Argus: A Guided and Traceable Mutation Testing Engine

Zi Yang
UC Riverside
San Jose, USA
zyang253@ucr.edu

Zhaorui Yang
UC Riverside
Riverside, USA
zyang247@ucr.edu

Jiyuan Wang
Tulane University
New Orleans, USA
wjiyuan@tulane.edu

Qian Zhang
UC Riverside
Riverside, USA
qzhang@cs.ucr.edu

Abstract

Mutation testing evaluates test suite quality by injecting intentional faults into source code to verify if tests detect them. Existing engines typically generate mutants by exhaustively applying mutation operators. This creates an implicit blind spot where trivial syntax changes and complex semantic drifts are treated as equally significant. This lack of prioritization wastes computational resources on "easy-to-catch" issues. As a result, the assessment often fails to show whether the test suite is actually capable of catching the most dangerous, subtle bugs that actually matter.

We present Argus, a mutation engine that enhances exhaustive syntactic perturbation with intentional semantic exploration. Our key innovation is modeling mutation as a seed-controlled walk over AST locations, enabling the generation of compositional mutation sequences (*i.e.,* structured chains or trees of dependent edits) that simulate complex regressions rather than isolated, trivial shifts. Argus implements this through three core mechanisms: (1) mutations guided by syntax coverage and semantic properties in tandem, (2) dependency-aware steering toward critical APIs, and (3) an interactive viewer for step-wise traces and mutation trajectories. By allowing developers to interpret and reproduce complex mutation paths, Argus provides a transparent assessment of a test suite's capability to detect deep semantic drifts that existing tools overlook.

CCS Concepts

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

Keywords

Mutation Testing, Software Testing

ACM Reference Format:
Zi Yang, Zhaorui Yang, Jiyuan Wang, and Qian Zhang. 2026. Argus: A Guided and Traceable Mutation Testing Engine . In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3806410

1 Introduction

Mutation testing assesses test adequacy by injecting faults to verify if a test suite can detect them [1]. Existing mutation engines often focus on the loop of generating mutants, running test cases, and computing scores by iterating over valid syntactic locations [2–4]. Because these tools inherently treat trivial syntactic perturbations and complex semantic drifts as equally significant, they operate as black boxes that obscure the most critical regressions. Consequently, it tends to waste computational resources on "easy-to-catch" issues while leaving the test suite's ability to detect subtle, high-value bugs largely unmeasured.

In practice, developers often want to answer a fundamental question: *Which mutants should be generated to most effectively assess a test suite's capability of catching deep, high-value regressions?* They must overcome barriers that existing tools fail to help:

- **Search Blindness**: There is a lack of tools to strategically navigate the mutation generation process toward critical logic. Without guidance, the engine cannot distinguish between trivial edits and changes to critical logics, leading to a combinatorial explosion of insignificant mutants.
- **Traceability Gaps**: There is rarely a clear link between test failures and underlying program logics. When a test fails to catch a mutant, developers lack a logical explanation of the specific property being violated, making it difficult to determine if the test suite is truly deficient.
- **Replication Instability**: Mutation testing is often stochastic. Without reproducable mutants, it is difficult for developers to conduct further diagnosis.

To bridge these gaps, this paper presents Argus, a mutation engine that augments random, syntax-location guided generation with intentional semantic exploration. Our key innovation is modeling the mutation process as a seed-controlled walk over AST locations. Unlike standard tools that produce unordered sets of independent variants, Argus generates compositional mutation sequences (*i.e.,* multi-step trajectories as structured chains or trees of dependent edits) that realistically simulate complex regressions.

Argus uniquely supports (1) mutation generation guided by the interplay between syntax coverage and semantic properties; (2) dependency-aware steering that directs mutation toward critical APIs or user-specified symbols; and (3) an interactive viewer that presents a serialized manifest for exact replay and a step-wise explanation trace that enables developers to interpret and reproduce complex mutation paths.

Argus is open source and ready to use. It can serve as a practical building block for diverse software engineering tasks, such as program analysis, debugging, source-to-source transformation, patching, and code generation, where auditable, explainable source code edits are often needed.

2 Tool Design and Implementation

2.1 Overview

Our tool Argus takes as input a program. It parses the program into an Abstract Syntax Tree (AST), enumerates mutation sites and operators, and performs a guided, seed-controlled walk. It consists

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/26/07
https://doi.org/10.1145/3803437.3806410

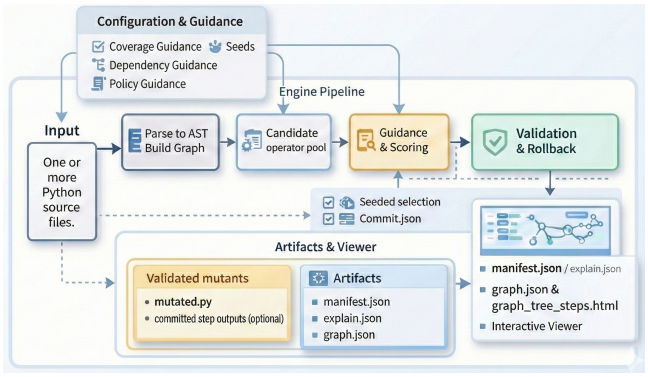

Figure 1: Argus Tool Design

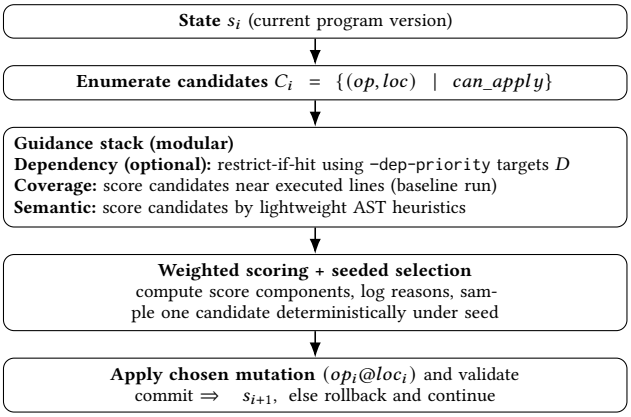

Figure 2: Guided Mutation. Argus identifies applicable candidates, applies a modular guidance stack, then performs weighted scoring and seeded selection to choose the next committed mutation.

of three major components: (1) *guidance*, which determines where and how the search should focus; (2) *scoring and selection*, which ranks candidate mutations and chooses one deterministically under a fixed seed; and (3) an *interactive viewer*, which enables step-by-step replay, inspection, and comparison of mutation trajectories. Argus is currently used for Python programs, but it is general for all programs that can be parsed into ASTs.

2.2 Mutation Guidance

Argus models mutation not as isolated edits, but as a trajectory (*i.e.*, a seed-controlled walk on the program), represented as:

$$s_0 \xrightarrow{op_1@loc_1} s_1 \xrightarrow{op_2@loc_2} \cdots \xrightarrow{op_k@loc_k} s_k$$

where each state s_i is a concrete program version, and each transition applies a mutation at a specific AST location. This abstraction naturally supports higher-order mutants and provides a clear notion of how a mutant evolved. At each step i, the engine enumerates the set of applicable mutator candidates $O(s_i)$ whose `can_apply` predicates hold. A stack of guidance is then applied to prioritize and filter these mutator candidates before a weighted, prioritized selection in Section 2.3. The guidance signals include:

- **Code Coverage.** When available, Argus uses line-level coverage from a baseline execution to identify which parts of the program are exercised.
- **Semantic Heuristics.** Argus derives coarse semantic signals from AST structure (e.g., arithmetic expressions, conditionals, and loops) to characterize which kinds of program behavior are involved.
- **Dependency or Target Region.** Argus optionally incorporates user-specified identifiers or APIs to identify executed code regions that are relevant to a particular analysis focus.

All guidance signals are modular. They can be enabled/disabled independently and reweighted as policies. For transparency, Argus logs per-step score components and reasons so users can inspect why a particular mutation was selected (Figure 2).

2.3 Weighted Scoring and Seeded Selection

To avoid spending efforts on low-impact edits, Argus combines guidance signals into a single score and picks one candidate accordingly. This makes the selection both steerable (by changing which signals are enabled and how they are weighted) and inspectable (by logging the score breakdown).

Scoring. For a candidate c, Argus computes:

$$\text{score}(c) = w_0(c) + \alpha\,\text{cov}(c) + \beta\,\text{sem}(c) + \gamma\,\text{dep}(c),$$

where w_0 is the base operator weight, and the remaining terms are optional guidance signals (coverage/semantic/dependency). The trace records each component for the chosen step. Mutation guidance differs in which signals are enabled and their weights, which are updated on-the-fly or can be controlled by users:

- Random: w_0 only.
- Coverage: $w_0 + \alpha\,\text{cov}(c)$.
- Composite: $w_0 + \alpha\,\text{cov}(c) + \beta\,\text{sem}(c)$
- Dependency: $w_0 + \alpha\,\text{cov}(c) + \beta\,\text{sem}(c) + \gamma\,\text{dep}(c)$. When specific APIs or code regions are provided via `-dep-priority`, Argus boosts candidates that touch the specified targets via $\text{dep}(c)$, concentrating edits around the intended region.

Replication. Argus uses a single Random Number Generator (RNG) stream seeded by the run configuration and deterministic candidate ordering, so identical seed/config yields the same committed walk. In other words, increasing `-steps` extends the same walk while preserving the prefix.

2.4 Interactive Viewer

Argus includes an offline interactive viewer that reconstructs and visualizes a mutation run from exported artifacts, as shown in Figure 1. The viewer loads `manifest.json` to obtain the ordered list of committed steps and run configuration, streams `trace.jsonl` to index per-step explanations, and materializes concrete program versions from `artifacts/`. For each step, the viewer shows the code diff, the applied operator and location, and a concise breakdown of the guidance signals that led to the choice. Steps are pre-indexed, enabling fast navigation during live walkthroughs without rerunning mutation. It also supports side-by-side comparison of multiple runs with different guidance, making the impact of guidance signals easy to inspect and explain.

Run A (committed steps)

```
1 ArithmeticOperatorMutator 12 31
2 ComparisonOperatorMutator 15 71
3 FunctionCallParameterMutator 2 7
...
```

Run B (replay)

```
1 ArithmeticOperatorMutator 12 31
2 ComparisonOperatorMutator 15 71
3 FunctionCallParameterMutator 2 7
...
```

Figure 3: Two independent runs with the same seed and configuration produce identical committed mutation paths.

3 Demonstration and Case Studies

We showcase the usefulness of Argus using four *case studies*:

- **Case 1 (Replication)** shows that a mutation run can be exactly reproduced, enabling reliable replay and inspection.
- **Case 2 (Guidance)** shows how guided mutation surfaces more impactful edits earlier than unguided mutation.
- **Case 3 (Extensibility)** shows that a mutation run can be safely extended with additional steps while preserving previously generated results.
- **Case 4 (Steerability)** shows how users can direct mutation toward specific code regions or APIs.

3.1 Case 1: Replication and Explanation Trace

This case study shows that a mutation run produced by Argus is both reproducible and auditable. For this, we run Argus twice on the same program using identical settings. Although mutation is typically associated with randomness, Argus produces the same committed mutation walk in both runs, selecting the same operators at the same code locations in the same order. The explanation trace recorded for each step is also identical, allowing users to inspect not only *what* changed, but *why* it was selected. For example, Figure 3 shows the traces reported in the `manifest.json` file for running the following command:

```
python -m codemutationengine.demo_walk_cli \
  --input codemutationengine/sample_input.py \
  --policy random --steps 10 --seed 123 --validate
```

We then compute the SHA-256 digest of such exported traces. With identical seed and configuration, both runs produce the same manifest hash:

```
3f491b30ad5dc580725eb8bc8c762814435038eba8690c69d9f2255294cb77c5
```

With Argus enabled replication capability, developers can share a mutation trace, reproduce it exactly, and reason about each decision using the recorded explanation signals, rather than treating mutation as an opaque or irreproducible process.

3.2 Case 2: Guidance

This case shows that guidance meaningfully changes *where* mutation effort is spent: with the same seed and step budget, guided mutation focuses earlier on executed and decision-critical code, while unguided mutation does not. For this, we run Argus twice on the same program with identical seeds and budgets. The first run uses a Random policy, while the second uses the Composite policy that combines coverage and lightweight semantic guidance.

```
# Random baseline (same seed/budget)
python -m codemutationengine.demo_walk_cli \
  --input codemutationengine/sample_input.py \
  --walk guided --policy random --steps 10 --seed 7

# Composite policy (same seed/budget)
python -m codemutationengine.demo_walk_cli \
  --input codemutationengine/sample_input.py \
  --walk guided --policy composite --steps 10 --seed 7
```

Step	**Random** (op @ line)	**Composite** (op @ line)	**Composite score)**			
			Total	**Cov.**	**Sem.**	**Dep.**
1	ComparisonOperator @ 15	DataStructureMutator @ 13	3.10	1.00	0.50	0.00
2	NumberSignFlip @ 17	DataStructureMutator @ 13	2.85	1.00	0.35	0.00
3	StatementDeletion @ 22	BooleanConditionFlip @ 13	3.40	1.00	0.80	0.00
4	DataStructureMutator @ 13	DataStructureMutator @ 13	2.70	1.00	0.20	0.00
5	VariableRenameMutator @ 12	NumberSignFlipMutator @ 14	3.05	1.00	0.45	0.00

Table 1: Case 2 (seed=7, steps=10): first five committed steps under Random and Composite. The rightmost columns show how different guidance signals combine into the total score used to rank mutation choices.

We use two artifacts that are produced automatically by the tool: (i) the committed step list in `manifest.json`; and (ii) the per-step score breakdown in `trace.jsonl` for Composite. To keep the comparison concrete, Table 1 shows the first five committed steps side-by-side. For Composite, we additionally report the logged score components (`total`, `coverage`, `semantic`, `dependency`); since this run does not use targets, `dependency` should be zero throughout.

As shown in Table 1, the Composite policy consistently selects mutations on executed lines and control-flow constructs (e.g., branch conditions) in the early steps. In contrast, Random selects edits without regard to execution relevance, often mutating peripheral or low-impact locations. Guidance steers mutation toward higher-leverage edits. This means fewer wasted steps and earlier exposure of behaviorally significant changes, while retaining full transparency through per-step score breakdowns recorded in the explanation trace.

3.3 Case 3: Extensibility

This case shows that mutation runs produced by Argus are safely extensible: increasing the step budget grows the mutation path without altering previously generated results. For this, we run Argus multiple times on the same program using the same seed and configuration, varying only the step budget (`–steps 1/2/3`). Each run produces a committed mutation `path` recorded in `manifest.json`. When the budget increases, new mutations are appended to the end of the path, while all earlier committed steps remain unchanged.

```
python -m codemutationengine.demo_walk_cli \
  --input codemutationengine/sample_complex.py \
  --walk guided --policy composite --seed 7 --validate \
  --steps 1 or 2 or 3
```

Table 2 shows that the path generated with a smaller budget is an exact prefix of the path generated with a larger budget. For example, the first step produced with `–steps 1` is identical to

Budget	Exported committed path	Prefix check
`-steps 1`	`1:BooleanConditionFlipMutator @ 56 (180)`	N/A
`-steps 2`	`1:BooleanConditionFlipMutator @ 56 (180)` `2:BooleanConditionFlipMutator @ 44 (112)`	steps(1)→steps(2)

`-steps 3`	`1:BooleanConditionFlipMutator @56 (180)`
	`2:BooleanConditionFlipMutator @44 (112)`
	`3:ParamRenameMutator @169 (771)`
		steps(2)→steps(3)

Table 2: Case 3: Prefix-consistent extension under increasing mutation efforts.

	Without `-dep-priority`	With `-dep-priority`
Committed path	1: BCFlip	1: BCFlip
	2: DS	2: ArithOp
	3: VarRen	**3: ParamRen**
	4: CallArg	4: CallArg
	5: StmtDel	**5: NumSign**
	6: VarRen	**6: VarRen**
	7: CompOp	**7: VarRen**
	8: VarRen	**8: ParamRen**
	9: ParamRen	**9: VarRen**
	10: DS	**10: ParamRen**
Target-touch@10	**0/10**	**8/10**

Table 3: Case 4 (seed=99, steps=10): committed mutations. Bold steps match the specified targets (`discount`, `fraud_risk_adjustment`).

the first step in the `-steps 2` and `-steps 3` runs, and the first two steps of `-steps 2` match those of `-steps 3`. Prefix-consistent extension allows users to start with short, inexpensive runs and later extend them without losing earlier results. This makes mutation-based exploration practical for iterative analysis, debugging, and demonstrations where budgets evolve over time.

3.4 Case 4: Steerability

This case shows that users can steer mutation toward code regions of interest: with the same seed and step budget, enabling dependency guidance concentrates edits on user-specified, non-trivial identifiers or APIs. For this, we run Argus twice on the same program with identical seeds and budgets, enabling `-dep-priority` only in the second run. The target identifiers (`discount` and `fraud_risk_adjustment`) represent regions a user wishes to focus on. All other settings remain unchanged. The commands are as follows:

```
python ...  --steps 10 --seed 99 --validate
python ... --steps 10 --seed 99 --validate
  --dep-priority discount fraud_risk_adjustment
```

Table 3 compares the exported committed paths (reported as `index: operator @ line (node_id)`). We mark in **bold** the steps whose recorded edit touches the dependency targets (by identifier/API match). Without dependency guidance, none of the committed steps touch the targets. With `-dep-priority` enabled, most committed steps do.

4 Related Work

Mutant Prioritization. In mutation testing, the challenge is often not generating mutants but deciding which ones matter. Prior work and industrial practice often prioritize mutants using program context, coverage, and learned signals [2, 5–8]. Our tool treats prioritization as an interactive, step-wise process that users can steer and compare under a fixed seed and budget.

Context-Aware Mutation. Many mutants are trivial or invalid, and their usefulness is highly context-dependent [6]. Rather than learning specialized operators, we keep operators general and apply lightweight semantic filters and staged validation to discard low-yield outcomes early, while explicitly recording when these filters affect decisions.

Explanations in Mutation Testing. When mutation testing is part of CI and regression workflows, reproducibility is critical [2, 3, 9, 10]. We therefore make explainability and replayability first-class outputs, exporting per-step explanations and a replayable manifest that enable auditable comparisons and controlled ablations.

5 Conclusion

We presented Argus, a new mutation engine that rethinks mutation generation as a seed-controlled walk over a program graph. Instead of treating mutation as a one-shot sampling, it enables live walkthroughs, replicable mutation traces, and fine-grained inspection of how and why each mutation is produced.

6 Availability

Argus is available as open-source: https://github.com/ucr-riple/Argus. The demonstration video is available at https://youtu.be/zLrDZpViLw0.

7 Acknowledgement

The authors of this work are in part supported by NSF CCF-2426161, UCR Senate Awards, and Tulane University Startup Funding. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Jeongju Sohn, Ezekiel Soremekun, and Michail Papadakis. Latent mutants: A large-scale study on the interplay between mutation testing and software evolution, 2025.
[2] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 163–171, 2018.
[3] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2022.
[4] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Does mutation testing improve testing practices? In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 910–921. IEEE Press, 2021.
[5] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. Prioritizing mutants to guide mutation testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2022.

[6] René Just, Bob Kurtz, and Paul Ammann. Predicting mutant utility with program context. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2017.

[7] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 342–353, 2016.

[8] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry—a study at facebook. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 268–277, 2021.

[9] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Mutation testing in evolving systems: Studying the relevance of mutants to code evolution. *ACM Transactions on Software Engineering and Methodology*, 32(1), 2023.

[10] Zun Zeng, Yanjie Huang, Fabiano Pecorelli, Liguo Wang, and David Lo. Mutation-guided assessment of test suite porting for ci acceleration. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2024.