PyMut4SE: Comprehensive Mutation Testing for Python

Laura Plein
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
laura.plein@cispa.de

Matthieu Jimenez
University of Luxembourg
Esch-sur-Alzette, Luxembourg
matthieu.jimenez@uni.lu

Mike Papadakis
University of Luxembourg
Esch-sur-Alzette, Luxembourg
michail.papadakis@uni.lu

Abstract

Mutation analysis involves examining the behavior of program mutants to identify meaningful behavioural differences, with the aim of supporting a wide range of software engineering activities, including testing, debugging, program improvement, change impact analysis, and the understanding of modified program behavior. A critical aspect of mutation analysis lies in the quality of the mutations employed, as inadequate mutation sets can lead to poor observations and, consequently, unreliable results. To address this challenge, we present PyMut4SE, a novel mutation tool for Python that focuses on a comprehensive set of mutations for any Python project. PyMut4SE stores mutants, along with their associated metadata and behavioral outcomes, in an SQLite database, allowing for a variety of downstream software engineering tasks. The tool is designed to support fundamental research in software engineering by providing a rich and extensible set of mutation operators, access to mutated source code and its characteristics, detailed execution and behavioral observations, and support for both selective mutation and the generation of higher-order mutants.

ACM Reference Format:
Laura Plein, Matthieu Jimenez, and Mike Papadakis. 2018. PyMut4SE: Comprehensive Mutation Testing for Python. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Mutation analysis [4, 7] has been widely adopted in software engineering research to support a variety of development and analysis activities, including automatic test generation [10], debugging [8], program improvement [1], and prediction of change effects [11]. A common step across these applications is the analysis, prediction, and understanding of the dynamic behavior of programs in response to syntactic changes introduced by mutations. In other words, mutation analysis enables the systematic study of how small code modifications influence program behavior.

Mutation analysis is highly sensitive to the quality of the mutations used [6, 7]. Inadequate or poorly designed mutation sets can lead to limited behavioral diversity, resulting in incomplete observations and, consequently, unreliable conclusions [7, 8]. At the same time, mutation analysis provides a systematic approach for generating program variants (mutants), offering a reproducible and controlled mechanism for experimentation. This makes it particularly well-suited for empirical software engineering, where repeatability and controlled study conditions are essential.

We present PyMut4SE, a Python mutation tool capable of generating both first-order and higher-order mutants for any project written in Python. The tool systematically stores all generated mutants alongside the original program version, including their precise location within the codebase, the actual modified lines, and the applied mutation operators. PyMut4SE executes each mutant against a set of tests within an isolated execution environment. For each execution, the tool records the resulting outputs, with corresponding error types and messages. This detailed tracing enables a direct mapping between software changes and their behavioral effects. Such mappings can be subsequently analyzed and leveraged as training data to learn the relationship between code modifications and program behavior, and vice versa [9].

Current research has focused on Java [2, 12], with only few exceptions for Python [3]. MutPy [1], and MutMut [2] are Python tools that compute mutation scores while providing some statistics about the number of mutant kills. However, they do not provide the actual mutated code, which is needed in several software engineering tasks. CosmicRay [3] is, to the best of our knowledge, the only Python mutation tool that provides some information about the mutants. However, further tooling is required to reconstruct the mutants. For some of the mutations of CosmicRay, e.g., the `NumberReplacer`, the information about what number is replaced by which new value is not available, making them hard to reconstruct. Furthermore, it is not possible to execute specific mutants directly on a set of test inputs and collect the behavior changes.

Perhaps more importantly, none of the Python mutation tools implements a comprehensive set of mutations, and none supports higher order mutations thereby limiting their effectiveness, applicability and research utility. Overall, the existing mutation tools can only be applied to briefly assess the robustness of a test suite by applying the most common standard operators but they cannot be used as a tool to generate and execute a large diversity of single and higher order mutants on a diversity of test inputs.

PyMut4SE is publicly available at https://github.com/LaPlei96/PyMut4SE and has the following distinguishing features:

(1) automatic *exploration* and identification of mutation targets from a given project repository path.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

[1] https://github.com/mutpy/mutpy
[2] https://mutmut.readthedocs.io/en/latest/
[3] https://cosmic-ray.readthedocs.io/en/latest/index.html

(2) static inference of related test cases to mutation target to enable granular test suite execution on mutants.
(3) supports a comprehensive set of mutation operators, including *novel operators* for mutating function calls, enabling the generation of a large and diverse collection of first- and *higher-order mutants*.
(4) employs a *modular architecture* that separates mutant generation and execution, enabling straightforward *extensibility*, repeated execution on new inputs, and the generation of additional higher-order mutants.
(5) provides an isolated and reproducible execution environment, it stores the mutant's code and meta data with their execution results in a database, enabling fine-grained comparisons of program behavior beyond pass/fail outcomes.
(6) supports parallel execution allowing the number of workers to be configured according to available hardware.

2 Tool Features

Given a software project containing Python files, PyMut4SE starts an *exploration* phase that goes through the project to identify all the packages, modules, functions and test cases as illustrated in Figure 1. Then it moves to the *mutant generation* phase that creates mutants according to the operators recorded in Table 1. Then, during the *mutant execution*, all the mutants are executed either against crafted inputs or related test cases in a dedicated execution environment to avoid side effects. All related data, mutant locations, types and their execution data are stored in an SQLite database.

2.1 Exploration

Mutants can be used for several software engineering tasks. Depending on the task, one might only be interested in mutating a single function, a single module, a package, or even an entire project. To support this, PyMut4SE can operate at different levels of granularity. It starts with an exploration phase where, given a path, the explorer recursively traverses all folders, identifying packages, modules, and even functions (referred as Code Chunks) within modules, which are then stored in the database using an SQLAlchemy-backed data model. Furthermore, the explorer looks for all test files. Test files following common naming conventions, such as starting with `test_` or ending with `_test`, are identified first. The explorer then checks which modules are imported by the files in the test folder and iterates over the test cases to determine which function calls are made. This allows it to statically infer which test cases are related to which functions and modules. Thus, if a developer only wants to mutate a single function or a single file, the explorer automatically detects the relevant tests and function calls. Since executing mutants is expensive, it is essential to avoid running the entire test suite, which may contain many irrelevant test cases for each mutant. Once all the `Code Chunks`, together with their corresponding test cases and test inputs, have been stored in the database, the mutant generation phase can begin.

2.2 Mutant Generation

The standard mutation operators commonly discussed in the literature, like `Arithmetic`, `Relational`, `Constant Replacement`, and `Unary`, are included in PyMut4SE. Additionally, several `Deletion` mutation operators have been added. Other general operators, such as `Control` and `Logical Connector`, are also available. Since Python is a dynamically typed programming language, PyMut4SE also includes `Type Cast` mutations. Furthermore, because Python allows functions to be called with different combinations and types of arguments, the framework provides `Swap Arguments` as well as `Optional ParamCallee` and `Optional ParamCaller`. These operators represent common mistakes in Python projects that can lead to subtle or severe behavior changes.

Table 1: Implemented Mutation Operators

Mutation Operator	CosmicRay	PyMut4SE
`Arithmetic`	✓	✓
`ConstantReplacement`	✓	✓
`ControlReplacement`	✓	✓
`DeleteAssignment`		✓
`DeleteDecorator`	✓	✓
`DeleteIfStatement`		✓
`DeleteReturn`		✓
`DeleteWhile`		✓
`ExceptionReplacer`	✓	
`IfNotNull`		✓
`LogicalConnector`	✓	✓
`OptionalParamCallee`		✓
`OptionalParamCaller`		✓
`Relational`	✓	✓
`ReturnPass`		✓
`SwapArguments`		✓
`TypeCast`		✓
`Unary`	✓	✓

Table 1 gives an overview of all currently implemented mutation operators. For simplicity, it presents the general operator categories, e.g. Arithmetic without enumerating every single transformation rule that can be applied by replacing one of the 11 concrete operators like replacing *ast.Add()* by *ast.Sub()*. The operators were implemented following the Abstract Syntax Tree (AST) documentation [4]. Overall, PyMut4SE applies a large diversity of mutation operators allowing to generate more single order and higher order mutants than any of the previously introduced Python mutation tools as discussed in Section 4. Note that a more detailed description of each operator can be found on Github[5].

PyMut4SE can easily be extended to incorporate new mutation operators by extending the `Mutation` abstract class, or if the operator rely on the ast package of python, the `PythonASTMutation` class. To fully integrate, it will need to be incorporated to the api as well, although a dynamic resolution of the operators is planned in the future to ease the process.

2.3 Mutant Execution

The current mutant execution workflow distinguishes between input-based execution and test-based execution. For input-based execution, a code chunk or mutant is executed against explicitly provided function inputs. Inputs can be added through the workspace

[4] https://docs.python.org/3/library/ast.html
[5] https://github.com/LaPlei96/PyMut4SE/blob/main/docs/operators.md

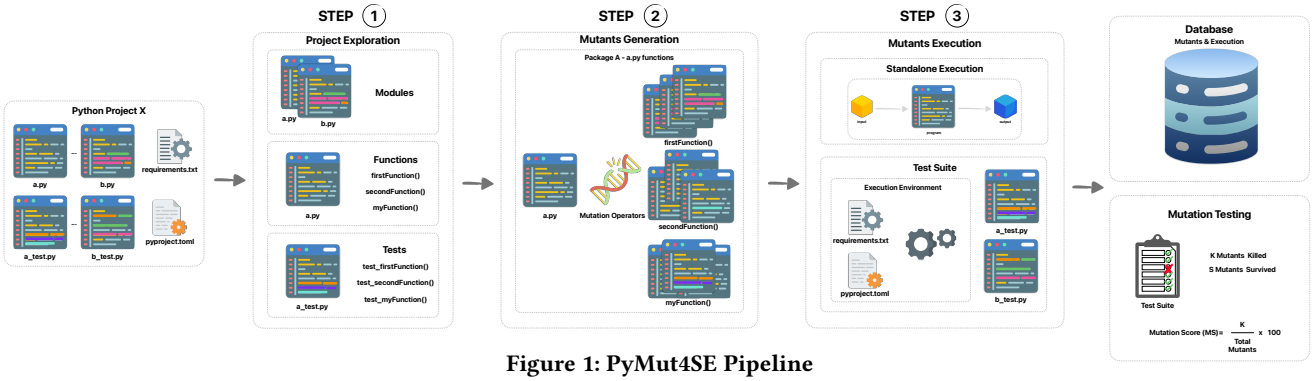


Figure 1: PyMut4SE Pipeline

API either as trusted Python values, serialized with `pickle`, or as textual representations such as JSON, Python literals, or function calls. This enables users to define inputs manually or import them from external generators and fuzzers [13]. Each execution runs in an isolated temporary copy of the project using a prepared Python environment, limiting side effects while associating results with the corresponding mutant, input, and execution environment.

For test-based execution, PyMut4SE leverages the test relations inferred during exploration. Since mutants inherit the test relations of their original code chunk, execution can be restricted to mutants covered by at least one inferred test case, avoiding unnecessary execution of uncovered mutants. Test execution supports both sequential and parallel modes. For each selected mutant, PyMut4SE creates an isolated temporary workspace, installs the mutated module alongside the relevant test files, and executes the selected `pytest` cases. The resulting execution status, output, return code, execution time, and associated mutant, test, and environment identifiers are stored as execution results. They can later be used to compute mutation scores, inspect surviving mutants, or re-execute selected mutants with additional tests or inputs.

3 Tool Availability

PyMut4SE is publicly available on GitHub at https://github.com/LaPlei96/PyMut4SE and is actively being developed. At the moment of writing, the *v1.0.0*[6] has just been released and provides sources and a ready to install package. The `README.md` contains a link to the *getting-started* which provides all the details on how to start running PyMut4SE, including a YouTube video showcase the use of the tool. The required dependencies are a Python installation, the uv [7] package manager, and sqlite [8].

To easily get started using PyMut4SE you can follow the `getting-started.md` start or jump into mutation testing with an example jupyter notebook to explore a project, generate and execute mutants, and compute the mutation testing score, available at https://github.com/LaPlei96/PyMut4SE/tree/main/notebook.

4 PyMut4SE Showcase

We showcase the use of PyMut4SE to *explore* a project and *generate* a large set of diverse mutants. Those mutants can be used for *testing* purposes and can be *executed* on a set of test inputs.

[6]https://github.com/LaPlei96/PyMut4SE/releases/tag/v1.0.0
[7]https://docs.astral.sh/uv/
[8]https://sqlite.org/

We also compare our tool with the other existing tools, i.e., CosmicRay. CosmicRay is the most promising existing Python mutation testing tool as indicated by a recent study [3]. CosmicRay is also the only Python mutation tool that claims to provide the mutated source code while implementing similar operators than MutPy and MutMut.

4.1 Mutation Testing

In our first experiment, we demonstrate the usefulness of our mutation tool for mutation testing. For this experiment we collected a set of well known open-source Python projects with test-suites that are available on GitHub (Requests[9], FastAPI[10]). For each project, we report the number of single-order mutants generated by PyMut4SE and the CosmicRay baseline that were killed or survived, along with the resulting mutation score. While PyMut4SE can generate a large amount of mutants for every function in the repository, many of those code parts are not covered by the existing test cases. For this study, we focused only on executing mutants that can be reached with the existing test suite. For CosmicRay, it is unclear whether these mutants are executed and included in the final mutation score, it seems that the full test suite which doesn't target the mutants is successfully executed since not impacted by the mutation and thus those mutants are all counted as killed. Additionally, CosmicRay does not provide details on how many executions failed e.g. due to timeouts; they seem to simply be counted towards the "killed" mutants too. PyMut4SE provides detailed information on what mutants were executed by the existing test suite and which ones weren't. For the mutation score, we only count the mutants that were actually tested. As can be seen in Table 3 our tool was able to generate a large portion of mutants that survived, further analyzing those mutants could provide valuable insights to improve the existing test suites. Additional preliminary experiments [11] while applying PyMut4SE to itself have also highlighted the usefulness of higher order mutants to generate additional mutants that survive.

4.2 Mutants & Diversity

In this experiment, we evaluate PyMut4SE with respect to two complementary aspects compared to existing mutation testing tools. First, we assess its ability to generate unique mutants. Second, we analyze the diversity and distribution of the mutation operators

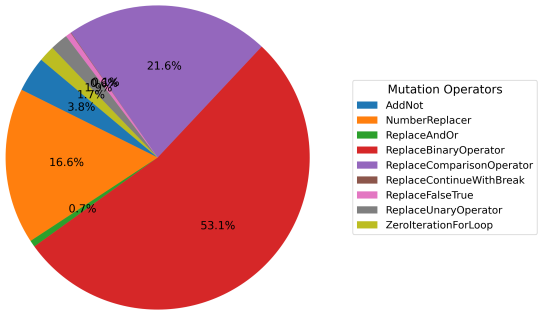
[9]https://github.com/psf/requests
[10]https://github.com/fastapi/fastapi
[11]https://github.com/LaPlei96/PyMut4SE/blob/main/notebook/example.ipynb

used during mutant generation. The evaluation is conducted on the QuixBugs [5] benchmark, the same benchmark was used in Section 4.3. QuixBugs consists of a collection of buggy and corrected Python programs along with their corresponding tests.

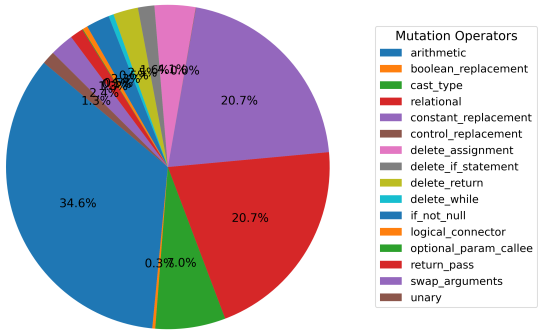
Table 3 summarizes the number of mutants generated by each tool. Since CosmicRay and other existing Python mutation testing tools generate only first-order mutants (SOMs), we first report that PyMut4SE generates over thousand more unique SOM than CosmicRay. The difference increases drastically upon considering HOMs, as with only a degree of 2, PyMut4SE generates over 120,327 unique mutants. Still, the raw value of unique mutants is not sufficient to assess the quality of a mutation tool. For example, a tool could artificially increase this number by generating many close variants through constant replacement, with limits diversity.

Consequently, we also investigate the distribution of mutation operators used by each tool, as a broader range of mutation operators is more likely to produce diverse source code modifications and behavioral changes. Figure 2 part (a) illustrates the distribution of the mutation operators applied to the QuixBugs benchmark by CosmicRay. As can be seen, half of the mutants were obtained by replacing a binary (`Arithmetic`) operator. The second and third mutation operators are replacing a comparison (`Relational`) operator and the number (`Constant`) replacement. Those three mutation types account for 91.3% of all the mutants generated by CosmicRay. This indicates that the generated mutant set is heavily concentrated on a small subset of mutation types.

Figure 2 part(b) presents the corresponding distribution for PyMut4SE. Although arithmetic, relational, and constant replacement remain the three most frequently applied mutation operators, they account for only 76% of the generated mutants. The remaining mutants are distributed across a wider range of mutation operators, resulting in a more balanced mutation profile. This increased diversity is particularly beneficial for applications that rely on learning or analyzing the effects of software modifications, as it exposes them to a larger spectrum of transformation and resulting behavioral changes.



(a) CosmicRay



(b) PyMut4SE

Figure 2: Mutation Type Frequency

4.3 Additional Applications

PyMut4SE has successfully been used as mutation tool in Neural Change Prediction [9, 11]. The goal of Neural Change Prediction is to learn the effect of software changes on programs behavior and vice versa. To learn correlations between pairs of code and behavior changes, a large number of learning samples is essential. For this purpose, PyMut4SE was used to generate a large amount of diverse single and higher order mutants. In this work, the effect of software changes was successfully learned solely on data generated by PyMut4SE, achieving a correct code change generation in 68.5% of the cases, a fine-grained fault localization at the line level with 82.6% accuracy. Additionally, it was possible to predict the effect of software changes with 87.1% accuracy for subtle behavior changes and even with 95% accuracy for predicting failures and exact error types. Since this study required the source code of the mutants as well as detailed information about the software's behavior on a diversity of inputs, PyMut4SE was used since none of the previous Python mutation testing tools provided the necessary information. PyMut4SE is thus far more than just a mutation testing tool, but mostly a tool to generate diverse single and higher order mutants while providing insights into their behavior.

Table 2: Mutation Testing Results

Project	Requests		FastAPI	
Tool	CosmicRay	PyMut4SE	CosmicRay	PyMut4SE
Total Mutants	4994	8267	17161	26854
Killed Mutants	100%	17.27%	100%	20.56%
Survived	0	48.49%	0	40.47%
Error/Timeout	-	34.23%	-	38.97%
Mutation Score	100%	26.27%	100%	33.69%

Table 3: Number of Mutants for Quixbugs

	CosmicRay	PyMut4SE SOM	PyMut4SE HOM
Mutants	1670	2739	120327

5 Conclusion & Future Work

In this work, we presented PyMut4SE, a novel mutation tool for Python that enables a customizable and extensible mutation analysis of Python-based projects. PyMut4SE supports a wide range of mutation operators and provides full support for the generation and execution of both first-order and higher-order mutants. The generated mutants, along with their source code and execution data, can be used to compute mutation testing scores and to collect fine-grained dynamic information about mutant behavior.

Beyond mutation testing, the mutants produced by PyMut4SE can support a variety of software engineering tasks, including program analysis, testing, and learning-based approaches. The tool is currently under active development, with ongoing efforts to extend its mutation capabilities, particularly by introducing operators that target library calls.

While PyMut4SE already enables the efficient generation of large volumes of mutants, we plan to further improve its execution module to enhance scalability and robustness. In addition, we intend to develop a graphical user interface that will allow users to visualize and inspect mutants, analyze their behavior, and execute them with custom inputs. Overall, we envision PyMut4SE as a valuable contribution to the software engineering community, facilitating comprehensive and scalable mutation analysis for Python programs.

Acknowledgements

This work is funded by the European Union (ERC S3, 101093186). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. Repairagent: An autonomous, llm-based agent for program repair. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2188–2200.

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

[3] Lucca Renato Guerino, Pedro Henrique Kuroishi, Ana Cristina Ramada Paiva, and Auri Marcelo Rizzo Vincenzi. 2024. Static and Dynamic Comparison of Mutation Testing Tools for Python. In *Proceedings of the XXIII Brazilian Symposium on Software Quality*. 199–209.

[4] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

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

[6] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118. doi:10.1145/227607.227610

[7] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in computers*. Vol. 112. Elsevier, 275–378.

[8] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verification Reliab.* 25, 5-7 (2015), 605–628. doi:10.1002/STVR.1509

[9] Laura Plein. 2025. Predicting Software Changes from Desired Behavior Changes. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 1019–1021.

[10] Laura Plein, Wendkûuni C Ouédraogo, Jacques Klein, and Tegawendé F Bissyandé. 2024. Automatic generation of test cases based on bug reports: a feasibility study with large language models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 360–361.

[11] Laura Plein, Souhila Zidane, Jordan Samhi, and Andreas Zeller. 2026. Neural Change Prediction: Relating Software Changes to Their Effects and Vice Versa. *arXiv preprint arXiv:2606.03378* (2026).

[12] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient mutation testing for Java. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 297–298.

[13] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: evolving language-based testing. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 894–916.