

Mutation Testing: Who Tests the Tests?

Measuring what a test suite verifies, not just what it executes

Adam Daw

October 2026

ABSTRACT

Code coverage reports which lines a test suite executed, but not whether the tests would fail if those lines were wrong. Mutation testing addresses this gap by injecting small, deliberate faults (*mutants*) into the code and checking whether the suite detects each one. This paper introduces the technique and grounds it in the author's own measurements: three small JavaScript files with 100% line, branch and function coverage whose mutation scores are 0%, 81.82% and 66.67%. It explains how to read a mutation score and why the score depends on the operator set, summarises the evidence that mutants stand in for real faults, and presents a case study of investigating surviving mutants in three manual audits. It closes with practical guidance on adopting the method.

1 Introduction: the limits of coverage

A green test suite is reassuring, but it answers a narrower question than it appears to. The question a team actually cares about is: *if this code were wrong, would the tests go red?* The most common proxy for that question is code coverage, which records which lines, branches or functions were executed while the tests ran. Coverage is cheap to measure and easy to report, and a low figure is genuinely informative: code that never runs under test is certainly not being checked.

A high figure is much less informative. Listing 1 is deliberately exaggerated: the single test calls `is_adult` and discards the result. Every line of the function executes, so line coverage is 100%, yet the test contains no assertion and would pass whatever the function returned.

```
def is_adult(age):  
    return age >= 18  
  
def test_is_adult():  
    is_adult(30)          # no assert!
```

Listing 1. Full line coverage with no verification.

Real suites rarely look this bare, but weaker versions of the same pattern are common: tests that only assert that no exception was raised, tests that assert `len(result) > 0` rather than what the result should be, and snapshot tests whose recorded output nobody reads. In each case the code runs and the coverage report counts it, but very little is actually checked.

The gap is not confined to contrived examples. Table 1 shows measurements the author recorded on three small JavaScript files, each with its own tests [1]. All nine tests pass, and every file has 100% line, branch and function coverage. Running the mutation testing tool Stryker 10.0.0 over the same files with its full operator set produced 30 mutants, of which 15 were killed and 15 survived. The per-file scores range from 0% to 81.82%; on `alerts.js`, a file the coverage report marks as fully exercised, the tests detected none of its ten mutants. The run was recorded on Node 24.18.0 and later reproduced exactly on Node 24.21.0 with the same Stryker version.

Table 1. Coverage and mutation results for the author’s three examples [1] (Stryker 10.0.0, full operator set).

File	Line / branch / function coverage	Mutants	Killed	Survived	Score
alerts.js	100% / 100% / 100%	10	0	10	0.00%
dates.js	100% / 100% / 100%	11	9	2	81.82%
discount.js	100% / 100% / 100%	9	6	3	66.67%
Total		30	15	15	50.00%

Coverage tells us what code *ran*; it does not tell us what the tests *checked*. Mutation testing is a way of measuring the second thing directly.

2 Mutation testing

The idea is simple to state. Make a small, deliberate change to the program under test, producing a variant called a *mutant*, and run the existing test suite against it. If at least one test fails, the suite has noticed the change and the mutant is said to be *killed*. If every test still passes, the mutant has *survived*: the code was changed and no test noticed.

The technique needs no tooling to start. Done by hand, it is “two test runs, on the one line you already doubt. You need no tooling and nobody’s permission” [1]: one run with the line mutated, and one with it restored, to confirm the suite returns to green. That makes manual mutation a cheap, targeted check on a specific guard or comparison during development or review. A mutation testing tool automates the same loop at breadth: it generates mutants systematically across the code, runs the suite against each, and reports which survived. In both forms each mutant contains exactly one small change, so a surviving mutant points at one specific place and one specific kind of mistake.

The kinds of change a tool makes are defined by its *mutation operators*. Table 2 lists a representative set. Each operator mimics a small slip a programmer might plausibly make: an off-by-one comparison, a wrong arithmetic sign, an inverted condition, a mistyped constant, a missing return value or a dropped side-effecting call.

Table 2. Common mutation operators. Each mutant applies exactly one such change.

Operator	Original	Mutant
Relational	a >= b	a > b
Arithmetic	a + b	a - b
Logical	a and b	a or b
Negation	if ok:	if not ok:
Constant	18	19
Return value	return x	return None
Statement deletion	save(user)	(removed)

3 Worked example

Listing 2 repairs the test from Listing 1 by adding assertions. It now checks one clearly adult age and one clearly non-adult age, which is the kind of test many developers would write first.

Table 3 shows four mutants of `is_adult` and the outcome of running `test_is_adult` against each. Inverting the comparison to `age <= 18` makes `is_adult(30)` return `False`, and replacing the body with `return True` makes `is_adult(5)` return `True`; both are killed. The other two survive. Changing `>=` to `>`, or the constant 18 to 19, alters the function’s behaviour only for inputs at or next to the boundary, and the test never exercises those inputs.

```
def is_adult(age):
    return age >= 18

def test_is_adult():
    assert is_adult(30) is True
    assert is_adult(5) is False
```

Listing 2. `is_adult` with a plausible first test.

Table 3. Mutants of `is_adult` run against `test_is_adult`.

Mutant	Result
<code>age <= 18</code>	killed
<code>return True</code>	killed
<code>age > 18</code>	survived
<code>age >= 19</code>	survived

The two survivors point at the same gap: nobody tested the boundary. Under the `age > 18` mutant an 18-year-old is no longer an adult, and under `age >= 19` the same is true; in both cases the tests remain green. The fix the survivors ask for is the test in Listing 3, which pins the behaviour on both sides of the threshold. With it added, every mutant in Table 3 is killed.

```
def test_is_adult_boundary():
    assert is_adult(18) is True
    assert is_adult(17) is False
```

Listing 3. The boundary test suggested by the surviving mutants.

It is worth being precise about what a survivor means. It is not a failure, and it does not show that the code is wrong; the original `is_adult` was correct throughout. A survivor is better read as a question: *did you mean for this behaviour to be untested?* Usually it points at a test gap, as here. But not always: some mutants change the code without changing its behaviour, and no test can kill those (Section 6.2). A survivor is therefore either a test gap or an equivalent mutant, and telling the two apart is part of reading it.

4 Mutation score and how to read it

Results across many mutants are summarised as a *mutation score*, but the term has two definitions in use. The academic definition, as set out in Jia and Harman’s survey of the field [2], divides the mutants killed by the number of *non-equivalent* mutants, so that mutants no test could kill do not count against the suite:

$$\text{academic score} = \frac{\text{killed}}{\text{non-equivalent mutants}} \times 100 \quad (1)$$

A tool cannot compute that denominator, because deciding whether a mutant is equivalent is undecidable in general (Section 6.2). Tools therefore report something different. Stryker documents its score as detected mutants over valid mutants [3]:

$$\text{Stryker score} = \frac{\text{detected}}{\text{valid}} \times 100 \quad (2)$$

Any equivalent mutants remain in the tool’s denominator, which means that wherever they exist the reported score cannot reach 100% however good the tests are. The two numbers share a name but measure different things, and a reported score should be read with the tool’s definition in mind.

A tool’s score also depends on which operators it applies. The author re-ran the examples of Table 1 with only Stryker’s `EqualityOperator` enabled: the same source and the same tests, with only the operator set changed [1]. That run produced 7 mutants, of which 3 were killed and 4 survived, for an overall score of 42.86%, against 50.00% from 30 mutants under the full operator set. Nothing about the tests changed between the two runs; only the population of mutants did. A score is therefore meaningful only alongside the operator set and mutant count that produced it, and two scores from different configurations should not be compared as if they measured the same thing.

Even within one configuration, the score is a useful signal of test *strength* but a poor target. Treating 100% as a goal invites tests written to kill particular mutants rather than to express what the code is meant to do, which produces a suite tuned to the tool instead of to the requirements. The score is most valuable as a pointer to where to look. The real information is in the individual survivors: each one names a line and a specific change that went unnoticed, and reading them one at a time is the best use of the technique.

5 Why it works

Mutants are deliberately tiny, while real bugs can be anything. Why should a suite’s ability to catch one-token changes say anything about its ability to catch the faults that actually occur? The idea of mutation testing was proposed by Richard Lipton around 1970–71, and its classic published treatment is the 1978 article by DeMillo, Lipton and Sayward [4], which set out the two ideas that underpin the method.

The first is the *competent programmer hypothesis*: real code is usually *nearly* right, so real bugs tend to be small deviations from a correct program rather than wholly wrong programs. If that holds, small syntactic changes are a reasonable model of the mistakes that occur in practice. The second is the *coupling effect*: tests that catch simple faults tend to catch complex ones too, because a test sensitive enough to notice a one-token change is usually sensitive to larger changes in the same behaviour.

Whether mutants are a valid substitute for real faults has also been studied empirically. Just et al. [5] examined 357 real faults and found that a test suite’s detection of mutants correlates with its detection of real faults, and that this relationship holds even when controlling for code coverage. The same study also marks the method’s limits: 73% of the real faults were coupled to the generated mutants, but 27% were not. Of all 357 faults, about 10% would require new or stronger mutation operators, while 17% were not coupled to any mutant at all, a more fundamental limit of the approach; together these make up the uncoupled 27%. Mutation testing is therefore strong evidence about a suite, but not a complete account of every fault it might miss.

6 Limitations

6.1 Cost

The basic automated procedure runs the test suite once per mutant, so its cost is roughly the number of mutants multiplied by the suite’s runtime. For a large codebase with a slow suite, naive whole-repository mutation testing quickly becomes impractical. Tools offer ways to reduce the cost. Stryker,

for example, can use coverage analysis to select tests: with `coverageAnalysis` set to "off" it runs all tests for every mutant, while its per-test mode runs only the tests that cover the mutated code [6].

For scale, the author’s recorded timings for the tiny examples of Table 1 are 0.17 s for the coverage run, 0.94 s for the full mutation run and 0.74 s for the reduced run, each the median of three consecutive whole-command runs on an Apple M5 Pro [1]. These figures describe three small files with fast tests on one machine; they are not a prediction for real projects. Targeted manual mutation avoids the whole-repository scaling problem by limiting the exercise to the line in doubt.

6.2 Equivalent mutants

Some mutations change the code without changing its behaviour. In Listing 4, a loop that starts at `i = 0` and adds 1 on each pass behaves identically whether its condition is `i < 10` or `i != 10`. Under that initialisation and step, `i` takes only the values 0 to 10, so both conditions first become false at exactly `i = 10`; the two conditions disagree only for values above 10, which `i` never reaches. Without the initialisation the mutant would not be equivalent: started above 10, the `!=` version would never terminate. Such an *equivalent mutant* cannot be killed by any test. Jia and Harman state the underlying difficulty directly: “To detect if a program and one of its mutants programs are equivalent is undecidable” [2]. Automated detectors therefore find only a fraction of equivalent mutants; as one data point from their survey, a constraint-based detector found 47.63% of the equivalent mutants across 11 programs [2]. In practice, some survivors turn out on reading to be equivalent, and dismissing them requires human judgement or, as Section 7 shows, a deliberate comparison.

<code># original</code>	<code># mutant: same behaviour, cannot be killed</code>
<code>i = 0</code>	<code>i = 0</code>
<code>while i < 10:</code>	<code>while i != 10:</code>
<code>i += 1</code>	<code>i += 1</code>

Listing 4. An equivalent mutant.

6.3 Noise

Not every surviving mutant is worth acting on. Mutants in logging statements, default values and boilerplate often survive because nobody would reasonably test that behaviour. Left unfiltered, they bury the useful survivors in a long report that developers learn to ignore.

7 Case study: investigating survivors in manual audits

The author used manual mutation to check work on GitNexus, a publicly available code intelligence engine created by Abhigyan Patwari [7] that indexes source code into a graph of symbols and the edges between them; the audited work was on the author’s own fork, which adds support for Salesforce Apex [8]. Three targeted *manual* audits were carried out: each mutant was chosen by hand, applied, run against the relevant tests and then reverted by hand. Table 4 summarises the results. These are counts from hand-selected mutants, not automated mutation scores, and because every mutant was reverted after its run, the committed records rather than the current code are the evidence for them.

Table 4. Results of three targeted manual mutation audits [8].

Audit	Mutants applied	Killed	Survived	Survivor disposition
1	8	8	0	none
2	15	15	0	none
3	15	11	4	no observable difference on the audited fixture

7.1 *Checking survivors for an observable difference*

The third audit left four survivors. Rather than argue equivalence from reading the code alone, the author compared outputs: for each survivor, the graph the tool produced from that site’s own test fixture with the mutant applied was checked against the graph produced without it. An identical graph shows that the mutation produced no observable difference on that fixture; it does not establish semantic equivalence for every possible input. A comparison like this is only trustworthy if it can detect a difference when one exists, so the harness was first validated with a positive control: a mutant known to matter (disabling the resolved return of one resolution step) was run through it and correctly produced a graph difference. With the instrument validated, all four survivors produced identical graphs, showing no observable difference on the audited fixture. On that basis the audit justified dismissing them as fixture-equivalent, within that fixture’s scope rather than globally.

The same discipline applies well beyond this project. An “identical” result from a comparison that has never been shown to detect a difference says more about the comparison than about the mutant.

7.2 *What a negative assertion cannot kill*

The audits also produced a lesson about assertions. An assertion that expects *no* edge to be produced cannot kill a mutant that also produces no edge on the same input: both the correct code and the mutant satisfy it. In the third audit, the fixture that killed such a mutant (`PCollField.cls`) was one on which the mutant emits an *extra, incorrect* edge, a difference the tests can then detect. The general lesson is that an assertion expecting nothing to happen cannot kill a mutant that also does nothing on the same input; killing it requires an input on which the mutant does something wrong and visible.

8 Practice

8.1 *Start by hand*

The cheapest entry point is the manual form: pick the line you doubt, mutate it, run the suite, restore it and run again. The author’s guide puts the cost plainly: “By hand: two test runs, one mutated and one restored, plus the few seconds of thinking that step 2 takes” [1], where that step is naming, before running anything, the test expected to fail. This requires no tool adoption and fits naturally into writing or reviewing a change.

8.2 *Mutate only changed code*

For automated runs, the most effective way to control cost is to mutate only the code that has changed, and to run the analysis as part of code review rather than over the whole repository. This keeps the number of mutants proportional to the size of the change and puts each survivor in front of the person best placed to judge it. Petrović and Ivanković [9] describe this approach at Google: diff-based, probabilistic mutation analysis that omits uncovered and “arid” lines, and whose surviving mutants are surfaced to developers during code review as automated “findings”, used by approx-

imately 6,000 engineers as part of mandatory code review. They also report that computing an absolute, repository-wide mutation score was infeasible, and that they “were also unable to find a good way to surface it to the engineers in an actionable way” [9].

8.3 Prioritise critical modules and skip the noise

Where whole-module analysis is wanted, start with the code whose failures matter most, such as pricing, authentication and parsing, rather than attempting everything at once. Exclude logging and generated code from mutation so that reports contain fewer noise survivors (Section 6.3). Because a full run can be slow, run it nightly or on demand rather than on every commit.

8.4 Read survivors; do not chase the score

As Section 4 argued, the score is a signal and the survivors are the substance. For each survivor, ask whether the unnoticed behaviour change matters. If it does, write the test that would have caught it, as Listing 3 did; if an assertion that expects nothing to happen cannot catch it, look for an input on which the mutant produces something visibly wrong (Section 7.2). If the mutant appears equivalent, check it with a validated comparison (Section 7.1) or dismiss it with a recorded reason; if it sits in noise, exclude it.

8.5 Tools

Table 5 lists mutation testing tools for several languages. PIT describes itself as “a state of the art mutation testing system” for Java and the JVM [10]; Stryker is the tool used for the author’s examples in Table 1. Some tools address cost directly. One example is apex-mutation-testing, a third-party plugin for Salesforce Apex by Sébastien Colladon [11]. Its documentation lists 26 mutation operators and describes an optional graph-colouring grouping of mutants whose covering tests are disjoint, computed with a clique lower bound, the DSATUR heuristic and exact backtracking [11].

Table 5. Mutation testing tools by language.

Language	Tool
Java / JVM	PIT (pitest)
JS / TS, C#, Scala	Stryker
Python	mutmut, Cosmic Ray
Rust	cargo-mutants
PHP	Infection
C / C++	Mull
Salesforce Apex	apex-mutation-testing

9 Exercise for the reader

Listing 5 gives a function that applies a 10% discount to totals above 100, together with a test. The test passes and executes every line of `discount`. Using the operators in Table 2, find a mutant of `discount` that survives this test, and then write the test that would kill it. A suggested answer is given in Appendix A.

```
def discount(total):
    if total > 100:
        return total * 0.9
    return total

def test_discount():
    assert discount(200) == 180
    assert discount(50) == 50
```

Listing 5. Exercise: find a mutant that survives `test_discount`.

10 Conclusion

Coverage measures execution; mutation testing measures verification. The author’s own examples show the difference plainly: three files with complete line, branch and function coverage score between 0% and 81.82% under mutation. By asking whether the test suite notices small, deliberate faults, the technique turns the vague question of whether tests are “good enough” into a list of specific survivors, each of which is either a test gap to close or an equivalent mutant to justify. Its costs are real, and it does not model every kind of fault, but they can be kept manageable: start by hand on the line you doubt, mutate changed code, focus on critical modules, and read the survivors rather than chasing the score.

References

- [1] A. Daw. Mutation testing, in five minutes (guide, `guides/mutation-testing.md`) and runnable examples with recorded results (`examples/README.md`). *claude-code-skills* repository, 2026. <https://github.com/adamdaw/claude-code-skills>
- [2] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi:10.1109/TSE.2010.62
- [3] Stryker Mutator. Mutant states and metrics. Documentation, accessed 7 October 2026. <https://stryker-mutator.io/docs/mutation-testing-elements/mutant-states-and-metrics/>
- [4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978. doi:10.1109/C-M.1978.218136
- [5] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are Mutants a Valid Substitute for Real Faults in Software Testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 654–665, 2014. doi:10.1145/2635868.2635929
- [6] Stryker Mutator. StrykerJS configuration. Documentation, accessed 7 October 2026. <https://stryker-mutator.io/docs/stryker-js/configuration/>
- [7] A. Patwari. GitNexus. GitHub repository, accessed 7 October 2026. <https://github.com/abhigyanpatwari/GitNexus>
- [8] A. Daw. Gate 5 pass records for work items 2–4 (`gate5-wi2.md`, `gate5-wi3.md`, `gate5-wi4.md`), on the author’s Apex fork of GitNexus [7], 2026. <https://github.com/adamdaw/GitNexus/tree/main/.vsdd/pass-records>
- [9] G. Petrović and M. Ivanković. State of Mutation Testing at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2018)*, pages 163–171, 2018. doi:10.1145/3183519.3183521
- [10] PIT. PIT mutation testing. Project website, accessed 7 October 2026. <https://pitest.org/>
- [11] S. Colladon. apex-mutation-testing. GitHub repository (README), MIT licence, accessed 7 October 2026. <https://github.com/scolladon/apex-mutation-testing>

Appendix A Exercise answer

Several mutants of `discount` survive `test_discount`. Changing the comparison `total > 100` to `total >= 100` survives, because the test never calls `discount(100)`, the one input where the two conditions disagree. Changing the constant 100 to 99 or to 101 also survives, since neither 200 nor 50 lies near the threshold. By contrast, changing the rate 0.9 to 0.8 is killed: `discount(200)` becomes 160 rather than 180, and the first assertion fails.

The fix is the same lesson as the `is_adult` example in Section 3: test the boundary. Listing 6 adds assertions on both sides of the threshold, which kill all three surviving mutants.

```
import pytest

def test_discount_boundary():
    assert discount(100) == 100
    assert discount(101) == pytest.approx(90.9)
```

Listing 6. Boundary tests that kill the surviving `discount` mutants.