that nowadays, systems are composed of code written in multiple languages. Instead, code editors and diff analysers are used, which are not well optimised for this particular activity. Where tools do exist, they are focused on specific mutation tools/programming languages only. In this paper, we introduce the MARV tool (Mutations Analysis, Review and Visualisation) to specifically address this problem. MARV provides in-code visualisations of mutants and information about them from multiple mutation analysis tools simultaneously. The tool allows users to complete formal textual reviews of each mutant and can export results from all of its supported frameworks in a standardised JSON format. MARV currently includes support for 13 different mutation analysis frameworks that collectively span 12 different programming languages. A screencast demonstrating MARV is available at <https://youtu.be/qMjVgXmWrnk>.

CCS Concepts

• Software and its engineering → Software testing and debugging.

Keywords

Software Testing, Mutation Testing, Mutation Analysis, Mutant Visualisation

ACM Reference Format:

Daniel Wells, Zalán Lévai, and Phil McMinn. 2026. MARV: An Interactive Tool for Visualising and Reviewing the Results of Mutation Analysis. 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/10.1145/3832783.3834643>

1 Introduction and Motivation

Mutation analysis is a technique that is used by practitioners and researchers to evaluate the effectiveness of a test suite by introducing small, deliberate changes—called *mutations*—into a program’s source code. These mutations are designed to mimic programming mistakes, such as changing an arithmetic operator. Each modified version of the program containing a mutation is known as a *mutant*. The test suite is then run against each mutant to see if one or more of the tests can detect the change. If a test fails as a result of the mutation, the mutant is said to be “killed”, otherwise—i.e., if all tests still pass—the mutant survives.

One value for testers using mutation analysis lies in examining surviving mutants. Each surviving mutant is a potential “blind spot” on the part of the test suite, indicating areas of the program under test where the test suite has failed to adequately validate the program’s behaviour. By investigating these mutants, testers can design more targeted tests or adapt existing ones to focus on these areas to “kill” these mutants. This ultimately builds greater confidence in the software’s correctness, and reduces the risk of actual defects reaching production. These observations are also useful for researchers designing test generation methods, where mutations provide a useful way of assessing how thorough generated test suites are, and how they compare to those generated by different techniques. That said, not all surviving mutants may be useful. They may be “equivalent” to the original program—i.e., no behavioural change occurs in the program despite the code change, or a “duplicate” of a different mutant—i.e., the behavioural change replicates that of another mutant. Finally, a mutant can also be unproductive, altering code regions not essential for correctness—e.g., the formatting of a log line.

For these reasons, both developers and researchers need to examine the code of mutants so that they can decide how to improve their test suites or test generation methods. However, as we will discuss in Section 3, there are a large number of mutation analysis tools (each of varying capability) for a range of different programming languages—for example, PIT for Java [2], Stryker for JavaScript [10] and mutest-rs for Rust [4], to name only a few. Beyond existing code editors, diff analysers, and analysis tools not specifically designed for examining mutants, there are no publicly available tools that allow users to study, in detail, mutants that have resulted from more than one mutation tool, due to the use of multiple programming languages. In particular, cross-language support is important in the modern era of software development, where individual software solutions comprise code in multiple programming languages [11]—a study from 2017 reports professional developers utilising an average of seven languages per project [6].



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/10.1145/3832783.3834643>

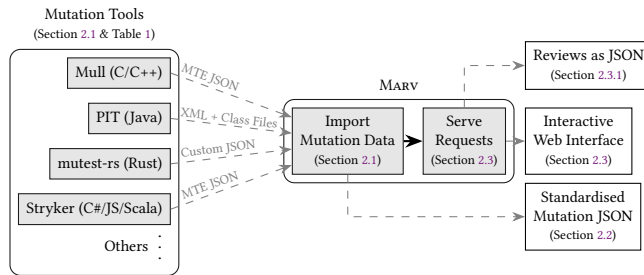


Figure 1: High-level overview of the sequence of steps taken by MARV to visualise mutations from various tools, producing an interactive web interface and standardised JSON.

In response to this, we designed and implemented a tool called MARV (Mutations Analysis, Review and Visualisation) to address this gap. Specifically, MARV provides in-code visualisations of mutants and information about them from multiple mutation analysis tools simultaneously, allowing its users to complete individual reviews. Support for each mutation tool is native to MARV, and it exports its results in a common format. To date, MARV provides support for 13 different mutation analysis frameworks, which collectively operate over 12 different programming languages, with plans to add support for more tools and languages in the future.

2 The MARV Tool

We designed MARV to not require changes to any of the existing mutation analysis tools, and instead conform to the various outputs generated by each of them. Rather than waiting for mutation analysis tools to provide support for a particular output format (as with the Mutation Testing Elements (MTE) approach, discussed in detail in Section 4), users and developers of MARV can instead add support for mutation tools directly in MARV itself. Developers can do this by implementing a simple “schema” (i.e., interface), which further allows MARV developers to maintain and improve support for unique features that a particular mutation analysis tool may provide. For simplicity, we designed MARV to be configurable with the various mutation analysis tools in a standalone configuration file (`.marv.yml`), and run using the `marv` command. This allows users to configure MARV for their project once with the specific mutation analysis tools used, and invoke it with the same command everywhere. Additionally, we designed MARV to aid users in the initial configuration by providing a “`marv init -f [FRAMEWORK]`” command to generate much of the configuration necessary for each tool. Running MARV performs all tasks necessary to process and visualise program mutations. First, it imports the mutation data from each tool according to the project-specific `.marv.yml` configuration file (Section 2.1). As part of this step, MARV also emits a standardised JSON-based representation of all mutations from all tools, which can be used by other external tooling to read mutation data without the added complexity of handling different tool output formats (Section 2.2). After importing all of the mutation data, MARV then runs as a local web server to provide users with an interactive interface showing the mutations from the various tools configured, and provides a clickable URL which opens the user’s browser. (Section 2.3). Figure 1 shows an overview of the steps MARV takes to import and visualise the mutations of multiple

mutation analysis tools. MARV is written in the Go programming language.

2.1 Data Importing from Different Mutation Analysis Tools

For MARV to be able to import the mutation output of a mutation analysis tool, it must contain an implementation of the generic Framework Go interface, defined by MARV, for the specific mutation analysis tool. This interface declares the public procedures that are required for MARV to understand a tool’s output. The methods load, process and transform the output data from a specific mutation analysis tool into the MARV format. Once the mutations are in MARV’s own standardised mutation format, it is able to treat any mutation from any tool in an identical way. Thus, all of the mutation visualisation and review facilities of MARV are completely generic over the mutations of all supported tools, reducing the differences between how the various mutation analysis tool outputs are handled to a minimum. This also has the advantage of only requiring the implementation of a small, simple interface for MARV to be able to import the data of any future, currently unsupported mutation analysis tool.

Because the output of the various mutation analysis tools has major, fundamental differences compared to the output of other tools, we had to implement a number of tool-specific considerations and solutions to support a great number of mutation analysis tools. We present these differences and tool-specific considerations in more detail in Section 3.

To configure the mutation analysis tool outputs to load for a project, our tool uses a `.marv.yml` YAML-based configuration file, typically placed in the root directory of a software project. The configuration file enumerates the tools used by the project, and the corresponding options, such as the location of the tool outputs. The initial configuration can be generated using the helper command “`marv init -f [FRAMEWORK]`”.

2.2 Standardised Format to Represent Mutations

Differences in the outputs of mutation analysis tools tend to make manual examination of mutants from different tools difficult, if not infeasible. This situation is inevitable for any project written in multiple programming languages, as is often the case in industry.

Because MARV already creates a common representation of program mutations regardless of what mutation analysis tool generated and evaluated them, our tool makes this available in JSON format, written to a file when it is run. This common, standardised representation of the mutations can then be used by other external tools, making cross-language, cross-tool mutation analysis feasible, regardless of the number of languages and mutation analysis tools used. This standardised representation of mutations includes all important information common amongst various mutation analysis tools, including a long-form textual description of the mutation, the name of the mutation operator that generated the mutation, the region of code that was changed by the mutation, the textual replacement for the region of code the mutation changes, and the mutation’s status as one of seven overarching statuses defined by MARV. It is worth noting that, in some cases, this information has

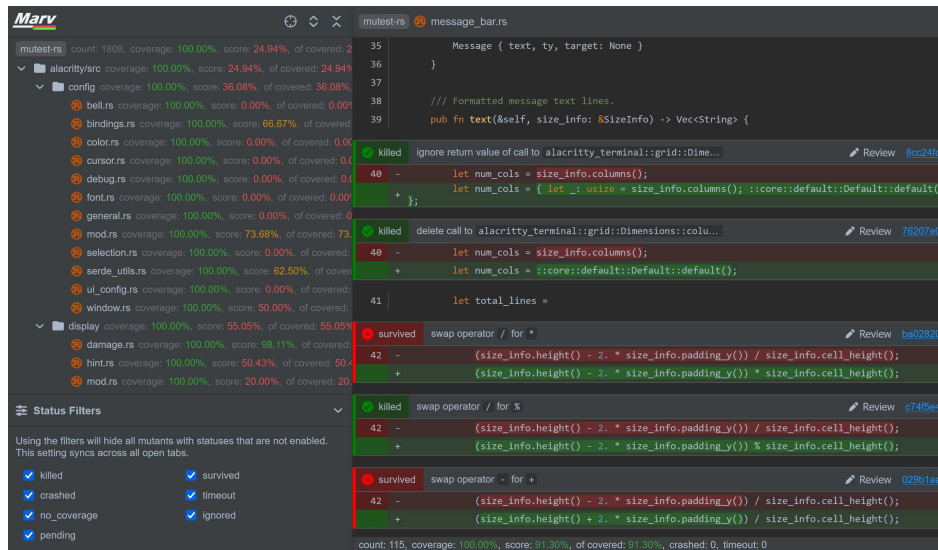


Figure 2: Screenshot of MARV showing multiple Rust mutations generated by mutest-rs. The left sidebar displays statistics for each file, and allows navigation, as well as toggling the visibility of certain mutations. Mutations are all displayed inline.

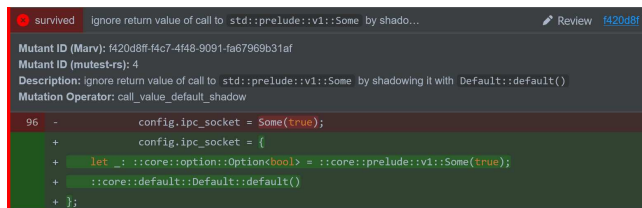


Figure 3: MARV showing the details of a particular mutation. A free-form review can be written into the textbox revealed by pressing the “Review” button.

to be deduced by MARV itself, as some mutation analysis tools do not provide all of the necessary data in a readily available format. We discuss these cases in more detail in Section 3.

MARV always validates the standardised mutation data it produces before exporting it and launching the web interface.

2.3 Interactive Web Interface for Mutations

Once MARV imports the mutation data from all configured mutation analysis tools, it starts a local web server to serve generated HTML pages for the various views of our tool’s interactive web interface. The choice of a web-based interface makes our tool accessible on any platform with a modern browser, and allows for future integrations into the various web-based code review platforms (e.g., GitHub). Figure 2 shows MARV’s main view. This shows mutations for a particular file inline with the original source code, with every mutation placed next to its corresponding original code. Because mutations do not have to be revealed one-by-one, it allows for getting an overview of the various mutations and their status for a particular snippet of code. More detail can be revealed for each mutation by clicking its inline representation. A sidebar on the left shows a tree of source files along with statistics of the mutations within them, and can be used for navigation. The per-file statistics

aid the user in prioritising the most critical files with low mutation coverage. A panel below can be used to toggle the visibility of certain mutations based on their status.

2.3.1 Reviewing Mutations through Comments. To facilitate the manual user review of the individual mutations generated by mutation analysis tools, MARV implements a system for writing and storing free-form reviews (depicted by Figure 3). By clicking the “Review” button placed next to every mutation, the user can write a review in the revealed textbox. These reviews are then stored into a separate JSON file alongside the standardised mutations JSON file. This output allows for the further processing of reviews authored through MARV’s user interface.

2.4 MARV in an Agentic Workflow

MARV processes data from supported frameworks into a standardised and intuitive format (Section 2.2). This makes MARV exceptionally well suited for integration into increasingly relevant agentic workflows, where AI agents rely on well-formatted and validated data to accomplish a series of tasks. We highlight two such agentic workflows:

Agentic Test Improvement Loops. The “marv export -m” command produces a single JSON output file containing the processed results from all mutation analysis tools included in the MARV configuration file. Given knowledge of a codebase and access to the relevant mutation analysis tools, an AI agent can run the mutation analysis tools on the target codebase, and use the MARV export command to standardise and unify the results. The agent can then edit existing tests or write new tests to counter the surviving mutations. This can be done iteratively with the agent building on its previous work each time until a high mutation score has been reached.

Human-Agent Cooperation. In addition, the “marv -m” command produces a single JSON output file, as above, but leaves the MARV

Table 1: Mutation Analysis Tools Currently Supported by MARV

Tool Name	URL	Language	Level of Support
Mewt	https://github.com/trailofbits/mewt	C/C++	Out of the box
Mull	https://mull-project.com	C/C++	Out of the box
stryker-net	https://github.com/stryker-mutator/stryker-net	C#	Out of the box
go-mutesting	https://github.com/zimmski/go-mutesting	Go	Out of the box
PIT/pitest	https://github.com/hcoles/pitest	Java	Requires external Java decompiler library
Major	https://mutation-testing.org	Java	Out of the box (Only for Major 1.3.0+)
stryker-js	https://github.com/stryker-mutator/stryker-js	JavaScript/TypeScript	Out of the box
infection	https://github.com/infection/infection	PHP	Out of the box
Cosmic Ray	https://github.com/sixty-north/cosmic-ray	Python	Out of the box
mutant	https://github.com/mbj/mutant	Ruby	Out of the box
cargo-mutants	https://github.com/sourcefrog/cargo-mutants	Rust	Out of the box
mutest-rs	https://github.com/zalanlevai/mutest-rs	Rust	Out of the box
stryker4s	https://github.com/stryker-mutator/stryker4s	Scala	Out of the box

web server running. This allows an AI agent to run the relevant mutation analysis tools, read the MARV output, and then author reviews containing its proposed test suite changes or additions into MARV’s review database via the review API. Then, via the textual review field in the MARV web interface, researchers and developers can moderate, approve or disapprove each change proposed by the agent. Once the reviewer ends the MARV session, the agent can use the exported JSON file to make the approved changes.

3 Evidence of Utility

MARV supports a large variety of mutation analysis tools, with only PIT requiring a small change to how it is invoked for full compatibility with our tool. This includes 13 tools covering 12 programming languages. We plan to extend the list of supported tools as part of our future work, continuing to implement the Framework interface for new tools. Table 1 shows the list of mutation analysis tools currently supported by our tool. Mutation analysis tools can further opt to use MARV by exporting results in MARV’s JSON schema, which MARV can process as a form of “generic” tool. This is the route taken by the recent LLM-based mutation framework multiplex [5], which uses LLMs to produce mutants for different program languages, rather than traditional rule-based operators. Since multiplex itself supports a number of LLM-based mutation tools and prompting techniques, it currently has no standard format of its own other than via MARV’s own schema.

Supporting the Various Outputs of Mutation Analysis Tools. To date, there is no standardised output format for mutation analysis tools. As such, each mutation analysis tool defines its own outputs. These output formats can differ greatly from each other, largely owing to fundamental differences in their underlying mutation methodologies. There are a few tools with notable output formats that require additional considerations: PIT, Infection and Mull.

PIT, in regular operation, only exports an XML file containing the mutation metadata and/or its formatted HTML output. Neither of these outputs contains the actual source code mutations which are required by MARV. As such, for compatibility with MARV, PIT must be invoked with the `-Dfeatures="+EXPORT"` flag which exports both the source and mutated class files for each mutation. This enables MARV to retrieve the source code for a mutation by decompiling both the source and mutated class files (the user can configure a specific decompilation method) and observing the differences. Source code mutations are then displayed in the correct

position within the original source code file. This works well for most mutations; however, due to inevitable formatting changes between the decompiled class files and the original source code file, for some mutations MARV will show an incorrect number of removed lines.

MARV supports both Mull and Infection through their MTE schema outputs; however, currently both produce errors in these outputs. These problems are primarily to do with where the mutation is located in the original source file. Currently, some Mull operators indicate their start position and not their end position, whilst some Infection operators provide incorrect end positions. In most cases MARV corrects these errors using logic provided in the framework implementation; however, in some cases these cannot be corrected, and MARV dumps the “broken” mutations into a separate JSON file, before continuing to display the remaining mutations as usual.

4 Existing Tools

There are a number of mutation analysis tools available today for producing mutants for a variety of programming languages, some of which are reported in Table 1. However, there are a surprisingly small number of tools that have built-in features, visual or otherwise, for helping developers analyse the mutants produced and the results of test suites when executed with them.

Mothra [3] was a pioneering interactive software testing environment developed in academia in the late 1980s and early 1990s, supporting the Fortran 77 and Ada programming languages. Among other features, the tool included a source code display showing mutant lines that were prefixed with a “#” character.

Two modern publicly available tools with in-built developer support include PIT [2] and Mutation Testing Elements (MTE) [8], designed for the Stryker suite of tools [9]. PIT is a Java mutation testing framework [2] with an HTML reporting tool offering two views. The first presents mutation statistics both as text and as colour-coded horizontal bars using red and green to indicate pass/fail ratios. It lists the mutated files with their individually calculated statistics as well as overview statistics for the whole package. This view is reused to list all packages that were tested. The second is a code view with no syntax highlighting. It shows which lines were mutated, and how many mutations were made on that line, with a numbered indicator positioned adjacent to the line number. Hovering the cursor over the indicator reveals a tooltip detailing textual descriptions of each mutant of that line. It uses

a two-tier colour system—light green and red for code coverage, and darker shades for killed versus surviving mutants—with a full list of all mutant descriptions at the bottom of the view. PIT makes mutations to the Java bytecode and therefore does not show the code changes that were created.

Stryker Mutator [9] is a mutation testing organisation that has developed mutation testing frameworks for different programming languages, along with an HTML reporting tool called Mutation Testing Elements (MTE) [8]. MTE is a suite of elements that can build a mutation testing report from any results formatted in the MTE JSON schema. As such, MTE could eventually support many tools, but currently is supported by a small subset of mutation testing tools, each implementing and maintaining the MTE schema (to varying degrees of completeness and correctness) as an optional output format. The MTE report provides two main views, an interactive files list and a code view. The file list represents a directory within the tested project, and displays a list of all the file entries contained within the currently open directory. Each entry is accompanied by a range of metrics. The code view displays syntax highlighted source code, where languages are supported, and indicates the locations of mutants using coloured dots that appear at the end of a line. MTE displays all mutants in a collapsed state by default. A mutant can be expanded by clicking its coloured indicator, which reveals the mutant's source code changes. Only one mutant can be viewed at a time—clicking a different mutant's coloured indicator will close any visible mutants and then show the new mutant. Mutants are associated with an expandable information panel attached to the bottom of the screen. This panel can provide descriptions, mutation IDs, operators and links to covering and killing tests; however, this data is optional and fields are frequently ignored by tools implementing the MTE schema. MTE has both light and dark themes and provides a file name search system in all views. MTE has no utility to identify or correct mistakes in the schema output of third-party tools, which can lead to confusing and misleading formatting errors where tools have implemented the schema incorrectly.

In industry, Google developed a mutation visualisation tool integrated into their code review workflow [7], where detected mutants—referred to as “findings”—are presented to reviewers in their code review interface outlining each mutation. Reviewers are shown a generated description of each finding and given two single-click response options: dismissing it as “Not useful”, or flagging it as “Please fix”. Similarly, Meta embed a mutation visualisation tool in their code review system [1]. However, their tool leverages the existing version control interface rather than a custom display. Unlike Google, Meta do not use guiding prompts and instead leave developers to assess and act on mutants independently.

While these tools demonstrate a range of approaches to mutation visualisation, none is suitable as a general-purpose solution. Mothra is now obsolete, handling legacy programming languages. The Google and Meta tools, while more modern, are proprietary systems deeply embedded in their internal code review infrastructures and are unavailable outside those organisations. PIT and Stryker MTE, though open-source and actively maintained, are either tied to a specific programming language (i.e., Java in the case of PIT), offer static HTML reports with no or basic interaction, support a small subset of languages, rely heavily on the existence, maintenance

and correctness of third-party data structure implementations, or offer inconsistent experiences depending on provided tool output. Collectively, these limitations highlight a gap in the availability of language-agnostic, accessible mutation visualisation tooling.

5 Conclusions

We introduced the MARV tool to allow developers and researchers to review the results of mutation analysis and individual mutants from one or several supported frameworks. MARV provides a standardised results format that can be used in analysis of results from multiple frameworks, or as input to future programs.

Acknowledgments

Daniel Wells and Phil McMinn are supported, in part, by the EPSRC grant “Test FLARE” (EP/X024539/1). Zsolt Lévai is supported by the EPSRC grant EP/W524360/1.

Data Availability Statement

Our open-source tool, MARV, is publicly available online under a BSD-3-Clause license at <https://github.com/SecretSheppy/marv>. The repository contains the tool's source code written in the Go programming language, as well as instructions on how to compile and run the tool in the README.md file. The MARV tool is also distributed as a single binary, and can be installed with the command “go install github.com/SecretSheppy/marv@latest”.

References

- [1] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry—A Study at Facebook. In *Proceedings of the International Conference on Software Engineering — Software Engineering in Practice track (ICSE-SEIP 2021)*. IEEE, Madrid, Spain, 268–277. doi:10.1109/ICSE-SEIP52600.2021.00036
- [2] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 449–452. doi:10.1145/2931037.2948707
- [3] R.A. DeMillo, D.S. Guindi, W.M. McCracken, A.J. Offutt, and K.N. King. 1988. An Extended Overview of the Mothra Software Testing Environment. In *Proceedings of the Workshop on Software Testing, Verification, and Analysis*. IEEE, Banff, AB, Canada, 142–151. doi:10.1109/WST.1988.5369
- [4] Zsolt Lévai, Donghwan Shin, and Phil McMinn. 2026. mutest-rs: Flexible, Efficient Mutation Analysis Tool for Rust Programs, Using Extensive Static Analysis. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST 2026)*. IEEE, Daejeon, South Korea, 216–220. doi:10.1109/ICST69053.2026.00038
- [5] Megan Maton, Gregory M. Kapfhammer, and Phil McMinn. 2026. multiplex: A Modular LLM-based Mutation Framework. In *Proceedings of the International Conference on Automated Software Engineering (ASE Tools and Datasets Track)*. ACM, Munich, Germany. doi:10.1145/3832783.3834617
- [6] Philip Mayer, Michael Kirsch, and Minh Anh Le. 2017. On Multi-Language Software Development, Cross-Language Links and Accompanying Tools: A Survey of Professional Software Developers. *Journal of Software Engineering Research and Development* 5, 1 (April 2017), 1. doi:10.1186/s40411-017-0035-z
- [7] Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *Proceedings of the International Conference on Software Engineering — Software Engineering in Practice track (ICSE-SEIP 2018)*. ACM, New York, NY, USA, 163–171. doi:10.1145/3183519.3183521
- [8] Stryker Team. 2026. Mutation Testing Elements (MTE). <https://github.com/stryker-mutator/mutation-testing-elements>
- [9] Stryker Team. 2026. Stryker Mutator. <https://github.com/stryker-mutator>
- [10] Stryker Team. 2026. StrykerJS. <https://github.com/stryker-mutator/stryker-js>
- [11] Haoran Yang, Yu Nong, Shaowei Wang, and Haipeng Cai. 2024. Multi-Language Software Development: Issues, Challenges, and Solutions. *IEEE Transactions on Software Engineering* 50, 3 (March 2024), 512–533. doi:10.1109/TSE.2024.3358258

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