Your tests pass. That is not the same as your tests working.

TestGuard takes the promises your code makes, breaks each one on purpose, and reports every promise your tests did not notice breaking.

THE PROBLEM

Coverage tells you a line ran. It never tells you anyone checked the result. So a codebase can be fully covered and completely undefended — and nothing in CI will say a word.

A REAL TEST, FROM THIS PROJECT'S OWN FIXTURE

```
expect(store.writeAudit).toHaveBeenCalledWith(
  expect.objectContaining({ action: 'MASK', scope: 'g1', ruleCount: 1 }),
);  // `content` is never named — so nothing checks it
```

`objectContaining` ignores keys it does not list. Swap the redacted text for the **raw secret** and this test still passes. Coverage of that line: 100%. The audit log now leaks the very thing it exists to protect.

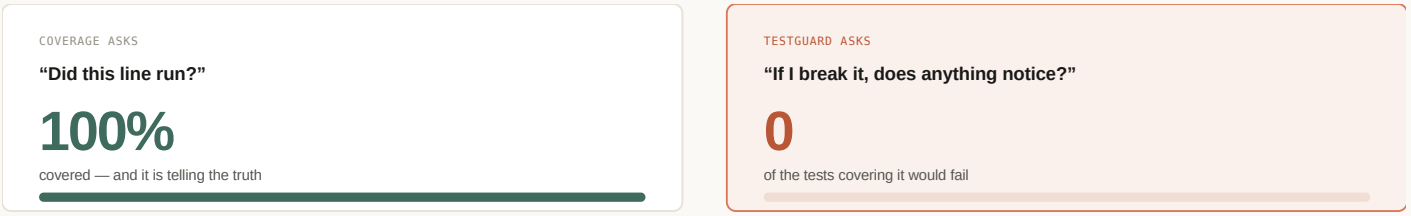


FIG. 1 — THE SAME LINE OF CODE, UNDER TWO DIFFERENT QUESTIONS. ONLY ONE OF THEM IS WORTH KNOWING.

THE METHOD

killed

You broke it — a test failed. **Good.** That behaviour is genuinely defended.

SURVIVED

You broke it — everything stayed green. **A blind spot.**

The word trips everyone once: it is the *fault* that survived, not the test. SURVIVED is the bad news. A healthy report is full of *killed*.

A tool that reports everything as caught is worse than no tool at all — because nobody questions good news.

THE DESIGN CONSTRAINT EVERYTHING ELSE FOLLOWS FROM

TWO INDEPENDENT RUNS, TWO AI-AUTHORED CODEBASES

Neither suite was sloppy. One had ~4,900 disciplined tests, no snapshots, and 0.4% zero-assertion. Both were green. Both were substantially blind.

FIELD REPORT A — AI-AUTHORED PRODUCTION CODEBASE, ~4,900 TESTS

8 of 9
real historical bugs the suite could not see

2,451
tests green, at once, on known-broken code

100%
coverage on the compliance path that leaked

FIELD REPORT B — A DIFFERENT CODEBASE, 458 TESTS, 24 SECURITY CLAIMS, 39 FAULTS INJECTED

BEFORE — EVERY TEST GREEN



21 faults SURVIVED a fully green suite

9 of them critical — super-admin gating, cookie flags, the whole authorization callback

AFTER — TESTS WRITTEN AGAINST THE SURVIVORS



39/39 killed

The suite was not bad.
It was unexamined.

KILLED — DEFENDED SURVIVED — A BLIND SPOT

FIG. 2 — MEASURED, NOT MODELLED. EACH DOT IS ONE INJECTED FAULT.

THE SHAPE OF THE FAILURE

It is not missing tests

One file had re-implemented the authorization logic *inside the test* and asserted against its own copy. Fifteen tests, all green, all passing for the wrong reason — the production path could be deleted outright without one of them noticing.

It is assertions that name too little

The largest gap ran through a compliance-critical path at 100% coverage, where the one assertion that mattered used `objectContaining` and omitted the field carrying the data. Exactly the exhibit on page one.

A green suite is evidence that the tests agree with the code. It is not evidence that either one is right.

WHY THE QUESTION HAS TO BE ASKED ADVERSARIALLY

ONE FAULT, START TO VERDICT

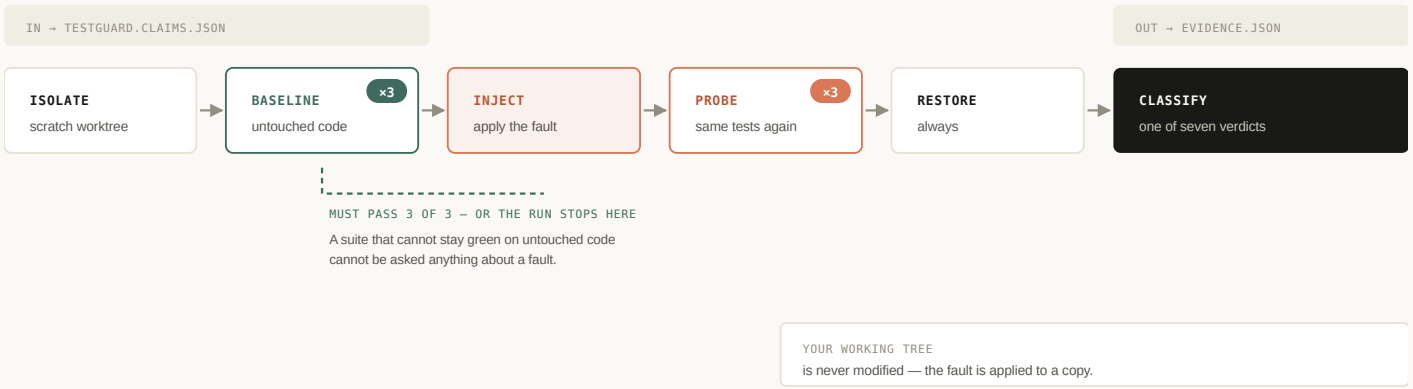


FIG. 3 — THE PROBE LOOP. EVERY STEP REPEATS PER FAULT; THE BASELINE IS CACHED PER DEFENDER SET.

HOW A VERDICT IS DECIDED

The order is the specification. Each check short-circuits the ones below it, so the tool never reaches an optimistic answer through a question it should not have asked.



FIG. 4 — VERDICT ORDER. EVERY AMBIGUITY RESOLVES DOWNWARD, TOWARD “UNPROVEN”.

WHAT A RUN COSTS, AND WHY IT IS PREDICTABLE

Wall clock is not a mystery. For each claim:

```
(baseline runs + faults × N) × cost of the
                    claim's defender set
```

Measured, not estimated: `testguard.cost-budget.json` records what each set actually cost in CI, and `--cost` reports the bill per claim and per file.

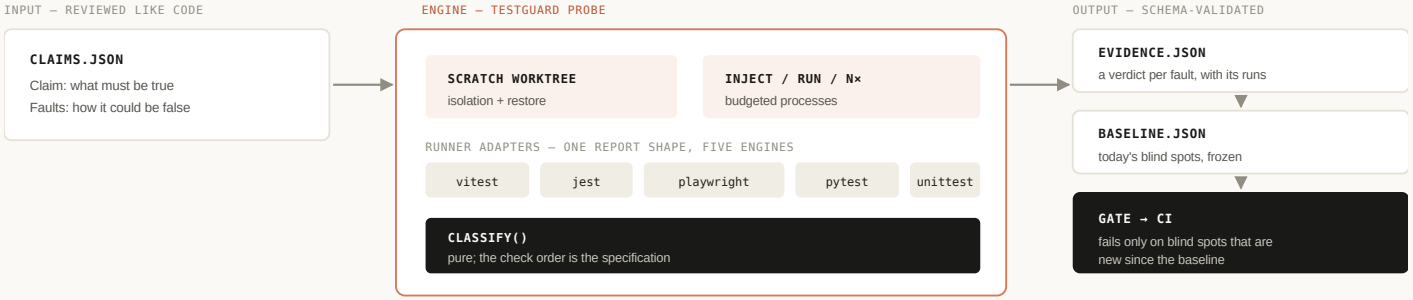
The defender *set* is the unit, not the file — a run executes all of a claim's tests at once. So one slow acceptance test named by five claims is paid for thirty times over, and the report names that file rather than leaving you to guess.

Baselines are cached per defender set, and a verdict is reused when the source, the defenders *and* the fault are all unchanged.

SEVEN VERDICTS, BECAUSE “DID THE TESTS FAIL?” IS A SLOPPY QUESTION

VERDICT	WHAT IT MEANS
`killed`	A test body ran and rejected the behaviour. The only good outcome.
SURVIVED	Everything passed. A real blind spot in the tests.
NOCOVER	No test even looks at this code. Not “weak tests” — *no* tests.
UNVERIFIABLE	The fault could not be applied: its anchor moved, or matches twice. Nothing was learned.
FAULT-INVALID	The break itself was broken — it did not compile. Our fault, not yours.
TIMEOUT	The suite hung. A hang is not a detection.
FLAKY-DEFENDER	The tests are not reliable enough on untouched code to be asked the question.

WHERE THE PIECES SIT



WHY THE BASELINE MATTERS

A legacy codebase does not start with four hundred failures. Freeze what is already blind, then gate only on what gets worse — so the tool is adoptable on a Monday instead of being switched off on a Tuesday.

FIG. 5 — CLAIMS IN, EVIDENCE OUT, AND A GATE THAT MEASURES THE DELTA RATHER THAN THE DEBT.

EVERY AMBIGUITY ROUNDS TOWARD “UNPROVEN”

- Three runs, never one.
- A green baseline before any fault is injected.
- A timeout is never a kill.
- A suite that fails to load is never a kill.
- Mixed results across runs are not a kill.
- A test that mocks what it defends is not a defender.

THE ENGINE IS ONE VERB. THE LOOP IS THE PRODUCT.

Probing proves a claim. It does not tell you which claim to write next, whether a new test earns its place, or whether the agent editing your code knows any of it.

VERB	WHY IT EXISTS
`status`	Where the project is and the *one* next action. `--json` is the machine entry point.
`init`	Installs the agent layer — skill, session-start hook, AGENTS.md section — at the git root.
`claims`	Lists the claims and reports drift against the `@claim` annotations in the code.
`scaffold`	Proposes faults mechanically for one file, as a draft claims document. A starting point, not an answer.
`sweep`	The cold start: probe files with *no* claim — a diff, the write surface, or a one-line `concern`.
`probe`	The engine. Inject each fault, run its defenders, report what survived.
`admit`	The two-gate rule as one verb: green on HEAD *and* failing on every fault of the claim. Otherwise the test is not a defender.
`baseline`	Freezes today's unproven findings so only new ones gate.
`gate`	Fails when a changed source file carries no claim and no excusing entry. Unclaimed code is the blind spot before the blind test is.
`brief`	Emits the blind-spot block for an agent's session-start context.
`mcp`	Serves status, brief, claims and evidence read-only over MCP, so the loop works in any agent harness.
`replay`	Would this suite have caught the bugs that already escaped? *Reports, never gates.*

THE HONEST QUESTION: DOES AN INJECTED FAULT RESEMBLE A REAL BUG?

Every mutation tool has to answer this and most decline to. `replay` answers it empirically: it walks real fix commits out of your history, reverts each fix, runs the tests that exist *now*, and records whether they notice.

CALIBRATION — A MISS RATE, SO A HIGH P IS BAD

```
guard-removed  missed 7/9  p=0.78  ci [0.45, 0.94]
field-dropped  missed 4/4  p=1.00  ci [0.51, 1.00]
```

Read as: *when a real bug of this class escapes, how often does the suite miss it?* Each bug is labelled with the fault class its diff most resembles — the join key that turns an uninterpretable score into a statement with a sample size. Every `p` and interval is recomputable from `n`, and the validator recomputes them.

- **Measurements enter the ratio:** `caught`, `blind`, and `nocover` — the last as a miss, because no tests at all must not score better than weak ones.
- **Failed measurements enter neither side:** `flaky` and `unverifiable` conclude nothing, and are not allowed to flatter the number.
- **Flake is biased pessimistically:** a flaky failure would read as detection, so mixed runs are never `caught`.

THE OPEN QUESTION THIS INSTRUMENT EXISTS TO ANSWER

Does a calibration learned on a repository *with* history transfer to a greenfield one that has none? AI-authored code has no history, so calibration is the only bridge. **That transfer is unproven.** It is the core product bet, and this is the instrument for testing it — not the answer.

THE BOTTLENECK WAS NEVER VERIFICATION

A verifier with no claim about a feature is silent about it by construction. Every escaped defect in the field reports was a **claim gap** — not a probe that missed.

So the scarce thing is not the engine. It is the *oracle*: a sentence about what must be true, written by someone who is not the code. `gate` names the changed files that carry no claim and stops there, correctly — stating a claim is a human act. But a team adopting this reads that list, has nothing to compare it against, and closes the tab.

FIELD REPORT C — WHERE A SAVE BUTTON ACTUALLY FAILS

Twenty-six faults on the persistence path of seven server actions in a real AI-authored application: the write itself, and the fields it carries.

12 of 26
survived a fully green suite
46% of the persistence path

12 of 12
survivors were a dropped payload field — not one was anything else

32 of 37
files that write are defended only by tests that mock the database

THE WORST CASE — USER REGISTRATION

Each of email, username, password_hash, terms_accepted_at and terms_version can be removed from the row that is written, and no test fails. The tests assert that the write was called. They never assert what it was called with.

FIG. 7 — THE SAVE BUTTON REPORTS SUCCESS. THE SUITE AGREES. THE COLUMN IS EMPTY.

The third number is the read-back answer. A test that replaced the database proves the call shape and nothing beyond it — a `where` matching no rows, a rolled-back transaction, a rejected constraint all pass against a mock. On this surface a clean probe is not evidence of persistence; it is evidence that nothing could have measured it, and the tool says so before any verdict. The evidence now names the mocked layer and the assertion shape — exact, partial or argument-free — beside the survivor's verdict.

A LADDER, SO A REPOSITORY WITH NO CLAIMS IS NOT STUCK

Nothing written

`sweep` takes the files the gate just called uncovered, proposes faults mechanically, probes a bounded selection, and reports what nothing noticed — ordering candidates by what this project's own runs have shown survives, the feedback that took Google's productive rate from 15% to 89%.

A sentence or ten

A *concern* names a kind of promise and where to look. Five lines of JSON — "every admin route refuses a caller without the admin role" — found a route whose owner/manager check can be forced to `false` with no test failing. One sentence, twenty-four routes, no model.

A claim

The survivors worth defending become the first claims: a statement, its faults, and a reproducer already in hand.

Four rules keep the lower rungs safe. A sweep **never writes the claims file** — a sentence nobody wrote is not a claim. Its evidence **never replaces** the canonical record. And it **does not fail on its own bad guess**: only a survivor or an untested file exits non-zero. And whatever it does not probe, it counts — past the cap, outside the concern, or presentational: an icon or a static wrapper nobody would test.

A sweep is weaker evidence than a probe, and says so. It is stronger than nothing, which is what a repository with no claims has.

WHY THE LADDER HAS A BOTTOM RUNG

WHAT IS NOT NEW, AND WHO SOLVED WHICH PART

Breaking code to test your tests is **mutation testing**, and it dates to the 1970s. Stryker, PIT, mutmut and Cosmic Ray all do it. Anyone claiming that part is novel is selling something. Two industrial programmes solved the parts that make it usable, and this tool takes both.

Google — making it affordable

Mutating two billion lines is infeasible, so Petrović et al. mutate only *changed* lines, suppress "arid" nodes nobody would test, cap what is surfaced at seven per file, and order candidates by the measured productivity of the operator in similar context. Developer feedback over six years took their productive rate from **15% to 89%**. A later study found reported mutants are coupled to real faults: the bugs they analysed would have been caught had a mutant been surfaced.

Taken here: the diff scope, the cap, and a productivity-ordered selection — measured on real probed faults rather than assumed. Their arid-node suppression was measured, not copied: Google's categories are under 1% of what the producers propose; the arid class here is presentational JSX, now set aside and counted.

Meta — making it specific

ACH inverts the generator: an engineer writes a *concern* in plain text, an LLM drafts faults for that concern, and each one must build, survive the existing suite, and not be equivalent before a test is generated to kill it. Engineers accepted **73%** of the resulting tests. Nothing is trusted because a model said it — every step is gated by execution, and the mutant is shown to the reviewer as proof the test catches something real.

Taken here: the concern as the unit a human writes, and an execution gate between every model step and the record.

What neither does is bind a fault to a *stated promise*. That is the remaining gap, and it is the next section.

WHAT IS DIFFERENT: THE QUESTION BEING ASKED

CLASSIC MUTATION TESTING	TESTGUARD
Mutates everything, mechanically	**Binds every fault to a stated claim**
"How good are my tests?"	"Is *this specific promise* defended?"
→ ***Mutation score: 73%***	→ ***"Your project says a missing scope fails closed. Nothing checks that."***
A number. Not actionable, not auditable, and it cannot tell you which promise is at risk.	A finding. An assessor reads your claims — reviewable like code — not a score.

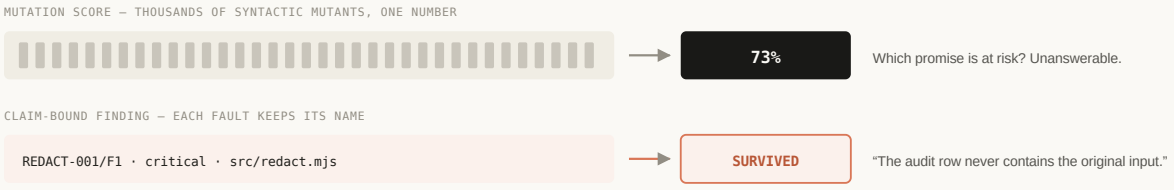


FIG. 6 — A SCORE COMPRESSES EVERYTHING INTO ONE NUMBER; A CLAIM KEEPS THE FINDING ATTRIBUTABLE.

That is the difference between a **metric** and an **audit**, and it is the whole design.

AND ONE THING THAT WAS MEASURED, NOT ASSUMED

“No model decides a verdict” is a design rule elsewhere in this brief. It is also a result. A calibrated commercial decision model was shadow-evaluated against real probed verdicts on an AI-authored codebase: asked whether a given test would catch a given fault, it scored **AUC 0.813** — and its errors ran in the forbidden direction, confidently reporting that a test would catch a fault that in fact survived. On hand-built fixtures the same model looked far better, which is the second finding: a synthetic fixture flatters any judge, because its blind spots were authored to be legible. Execution remains the only thing permitted to say `killed`.

WHAT IT DELIBERATELY IS NOT

- **Not a test generator.** It judges a test the agent wrote; the generating half stays with the agent, where the market is putting it.
- **Not a coverage tool.** A line executed says nothing about whether an assertion would notice.
- **Not a mutation-score dashboard.** No blanket mutants, no single number, no threshold to game.
- **Not a self-healing runner.** A defender that adapts itself to a change is a defender that did not observe it.

And one gap it names rather than hides: a claim that *disappeared* is invisible to all of the above — `probe` verifies the claims that exist, and deleting one is cheaper than weakening it. So removal is reported too, against any reference, with the verdict the claim last had. Removal is allowed; claims can be wrong, superseded or split. It is never silent.

THREE MORE THINGS THAT FOLLOW FROM IT

Paranoid about itself

Most tools mark a mutant killed when the test command exits non-zero — conflating “a test caught it” with “everything exploded.” Here the verdict set is closed and each member means something different.

Evidence, not output

Schema-validated JSON with stable fingerprints, shared with the other Guard tools. Freeze today's blind spots, gate CI on *new* ones.

Built for AI-written code

When an agent writes the tests, the failure mode is not “no tests.” It is confident, plausible tests that assert nothing. That is precisely what this detects.

Coverage tells you a line ran. TestGuard tells you which of your promises nobody is actually defending.

GITHUB.COM/RACCIOLY/TESTGUARD · NPM TESTGUARD-CLI · MIT