The Hook Audit

>_ PrimeLine
Build Systems Not Sessions

17 checks that tell you whether your Claude Code hooks actually do anything

PrimeLine · companion to the Claude Code hook measurements on primeline.cc/blog · every number from one live system

Group A: does it act at all

1. Has this hook ever acted?

Read the log and count actions taken, not rows written. Blocks, denials, injections, edits. Not invocations.

```
grep -c '"blocked": true' your-hook.jsonl
```

Mine: 3,885 rows over three months, 0 blocks.

How to read it: zero actions over a long window is a defect report, not a quiet success. Either the hook is misconfigured or it is not needed. Both are worth knowing.

2. Is enforcement behind a flag nobody set?

```
grep -o '"[A-Z_]*ENFORCE[A-Z_]*"' ~/.claude/settings.json | sort -u
```

Then check your shell profile too, because an unset environment variable is invisible from the settings file alone.

Mine: zero occurrences. My strictest gate needed one env var to block, and it had never been set.

How to read it: if the flag is absent, the hook is an observer. That may be correct. It is not enforcement, and anything you wrote assuming enforcement needs revisiting.

3. Does the evidence it checks for ever arrive?

If your hook looks for a marker in the output, grep the log for that marker rather than for the hook name.

Mine: the marker appeared in 1 row of 3,885. The gate had never once evaluated a real proof.

How to read it: a gate with no input has never been tested in production, whatever its test suite says.

4. Does your log have a denominator?

Count rows written against evaluations performed. If a non-fire writes nothing, the two differ and you cannot compute a rate.

```
wc -l < your-hook.jsonl
```

Compare that against your prompt or tool-call count, never against itself.

Mine: no denominator. "No row" was indistinguishable between ran-and-matched-nothing, suppressed, and never-reached-because-an-upstream-guard-exited.

How to read it: write a row on every evaluation with an explicit state, including the boring one.

5. Are the producer and consumer on the same channel?

For every pair, write down the medium: in-process object, message text, file on disk, environment variable, config key.

Mine: one component built evidence as an in-process object and passed it to a library, while the hook parsed the text of the message. Both halves worked. The pair was dead for months.

How to read it: "A produces X for B" without a named channel is not a dependency map, it is a hope.

6. What would the loose rule flag, and what would the tight one?

Run both variants of your matcher over the same historical window and compare.

Mine: the naive rule would have flagged 99.1% of turns. The narrowed rule, 5.2%.

How to read it: if loose is noise and tight is silence, there is no setting in between. Stop tuning and change what you read.

Group B: does it still land

7. Does it still land, or only still fire?

Join each fire to what happened next in the same session. Counting fires tells you the process ran. Only the join tells you anything landed.

Mine: 303 observable injections across 68 sessions, 49% followed overall. The aggregate hides everything that matters, which is check 8.

How to read it: a hook that fires perfectly and gets ignored looks identical to one that works. If you have no join, you have no answer.

8. Does the previous outcome predict the next one?

For each fire, look at whether the one before it in the same session landed.

Mine: after a followed reminder, the next one was followed 76% of the time. After an ignored one, 18%. That is a 4x gap on 303 injections, permutation p below 0.0001.

How to read it: this is the finding that survived when a simpler one did not. I first measured a decline by position in session, 92% on the first fire down to 31% by the sixth. Re-derived on four months of data it did not hold, and it reversed sign depending on how I treated fires with no observable follow-up. Path dependence held across all six specifications I tried, at ratios between 2.2x and 4.3x. Test both, and trust the one that survives the specification change.

9. How many different things has it ever said?

Deduplicate the findings across every run, not just count the runs.

Mine: a saturation monitor ran 2,720 times, reported a finding on 100% of them, 5,465 instances in total. Deduplicated: **two** distinct findings, one seen 5,440 times and one 25 times.

How to read it: rate is the wrong question. A monitor with a 1,360-to-1 repeat ratio is a stuck bit with a timestamp, whatever its firing frequency. This check also caught a stale number of my own: I had written that a second hook fired on 0.9% of edits and produced 11 rows in three weeks. Recounted over two months against 3,308 Edit and Write calls, it has written 115 warnings across 44 sessions, which is 3.48%. Recount before you cite, including your own notes.

10. What share of the input is human?

```
cat ~/.claude/projects/*/*.jsonl | jq -r 'select(.type=="user") | .message.content | strings' | wc -l
```

Then subtract agent briefs, command bodies and task notifications.

Mine: 1,803 of 52,020 user records were a person typing. 3.5%.

How to read it: if most of the input is machine text, a keyword matcher is tuned against robots, not against you.

11. What language is your scorer written in?

Read the keyword list. Then count how many genuine prompts score zero.

Mine: 6,131 of 10,273 rows scored exactly zero. The list was English. I write German.

How to read it: a scorer that cannot see its user is not under-tuned, it is blind.

12. Does the nudge have a budget?

Count how many times per session your hook interrupts, and rank what it interrupts for.

Mine: none of my hooks had one. 166 of my 303 observable injections were the sixth or later fire in their session.

How to read it: combined with check 8, a hook with no budget spends its credibility early and then fires into a session that has already stopped listening.

Group C: can it fail

13. When did retrieval last return zero?

Count result-set sizes across your whole lookup log.

Mine: 32,254 of 32,614 lookups returned exactly three results. Of the 348 zeros, 327 were crashes. 21 genuine empties, 0.06%.

How to read it: a retrieval layer that never returns nothing has no way to tell you it found nothing. It will hand you the least-bad row instead.

14. How old is what retrieval returns?

Compare the age distribution of what came back against the age distribution of the whole store.

Mine: 99.8% of injected results were under a week old, against an 11% base rate. About 7% of the corpus could ever compete.

How to read it: if the two distributions differ that sharply, recency is the primary sort key and match strength is only a tiebreak.

15. Does the relevance number mean match, or age?

Read the ranking code. Do not assume.

Mine: the exposed score was a decay function of age. Match strength was computed and discarded before output. A relevance floor would have kept the newest rows, not the best ones.

How to read it: a threshold on the wrong quantity makes the problem worse while looking like a fix.

16. Can your guard fail?

Delete the thing the guard exists to catch and re-run the suite.

Mine: deleting a leaking field outright left all 43 tests green. Not one test asserted what a row may contain.

How to read it: if the suite stays green, the guard is decorative. Name an input that would break it, or you do not have one.

17. Is your ground truth circular?

Grep for the call, not for the predicate name. If the thing producing your labels is also the thing being measured, the score is a tautology.

Mine: 126 of my positive labels were produced by the very function under evaluation. Neutering it changed the prediction and the label together.

How to read it: a witness drawn from the same pipeline measures consistency, not correctness.

The three things worth keeping

Count actions, not rows. A log line proves a process ran. It proves nothing about effect.

Name the channel. Every producer and consumer pair needs a stated medium, or the pair can be dead while both halves look healthy.

Give every check three states. True, false, and could-not-determine. Nagios calls the third UNKNOWN. Prometheus keeps it out of the metric's value entirely. Where it is missing, it gets encoded into one of the other two and every downstream reader inherits the ambiguity.

Honest limits

Seventeen checks, one system, three to four months, 47,609 log rows. The path-dependence figure is one hook, one user, 303 observable injections across 68 sessions.

One of these numbers already died in my hands. The position-in-session decline I measured in August did not survive being re-derived on four months of data, and it reversed sign under a different treatment of unobservable fires. I left that story in check 8 rather than deleting it, because the useful part is not the number. It is that a finding which flips on one analysis choice was never a finding.

Nothing here is a benchmark. It is a list of questions I could not answer about my own setup until I went and looked, and every one of them was hiding something.

The hooks, context router and session memory these were measured against live at github.com/primeline-ai/evolving-lite.