The 90-Minute Test Suite Audit

Six things to look for in an AI-written test suite, and what each one costs you.

Your team merges more code than it did last year. The test suite grew with it, most of it written by an agent, and it looks fine. Coverage is up. The dashboard is green.

This is the audit I run to find out whether any of that means anything. Six checks, about 90 minutes, by hand, plus one at the end you can't automate. No tool to install, no account, no upload. **Nothing leaves your machine.** You run it on your own repo, on a branch, and the only thing that ever exists outside your laptop is the number you decide to tell someone.

Every check has the same four parts: the question it answers, how to check it, what a bad result looks like, and what the finding costs you. There are no fixes in here. That is deliberate. Knowing what a broken suite costs is the part you can act on this week; rebuilding it is a bigger conversation.

One honest note before you start: **every "what a bad result looks like" line below is my judgement, not measured data.** Three surviving mutations, two mocked subjects out of ten, one assertion in five. Those are the lines I'd draw, and they're sharp enough to sort a suite into "fine", "normal" and "not load-bearing".

The same goes for the timings on each check. They assume a suite you already know your way around. Depending on how big it is and how familiar the code is, some people will be done in an hour and some will want two.

1. The mutation spot check

(~30 MIN)

The question. If the behaviour your most important code implements silently broke, would the suite tell you?

How to check it. You break the production code, never the tests. The tests stay exactly as they are: they're what's on trial here.

Pick the three functions your business would most hate to see broken. A payment calculation, a permission check, a discount rule, whatever your equivalent is. Do this on a branch, and revert each change before you make the next one. For each function, edit the function itself, making one small and *plausible* change, then run that module's tests unchanged:

1. Invert a boundary: `>` becomes `>=`.
2. Flip a boolean return, or delete one condition from an `if`.
3. Change a returned constant: a rate, a limit, a default.

Nine mutations across three functions, all of them in the source code. No mutation-testing framework needed, and no tooling beyond the test command you already run.

What a bad result looks like. Any mutation that leaves the suite green. Three or more green out of nine means the suite is decorative on the code that matters most. If all nine fail the tests, the suite is doing its job on those paths, and that's a genuinely good result. Write it down and move on.

What it costs. Every green mutation is a class of production bug your CI will wave through. It's also why the dashboard can't be your release evidence. If your suite caught all nine, the mutation result is the evidence and the percentage adds nothing. If it missed three, the percentage doesn't make up for it.

Worth knowing while you do this: both 2026 studies of agent-written tests counted assertions, coverage and flakiness. Neither one checked whether the tests actually catch bugs. The MSR '26 authors end their paper by saying mutation testing should settle that, which is what you're doing for the next half hour.

2. Is the thing under test mocked?

(~15 MIN)

The question. In your recent test files, is the unit being tested itself replaced by a stub?

How to check it. Open the ten most recently added test files. For each one, answer two questions.

1. What is this file supposed to be testing? Usually the file name tells you: `checkout-service.test.ts` is there to test `CheckoutService`.
2. What does it mock? Read the mock setup at the top and list the names.

If the thing from question 1 shows up in the answer to question 2, the test replaced its own subject with a stub and is now asserting against the stub. Same thing if the subject is a thin wrapper and the mocked collaborator is where the real logic lives: the test mocked the part that could actually be wrong.

Count how many of the ten do that.

What a bad result looks like. Two or more of ten. Also worth counting: any test file where the mock setup is longer than the assertions, and any assertion that only proves a mock was called with certain arguments.

What it costs. These tests pass forever. They pass after a refactor that breaks the behaviour, they pass when the dependency changes its contract, and they cost real maintenance time every time the internals move. They're the single most expensive category in an AI-written suite, because they look like coverage, cost like tests and verify nothing.

3. Tests that agree with the code instead of checking it

(~15 MIN)

A tautology is a test that can't fail. Not because the code is right, but because the assertion is built from the same thing it's supposed to be judging. It says "the code does what the code does", and then goes green forever.

The question. Does the assertion say what the behaviour *should* be, or does it just repeat what the implementation *does*?

How to check it. Same ten files. Read only the assertion lines and ignore all the setup above them. For each assertion, ask one question: *if the implementation were wrong, but wrong consistently, would this test still pass?*

Three shapes give it away:

1. The expected value is computed instead of written down. `expect(total).toBe(price * qty)` repeats the formula. `expect(total).toBe(59.90)` states the answer.
2. The assertion only proves something exists: not null, not undefined, no error thrown.
3. The assertion checks an object still has the fields the test itself just put into it.

What a bad result looks like. More than one in five assertions repeating rather than stating. The fastest smell to grep for: an expected value that comes out of a function call rather than being a literal.

What it costs. Tautologies raise coverage and mutation-survivability at the same time, so they're the mechanism behind a green Check 1. They also lock in whatever the implementation did on the day it was generated. The first genuine refactor then produces a wall of failures that tell you nothing about correctness, and your team learns to distrust the suite. That distrust is the expensive part, and it's slow to reverse.

4. Assertion-free and assertion-crowded tests

(~10 MIN)

The question. How many of your tests assert nothing at all, and how many assert so much that a failure is unreadable?

How to check it. You want two counts: test blocks with zero assertions, and test blocks with five or more. The first kind only executes code and fails only if something throws. The second kind fails without telling you which of five things broke.

This awk one-liner does both for a JS or TypeScript suite. It walks each file, counts `expect(` calls between one `it(` or `test(` and the next, and prints the outliers:

```
awk '/^[[:space:]]*(it|test)\(/ {if (seen) printf "%s:%d\t%d assertions\n", f, l, c;
                                 f=FILENAME; l=FNR; c=0; seen=1; next}
     /expect\(/ {c++}
     END {if (seen) printf "%s:%d\t%d assertions\n", f, l, c}' \
  $(find src -name '*.test.ts') | awk -F'\t' '{n=$2+0} n==0 || n>=5'
```

Adjust the two patterns for your stack and the rest still works: pytest is `def test_` and `assert`, JUnit is `@Test` and `assert/verify`, Go is `func Test` and `if .* !=`. If your suite is small enough, counting by eye in the same ten files is fine. The number is what matters, not the tooling.

What a bad result looks like. Any assertion-free test in a critical module. Assertion roulette, meaning many assertions crammed into one test, above roughly one in ten.

What it costs. Assertion-free tests are pure coverage inflation. They move the number and verify nothing. Assertion-crowded tests do verify things, but when one fails, whoever is on call has to read the test to work out what broke. Measured on agent-written suites this is the more common of the two: agents write a median of two assertions per test against a human median of one (MSR ’26). More assertions isn’t automatically a better test. Sometimes it’s just a longer failure message.

5. Drift and flakiness

(~10 MIN)

The question. Two questions, really. Does the same suite, run against the same code, give you the same answer every time? And do these tests check things the way your team checks things?

How to check it. Run the full suite three times in a row, changing nothing between runs. Every test should land on the same result all three times. A test that's green, then red, then green is flaky: nothing in the code moved, so the test is reacting to something it shouldn't be, usually timing, ordering, shared state, a real clock or a real network call. Write down which ones did it.

Then scan the same ten files for assertions that don't look like your codebase. What you're looking for is an agent that reached past your test library and invented its own way to check the result:

1. Hand-rolled comparisons. Instead of `expect(user).toEqual(expected)`, the test loops over the keys itself, or compares `JSON.stringify(a) === JSON.stringify(b)`, or builds a boolean and asserts that the boolean is true. The library already does this, better, with a readable failure message.
2. String matching on serialised output: asserting a formatted or stringified result contains a substring, rather than checking the value itself.
3. Custom truthiness helpers: a locally defined `isValid()` or `checkResult()` that wraps the real check, so the assertion line tells you nothing.

What a bad result looks like. Any test that changes result across three identical runs. Any file using a validation pattern nobody on the team would have written.

What it costs. Agents tend to reach for non-standard assertion patterns far more often than people do, and their tests tend to be flakier. Your linters won't flag those patterns. They aren't errors, just unfamiliar. And the reviewer skimming the diff sees a green test and moves on. Flaky tests are worse than useless. They teach a team to hit re-run instead of reading the failure. After that, the gate is just noise.

6. Coverage against the paths that matter

(~20 MIN)

The question. Are your five most important user journeys actually verified, or just executed?

How to check it. Write down the five things your product must never get wrong. End to end, in the user's language, not in modules. For each one, find the test that would fail if it broke. Not the tests that touch the code, but rather the test that pins the outcome.

You'll usually find a test that looks like it covers the journey. Most of them don't. Three things rule a test out:

1. It mocks the boundary the journey crosses. If the payment goes out through a gateway and the gateway is a stub, the test proves your code called a stub.
2. It asserts that a call happened, not what came out of it. "The charge function was invoked once" is not the same as "the customer was charged the right amount".
3. It covers one step, not the result. A test on the checkout form doesn't tell you the order arrived.

What a bad result looks like. Two or more of the five where no such test exists, or where the only test you found is one of the three above. Then compare that against your coverage percentage on the same modules. Both numbers can be true of the same suite. It reports "94% covered", and three of your five journeys can still break without a single test going red. Coverage counts lines that ran, not journeys that are checked. That's why one can look fine while the other doesn't.

What it costs. "Three of our five critical journeys have no test that would fail if they broke" is a sentence anyone in the company understands, and nobody can shrug off. It's the only result in this checklist stated in the business's own terms rather than in engineering's. A coverage percentage goes on a dashboard and nothing happens. This one doesn't, and that's exactly why it's the one to take to leadership.

Scoring

Six checks, pass or fail, against the bad-result lines above.

RESULT	WHAT IT MEANS
0–1 failed	The suite is doing its job. Re-run this after the next quarter of AI-assisted work. Your failure mode is drift, not the current state.
2–3 failed	Normal for a team that adopted AI tooling 12+ months ago. The coverage number is no longer load-bearing. Treat it as an activity metric.
4–6 failed	The instruments aren't measuring what anyone thinks they measure. Green builds shouldn't be used as release evidence until at least Check 1 and Check 6 pass.

If you landed in the first band, you're done. That's a real result, not a consolation prize, and the honest thing to do with it is to note the date and check again in a quarter.

One thing this checklist can’t measure

This one isn’t scored, and no scanner will ever run it. It’s here because it’s the finding that explains most of the other six.

Since you adopted AI coding tools, has review depth kept up with review volume? Pull the last 50 merged PRs and the 50 immediately before adoption. Compare median diff size, median time from open to approval and comments per PR. If diff size rose and time-to-approval fell, review has quietly become an approval step.

Nobody reviews three times the volume at the same depth for long. That isn’t a discipline problem and it isn’t anyone’s fault. It’s arithmetic. And no tool can spot it, because an approval nobody really read leaves exactly the same trace as a careful one: a green check and an approving comment.

Ran this? Write me and tell me which check you failed. I read every one, and I’ll tell you what I’d look at next.

info@thomaslanghorst.com

Writing more of these. Get them by email.

Sources. Yoshimoto et al., *Testing with AI Agents: An Empirical Study of Test Generation Frequency, Quality, and Coverage*, MSR ’26 (arxiv.org/abs/2603.13724). Jhanglani et al., *Beyond Test Presence: Assessing the Quality and Robustness of Agent-Generated Tests in Open-Source Projects*, 2026 (arxiv.org/abs/2607.12068).