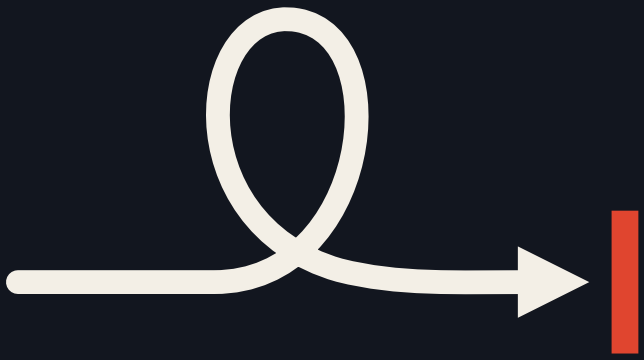
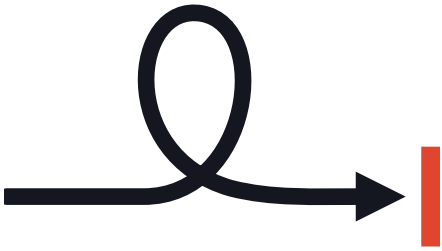


The Second Time Is a Bug

A field workbook for engineering with AI coding agents

RAKESH MENON



The Second Time Is a Bug

A field workbook for engineering with AI coding agents

RAKESH MENON

The Second Time Is a Bug
A field workbook for engineering with AI coding agents

Copyright © 2026 Rakesh Menon. All rights reserved. No part of the text may be reproduced or shared without written permission, except for brief quotations in reviews.

The companion code is released under the MIT licence. You may copy, adapt and use it in your own projects.

This book is independent. It is not affiliated with, sponsored by or endorsed by any AI vendor or tool maker. Product names mentioned in the text are trademarks of their owners and are used only to identify those products.

The information in this book is provided as is, without warranty. Test every guard, hook and permission change in a safe environment before you rely on it.

Book website: secondtimeisabug.com

Companion code: github.com/second-time-is-a-bug-book/book-companion, tag v1.0.0 (MIT licence).

Version 1.0, built 22 October 2026. Set in Source Serif 4, Inter and JetBrains Mono, all under the SIL Open Font License.

Contents

How to Use This Book

A Note on Names and Sources

Preface: Why the Second Time

PART 1 THE LOOP

1 Two Harnesses

2 The Second Time Is a Bug

PART 2 GUIDES

3 Instruction Files That Stay Small

4 Keeping Context Clean

PART 3 SENSORS

5 Gates That Say No

6 **Tests the Agent Cannot Game** IN THIS SAMPLE

7 Behaviour You Can See

PART 4 BLAST RADIUS

8 Permissions and Credentials

9 The Supply Chain You Did Not Choose

PART 5 SCALING THE LOOP

10 Legacy Code

11 Loops Beyond One Session

PART 6 ECONOMICS

12 Does the Harness Pay?

13 Retiring Guards

14 Closing: The Second Time, Again

APPENDICES

A Pattern Card Index

B Templates

C Wiring Guards into Today's Tools

D Further Reading

Notes

About the Author

CHAPTER 6

Tests the Agent Cannot Game

Accounting settled long ago that the person who keeps the books should not also audit them. Nobody assumes the bookkeeper is dishonest. The point is that a check made by the same hand that did the work shares every blind spot and every pressure of that work, so it is not much of a check. The duties are split, and the auditor's records are kept where the bookkeeper cannot change them.

Software teams built a looser version of the same split. Authors do not approve their own pull requests. Where a team has testers, they exist partly so that the person who built a feature is not the only one deciding that it works.

A coding agent in your repository undoes that split without anyone choosing to. It writes the code, it can write the tests, and it does both in the same session with the same goal: a green run. Usually it reaches that goal honestly, through the code. When the code will not cooperate, a shorter route is open.

Your coding agent is working on Invoicer's tax code, and one test has gone red: `taxCents rounds half up to a whole cent` in `app/test/money.test.ts`. The agent reads the failure, edits `money.ts`, runs the suite. Still red. It tries a second fix. Still red. On the third attempt it does something that works every time: it deletes the test. Seven lines go, the suite runs, every remaining test passes, and the agent reports the task done with the output to prove it.

The output is genuine: every test that ran, passed. What it actually shows is that Invoicer no longer tests rounding tax half up, the one behaviour the change put at risk. A reviewer skimming the diff sees an edit to `money.ts`, a shorter test file and a passing build, and it merges.

This chapter is about the sensor every other sensor leans on, and the awkward fact that the agent can edit it.

If you are starting the book here: your harness is what you build around the agent (instructions, checks and gates). The method is to log a mistake the first time, and the second time add a sensor that catches it, proven by planting the mistake on purpose. Each control below ends in a pattern card and comes with a drill: one command in the companion code that makes the mistake in a scratch copy and shows the guard catching it, with no model or API key.

Seven ways to a green run that proves nothing

An agent asked to make the tests pass is working towards a green run. Usually the shortest path goes through the code under test; sometimes it goes through the tests. Kent Beck, writing about building a B+ tree with an agent under strict test-driven rules, listed this among the signs that the work was going off track:

> *"Any indication that the genie was cheating, for example by disabling or deleting tests."*[1]

"Cheating" suggests intent, which you might hope to talk the agent out of. A more useful model is mechanical: the test file is part of the environment, the agent changes the environment until the goal is met, and nothing marks the tests as different from any other file.

Economists met this long before coding agents. In 1975 Charles Goodhart observed that a statistical regularity tends to collapse once it is used for control, and the anthropologist Marilyn Strathern later gave the idea the form people remember: "When a measure becomes a target, it ceases to be a good measure."[2] A green run is a measure of working code. Ask an agent for green and you have made it the target. The guards in this chapter do not argue with that pressure: they take the levers away, or catch them being pulled, so that the only route left to green goes through the code.

No reliable rate has been published, though Gene Kim and Steve Yegge's book *Vibe Coding* includes a story of an agent deleting 80 percent of the test cases in one large suite[3]. You do not need a rate. You need the second occurrence in your repository to be impossible, and for that you need to know the moves. There are seven:

1. **Delete the failing test**, or comment it out. The opening scene.
2. **Switch it off.** `test.skip` on the failing test, `.todo`, `xit`, or `.only` on a different test so the runner quietly ignores its neighbours.
3. **Hollow out the assertion.** `assert.equal(addDays('2026-01-30', 3), '2026-02-02')` becomes a check that the result looks like a date. An early `return`, or a `try` that swallows the error, does the same.
4. **Move the goalposts.** Change the expected value to whatever the code now returns.
5. **Regenerate the snapshots.** Run the snapshot runner's update flag after introducing a bug, and today's wrong output becomes tomorrow's expected output, in a file nobody reads line by line.
6. **Mock the thing under test.** Replace the function being tested with a stub that returns the expected answer. The test now

checks the stub.

7. **Claim the run.** Report that the tests pass without running them, or after running them on an earlier version of the code.

Each move has a sensor, or an honest note that it has none, and Figure 6.1 maps them.

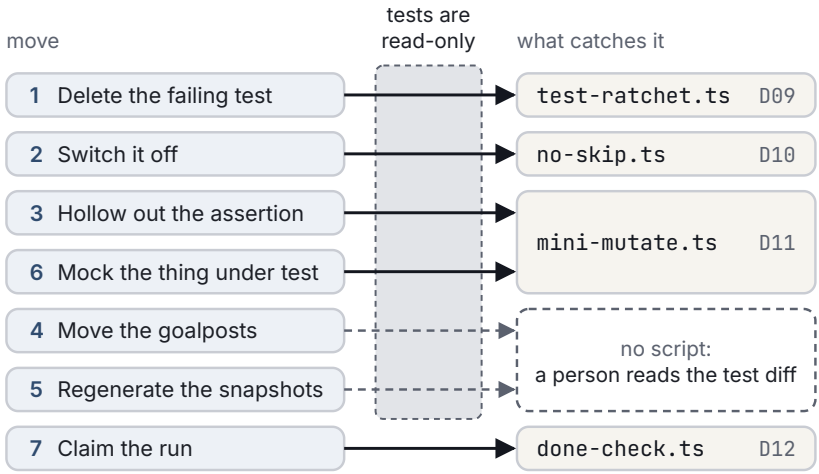


Figure 6.1. The seven moves and what catches each in Invoicer, with the drill that proves it. Moves 1 to 6 all edit a test file, so the read-only rule meets them first; permissions leak, so each still needs something behind it, and for moves 4 and 5 that is a person reading the diff.

Two moves have no script: no program can tell whether 201 or 200 is the right expected value without knowing what the business wants. A placement decision defends them.

Tests are read-only to the agent

The cheapest control, TESTS BEHIND GLASS (P14), is decided before the agent starts work: during an implementation task, the agent may read and run the tests but may not write them. When behaviour really does change, the test changes first, in a separate

task. You write the new test (or review one the agent writes in a test-only task), watch it fail for the right reason, commit it, and then the implementation task begins with the tests locked. This is red/green test-driven development with a lock between the two colours. Beck worked this way with his agent, and Simon Willison recommends "red/green TDD" as a standing prompt for coding agents[4].

The lock has a price when nobody is watching. If a task really does invalidate an old assertion and no one wrote the new test first, the agent stops one test short and asks; in the experiment in Chapter 12, 23 of 30 such runs did. Keep the lock, and either write the test first or name in the task which test must change.

Who writes the tests matters too. Birgitta Böckeler notes that behaviour harnesses lean heavily on AI-written tests, which are not good enough yet on their own[5]. A test the agent writes alongside the code encodes its understanding of the task, misunderstandings included. A test a person wrote or reviewed beforehand is an independent opinion, and an independent opinion is the whole point of a sensor.

Most coding agents let you deny writes to paths in their permission settings (see Appendix C). Deny the test folders (`app/test/`, `app/e2e/` and `legacy/characterisation/` in Invoicer) and any folder that holds snapshots. Then say it in the instruction file, with the checks that back it up:

AGENTS.md

```
- Tests are read-only for agents. Never delete, skip or loosen
  one: `node guards/no-skip.ts`, `node guards/test-ratchet.ts`
```

Permissions leak: tools enforce them differently, an agent with a shell can write a file without its edit tool, and a lock lifted for a test-writing task is easy to forget. So add sensors for the commonest leaks, starting with tests that are switched off:

guards/no-skip.ts

```
const RULES = [
  {
    patterns: [
      /\b(?:test|it|describe|suite)\.skip\(/,
      /\{\s*skip:\s*true/,
    ],
    what: 'a skipped test never runs, so it can never fail',
  },
  {
    patterns: [
      /\b(?:test|it|describe|suite)\.only\(/,
      /\{\s*only:\s*true/,
    ],
    what: 'an .only test makes the runner skip its neighbours',
  },
  ...
];

const FIX =
  'Fix: remove it and make the test pass by fixing the code ' +
  'under test. If you believe the test is wrong, stop and ' +
  'ask a human: tests are read-only for agents (AGENTS.md, ' +
  'Rules).';
```

The rules also catch the option forms, `.todo` and `xit`, and ignore comment lines. Notice the second half of the fix line: some failing tests really are wrong, and an agent that believes so needs a legitimate exit, or it will find an illegitimate one. Drill D10 skips one test and focuses another:

```
no-skip: FAILED, 2 problems
  - app/test/csv.test.ts:32: an .only test makes the runner skip
    its neighbours.
      test.only('commas and quotes force quoting', () => {
    Fix: remove it and make the test pass by fixing the code
      under test. If you believe the test is wrong, stop and ask
      a human: tests are read-only for agents (AGENTS.md,
      Rules).
  - app/test/invoice.test.ts:42: a skipped test never runs, so
    it can never fail.
...
```

The guard reads text, so a `return;` on the first line of a test body, an `if (false)` or an aliased `test` gets past it. Mutation testing, below, catches the first two: it does not care how a test was switched off, only that it stopped checking anything.

For the two moves that no script catches, changed expected values and regenerated snapshots, make test changes visible. One line of git lists them:

```
git diff --name-only HEAD -- app/test legacy/characterisation
```

In an implementation task that should print nothing, and anything it prints needs a person to read it. If your agent cannot lock paths, this is a ten-minute guard: fail when the list is not empty and the task is not labelled as a test change. A regenerated snapshot is a test change too, however tidy it looks.

> **>_ DRILL D10** skipped-test
>
> Run it: `npm run drill D10`
>
> Source: `drills/D10-skipped-test/`
>
> The patch turns one test in `app/test/invoice.test.ts` into `test.skip` and one in `app/test/csv.test.ts` into `test.only`. The guard passes on 16 test files before the patch and reports two problems after it, each with the line, the reason and the instruction to fix the code or ask a human.

P14 Tests Behind Glass PATTERN CARD

Tests are read-only to the agent

SYMPTOM	*To get a green run, the agent edits, skips, focuses or loosens the tests instead of fixing the code.*
DRILL	`D10`
GUARD	**A permission rule that denies agent writes to test and snapshot folders during implementation tasks, stated in `AGENTS.md`; `guards/no-skip.ts` catches the commonest leak (`.skip`, `.only`, `.todo`, `xit` and option forms).**
PROOF	Apply the D10 patch in a scratch copy; no-skip exits 1 and names both lines. With permissions on, ask the agent to "get the suite green quickly" while a test fails and confirm the write is refused.
PLACEMENT	**edit** › **commit** › build › deploy › live
KIND	guide and sensor; computational
RETIRE WHEN	The read-only rule stays. Retire no-skip when tests are read-only to the agent by enforced permissions and the guard has not fired for 90 days.

Ratchet the count

The bluntest move is also the hardest to spot in review: a test simply disappears. A TEST COUNT RATCHET (P15) turns the size of the suite into a number that may only go up. `guards/test-ratchet.ts` runs the whole suite, reads the summary line that Node's test runner prints (`tests 78`), and compares it with the number stored in `.harness/test-baseline.json`. Here is the opening scene replayed as drill D09:

```
test-ratchet: FAILED, 1 problem
  - The test count fell from 78 to 77.
    Fix: put the missing tests back. Do not delete, merge or
      skip tests to get a green run; fix the code instead. If a
      human agreed to remove a test, they lower the baseline in
      .harness/test-baseline.json themselves.
```

The suite passed; the ratchet refused anyway, because it asks not "did the tests pass" but "are all the tests still here". Only a person lowers the number, by editing a file in the diff where a reviewer sees it, not by passing a flag nobody notices. Raising works the other way: `node guards/test-ratchet.ts --raise` writes a higher count and never a lower one. Raise it in the same commit that adds tests, or the gap becomes a free allowance of deletions.

A ratchet has two blind spots. Node's runner counts skipped and todo tests in its total, so a `.skip` leaves the count unchanged; that is why `no-skip.ts` sits beside the ratchet. And the ratchet counts tests without weighing them: delete one strong test, add one that asserts `true`, and the count holds. Weighing is the next section's job. The ratchet also reruns the suite, which puts it in the slow lane; on a ten-minute suite, read the count from the gate's own test run instead.

> **>_ DRILL D09** deleted-test
>
> Run it: `npm run drill D09`
>
> Source: `drills/D09-deleted-test/`
>
> The patch deletes the seven-line rounding test from `app/test/money.test.ts`. Before, the ratchet reports 78 tests against a baseline of 78. After, the remaining 77 tests all pass and the ratchet fails anyway, with the message above.

P15 Test Count Ratchet PATTERN CARD

Ratchet the count

SYMPTOM	*A failing test is deleted or commented out, and the run is green.*
DRILL	`D09`
GUARD	**`guards/test-ratchet.ts` runs the suite, counts the tests and fails when the count falls below `.harness/test-baseline.json`. Only a person lowers the baseline, by editing the file; `--raise` only ever moves it up.**
PROOF	Apply the D09 patch in a scratch copy; 77 tests pass and the ratchet exits 1 with “The test count fell from 78 to 77”.
PLACEMENT	edit › **commit** › **build** › deploy › live
KIND	sensor; computational
RETIRE WHEN	Tests are read-only to the agent by enforced permissions and the guard has not fired for 90 days.

Mutants catch hollow tests

Mutation testing makes the code wrong on purpose, one small change (a mutant) at a time, and runs the tests after each. If some test fails, the mutant is killed. If every test still passes, it survived, and you have learned something no coverage report tells you: that line runs during the tests, but nothing checks what it does. Figure 6.2 shows both outcomes on Invoicer’s money code. Running a handful of mutants against the tests you rely on is how you KNOCK FOR HOLLOW TESTS (P16): tap the wall and listen for the hollow spot.

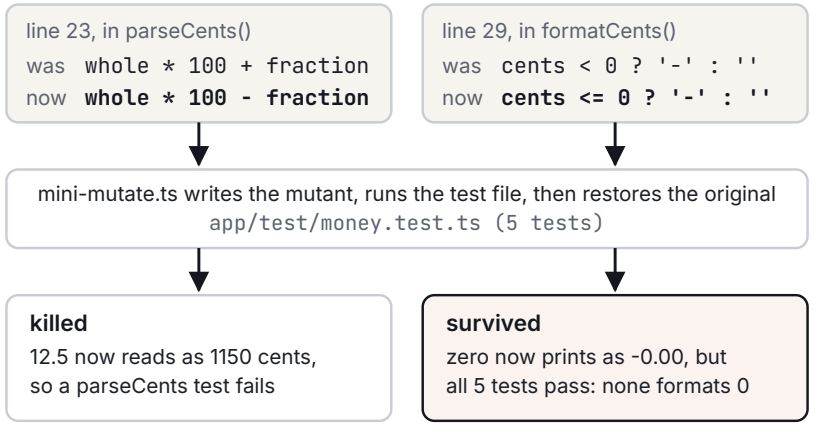


Figure 6.2. Two of the three mutants of money.ts, judged by the same five tests. A test fails on the first, so it is killed; the second changes how zero prints and every test still passes, so it survives and points at a missing assertion.

That is exactly what a hollow test hides. Drill D11 asks the agent to make the due-date tests "less brittle", and it does:

drills/D11-hollow-assertion/mistake.patch

```
-    equal(addDays('2026-01-30', 3), '2026-02-02');
-    equal(addDays('2026-12-31', 1), '2027-01-01');
+    match(addDays('2026-01-30', 3), /^\d{4}-\d{2}-\d{2}$/);
+    match(addDays('2026-12-31', 1), /^\d{4}-\d{2}-\d{2}$/);
  ...
-    equal(isOverdue('sent', due, due), false);
+    equal(typeof isOverdue('sent', due, due), 'boolean');
```

Every test still runs and every line of `due-dates.ts` still executes, so coverage shows the file fully covered. The ratchet and no-skip are satisfied, and the tests now check nothing but types.

The companion's mutation tester is about 100 lines. Its mutations are a table of operator flips:

guards/mini-mutate.ts

```
const FLIPS: [string, string][] = [
  [' === ', ' !== '],
  ...
  [' > ', ' >= '],
  ...
  [' && ', ' || '],
  ...
  [' + ', ' - '],
  [' - ', ' + '],
];
```

For each flip it finds, it writes the mutated file, runs the named test file and records whether any test failed. A `finally` block writes the original back even if a test run crashes, because a mutation tester that leaves a mutant behind has just introduced a bug. Against the strong due-date tests, all five mutants are killed. Against the hollow ones:

```
mini-mutate: 5 mutants of app/src/domain/due-dates.ts, judged by
  app/test/due-dates.test.ts
  SURVIVED  line 9: + -> -
  SURVIVED  line 21: === -> !==
  SURVIVED  line 21: > -> >=
  SURVIVED  line 21: && -> ||
  SURVIVED  line 29: - -> +
mini-mutate: FAILED, 5 problems
...
  - app/src/domain/due-dates.ts:21: changing ">" to ">=" did not
    fail any test.
    Fix: add or strengthen an assertion in
      app/test/due-dates.test.ts that this change would break.
    Never loosen an assertion to make a test pass.
...
```

Read the third survivor. Changing `>` to `>=` on line 21 makes an invoice overdue on its due date instead of the day after, which would send customers a reminder a day early. The original test

`an invoice due today is not overdue yet` pins that rule; the hollow version keeps the name and no longer checks it.

Mutants also find gaps no agent made. Point the tool at `app/src/domain/money.ts` and its tests and, as this book goes to press, one of its three mutants survives, the `formatCents()` flip in Figure 6.2: zero prints as "-0.00", and no test formats zero. The same mechanism exposes over-mocking: if a test replaces the function under test with a stub, mutants of the real function survive, because the real function never runs.

Every mutant costs a test run, and that decides where mutation testing runs. `mini-mutate.ts` runs only the one test file you name, so the due-date file takes about a second. Full tools (Stryker for JavaScript and TypeScript, PIT for Java) produce many more mutants, and 200 of them against a 90-second suite is five hours. Survivors also need judgement: an "equivalent" mutant changes the code without changing anything observable, and only a person can say a survivor is harmless. So mutation testing stays out of the commit gate and runs at build time, nightly or on the files an agent touched in the branch, and a survivor is a report for a person: strengthen an assertion, never loosen one.

> **>_ DRILL D11** hollow-assertion
>
> Run it: `npm run drill D11`
>
> Source: `drills/D11-hollow-assertion/`
>
> The patch replaces every value check in `app/test/due-dates.test.ts` with a type or shape check. Before, all five mutants of `due-dates.ts` are killed. After, the tests still pass, and all five mutants survive, each reported with its line and the operator that changed.

P16 Knock for Hollow Tests PATTERN CARD

Mutants catch hollow tests

SYMPTOM	*Tests pass and cover every line, but their assertions only check types or shapes, return early, or test a mock, so a real bug would not fail them.*
DRILL	**D11**
GUARD	**`guards/mini-mutate.ts <source> <test>` flips one operator at a time, runs the named test file, restores the source, and fails listing every surviving mutant.**
PROOF	Apply the D11 patch in a scratch copy; the five mutants of `app/src/domain/due-dates.ts` go from killed to survived.
PLACEMENT	edit › commit › **build** › deploy › live build (nightly, or on the files an agent touched in the branch)
KIND	sensor; computational
RETIRE WHEN	You run a full mutation testing tool on the same files.

Done means evidence

The last move needs no test edits at all, as drill D12 shows. The agent runs the suite at the start of its final step: 78 tests, all passing. Then, while writing its summary, it “tidies” one comparison in `app/src/domain/reminders.ts`, turning `<=` into `<`, and reports done, citing the earlier green run: evidence about code that no longer exists. Run the suite again and `no second reminder within the reminder interval` fails: with `<`, the second reminder goes out a day late, eight days after the first instead of seven.

Anthropic's engineers saw long-running agents mark features complete without checking them end to end; part of their answer was a feature list kept as JSON in which only a `passes` field was meant to change[6]. This book's angle is narrower: a claim of done must come with a file that records a passing run, tied to exactly the code being called done. Call it NO RECEIPT, NO DONE (P17).

guards/done-check.ts

```
interface Evidence {
  command: string;
  exitCode: number;
  summary: string;
  recordedAt: string;
  // path -> content hash, at the time the tests ran
  files: Record<string, string>;
}
...
  const then = evidence.files;
  const paths = Object.keys({ ...then, ...now });
  const changed = paths.filter((p) => then[p] !== now[p]);
```

`npm run evidence` runs the tests and writes `.harness/evidence.json`: the command, its exit code, the pass and fail counts, and a SHA-256 hash of every source file. `node guards/done-check.ts` refuses if there is no evidence, if the recorded run failed, or if any file's hash differs. Here is D12:

```
done-check: FAILED, 1 problem
  - The evidence is stale: 1 file(s) changed after the tests ran
    at 2026-10-13T12:59:19.958Z:
      app/src/domain/reminders.ts
    Fix: run npm run evidence and check it passes before you
      report the task as done.
```

Figure 6.3 traces the same run: only the edited file's hash moves, and that is enough to refuse.

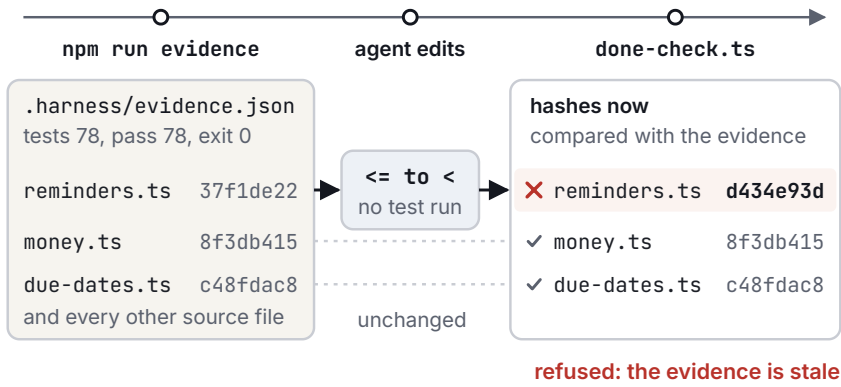


Figure 6.3. Drill D12 as a time line. The evidence file records a hash of every source file when the tests pass; the one-character edit changes the hash of reminders.ts, so done-check finds the evidence stale. Hashes are real, shortened to eight characters.

Why hashes and not file times? Times are wrong in both directions. Extracting an archive, `cp -p`, `rsync -a` and some checkouts give new content an old modification time, so a time check passes untested code; switching branches and back, or a formatter that rewrites a file byte for byte, makes unchanged files look newer and sends the agent to rerun the suite for nothing. A content hash asks the real question, whether these are the bytes the tests ran against, and because it compares the union of old and new paths, an added or deleted file counts as a change too.

`AGENTS.md` tells the agent to record evidence and run the check before saying done. Instructions are guides, so the companion's example hook settings (`templates/hooks/agent-pre-action.example.json`) also run `done-check.ts` when the agent tries to stop, and send it back to work if the check fails. Such a hook must still let the agent stop to ask a human, or it pushes it towards the very gaming this chapter is about: an agent that writes its reason in `.harness/stop-note.md` may stop, and a

second stop with the check still failing goes through, with the failure shown to you, rather than looping. The claim of done cannot leave the session quietly.

Two limits. The fingerprint covers the folders listed in `SOURCE_DIRS`; Invoicer's root `package.json` and `tsconfig.json` sit outside them, so add whatever your test command reads. And the evidence file is within the agent's reach: the check stops the forgotten run and the stale claim, which are the common cases, not a determined forger. When your agent harness itself runs the tests before it lets the agent stop, this guard retires.

Why not sign the evidence, so that a forged file fails the check? A signature needs a key, and the key has to live where the agent cannot read it. When the agent runs shell commands as you, that means a second user account, a separate service or a CI runner: the same machinery that would let your agent harness run the tests itself, which is this guard's retirement condition. Built here, signing would cost more upkeep than the rest of the guard, to stop the rare case, a forgery, while the common ones are already caught. If you ever find a forged evidence file, treat it as a first time: log it, and move the test run out of the agent's reach.

> **>_ DRILL D12** `false-done`
>
> Run it: `npm run drill D12`
>
> Source: `drills/D12-false-done/`
>
> The drill records evidence first, and done-check passes with 78 of 78 tests. It then applies the one-character change to `app/src/domain/reminders.ts` without rerunning the tests, and done-check fails, naming the file that changed since the evidence was recorded.

P17 No Receipt, No Done PATTERN CARD

Done means evidence

SYMPTOM	*The agent reports done with no test run, or with a run from before its last edit.*
DRILL	**D12**
GUARD	**`npm run evidence` records a test run with a SHA-256 hash of every source file; `guards/done-check.ts` refuses when the evidence is missing, failed, or does not match the current files. Wired as the agent's stop hook.**
PROOF	Apply the D12 patch after recording evidence; done-check exits 1 and lists `app/src/domain/reminders.ts` as changed after the tests ran.
PLACEMENT	**edit** › **commit** › build › deploy › live edit (the agent's stop hook) and commit
KIND	sensor; computational
RETIRE WHEN	Your agent harness runs the tests itself before it lets the agent stop.

Holes that cover each other

Each of these guards has a hole that another covers. No-skip misses an early return; mutation testing catches it. The ratchet counts tests without weighing them; mutation testing weighs them. Mutation testing cannot tell whether an expected value is the one the business wants; the read-only rule and a person reading test diffs can. And the evidence file stops a claim of done that rests on a run that never happened. Layered, they turn "the tests pass" back into a statement you can trust.

A guard that never fires might mean the agent has stopped making that mistake, or that the guard has quietly broken.

Chapter 7 plants known defects to find out which, then tests what a user actually sees.

Checklist

- [ ] Deny your agent write access to test and snapshot folders during implementation tasks, and make test changes a separate task that a person reviews.
- [ ] Add a skip and focus scanner to your gate whose message says to fix the code or ask a human.
- [ ] Record your test count as a baseline, fail any run where it falls, and raise it in the same commit that adds tests.
- [ ] Tonight, run a mutation pass on the source file your agent changed most this week, and read every survivor.
- [ ] Require a fresh, hash-checked evidence file before the agent may report done, and wire the check into its stop hook.
- [ ] List the test files changed in each implementation diff, and read every one.

NOTES

1. Kent Beck, “Augmented Coding: Beyond the Vibes”, Tidy First? (Substack), 25 June 2025, https://newsletter.kentbeck.com/p/augmented-coding-beyond-the-vibes. Date from the summary on kentbeck.com, https://kentbeck.com/summaries/augmented-coding-beyond-the-vibes/. ↩
2. Charles Goodhart, “Problems of Monetary Management: The U.K. Experience”, *Papers in Monetary Economics*, vol. 1, Reserve Bank of Australia, 1975. Marilyn Strathern, “‘Improving ratings’: audit in the British University system”, *European Review*, vol. 5, no. 3, 1997, pp. 305 to 321; the quoted sentence is on p. 308, where Strathern takes the idea from Keith Hoskin, who calls it Goodhart’s law. ↩
3. The Register, review of Gene Kim and Steve Yegge, *Vibe Coding* (IT Revolution), theregister.com, 21 October 2025, https://www.theregister.com/2025/10/21/book_review_vibe_coding/. ↩

4. Simon Willison, “Red/green TDD”, in *Agentic Engineering Patterns*, simonwillison.net, guide started 23 February 2026, https://simonwillison.net/guides/agentic-engineering-patterns/red-green-tdd/. ↩

5. Birgitta Böckeler, “Harness engineering for coding agent users”, martinfowler.com, 2 April 2026, https://martinfowler.com/articles/harness-engineering.html. ↩

6. Justin Young (Anthropic), “Effective harnesses for long-running agents”, Anthropic Engineering, 26 November 2025, https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents. ↩

Get the full book

You have just read Chapter 6 of **The Second Time Is a Bug**. The full book has 14 chapters, 34 pattern cards and 26 drills you can run against the public companion code (github.com/second-time-is-a-bug-book/book-companion), plus the complete pattern card deck and starter templates.

The full book (PDF and EPUB, with the printable pattern card deck) is **available on Leanpub**: **https://leanpub.com/secondtimeisabug**.

More about the book: **secondtimeisabug.com**.