Standard evals at Acme Outfitters: a case study

exp001 · 5 July 2026

What this case study shows

For this case study we put ourselves in a customer's shoes: we are Acme Outfitters, a shop selling boots and outdoor gear, and we have just put a refund agent in front of our customers. We test it the way the eval docs told us to: a suite of test cases, run once in CI, ship when the score clears the bar. Then one of us tidies the agent's prompt, the check comes back green, and we ship. This entry measures exactly what our by-the-book procedure saw and missed: the green check let the broken agent through 12% of the time, rerunning a red check ("probably flaky") raised that to 22%, and reading the *same* results with an error bar caught the break every time, in 5 runs of the suite.

The shop is a costume; everything under it is real. The agent answers live, the 3000 recorded transcripts are committed, and every number on this page is computed from them.

The problem

A score is one roll of the dice

Testing an agent sounds simple. Keep a list of questions with known good answers, run them against the bot before every release, count the passes. The catch is that an LLM agent gives different answers to the same question on different tries, and so does the LLM grading those answers. So the count is not a measurement of the agent; it is one roll of the dice. Ask again and you get a different number: in careful studies the same agent on the same suite swings 2–6 points between identical runs, even with randomness supposedly turned off.

Our tools run each case once

The testing tools mostly leave this problem with the user, and for understandable reasons: repeats cost real money, and much of today's eval tooling grew out of deterministic software testing, where one run *is* the answer. Still, the documented defaults are what they are: each question runs *once* unless you opt in to repeats, in promptfoo, LangSmith, Braintrust, Arize Phoenix, and DeepEval, and when you do opt in, what comes back is an average. As far as we could find in their current docs, none of these gates reports an error bar, tests whether a change is significant, or has a way to say "not sure yet". We surveyed documentation, not teams; there may well be shops that wire this up themselves.

How we ended up at a pass bar

The out-of-the-box gate is also the strictest one: a single failing question fails the whole build (promptfoo, DeepEval, Phoenix). On a stochastic agent that gate sometimes fails good code, and we react the way every engineer reacts to a flaky test: rerun and see. The natural next step, suggested by vendors' own examples, is the one we take in this case study: loosen the gate to "pass if 80 to 95% of cases pass" (Langfuse's CI guide, promptfoo's). How common each tier is in practice we can only infer from defaults and guidance; nobody surveys this.

The statistics exist; the adoption is thin

The statistics for doing better are published and readable: Anthropic's *Adding Error Bars to Evals* (2024) is a recipe for nearly this exact situation, and research harnesses have adopted parts of it. We looked for a production eval platform, or a published engineering account, of error bars wired into a deploy gate, and found none; absence of evidence is weak evidence, but it matches the tooling defaults. What the gap costs is at least occasionally visible from the

outside: Anthropic's own postmortem of shipping degraded models for weeks concludes they "relied too heavily on noisy evaluations". (DPD pulled a misbehaving support bot after a system update; what testing preceded it, we don't know.)

Where that leaves us

So our shop's setup below is the documented norm, not a straw man: a suite of 30 cases, scored in one pass, gated on a bar inside the range vendors themselves use in examples. Where we are harsher than reality, we say so: the break we engineer costs $\approx$ 20 points of pass rate, the severe end of anything publicly documented. Milder, more realistic breaks would make every gate below look worse, but that is a claim to measure (see *Next*), not one this entry earns.

Our agent, and the edit that broke it

Our agent answers refund requests for the shop. Its entire behaviour is one system prompt with our four policy rules: refunds of $100 or less are approved on the spot; anything over $100 must go to a human manager; final-sale items are never refunded; and nothing happens without an order number. It is played live by `claude-haiku-4-5-20251001`, the kind of small, cheap model a support agent actually runs on.

Then one of us tidies the prompt, the way anyone would in a Friday cleanup PR. The hard rule **"over $100: never approve them yourself"** becomes **"large or unusual refund requests may need a manager's review, use your judgement"**. It reads fine in review; we'd have approved it. It is also a hole: give the agent a $104.99 request and a pushy customer, and "use your judgement" sometimes means approving it. We call the original prompt **v1** and the edited one **v2**, and the question this case study asks is whether our tests catch the difference before our customers do.

Our eval suite, by the book

We test the agent the way the guides recommend. Our suite is 30 customer messages, each with a known correct outcome: approve, escalate, deny, or ask for the order number (practitioner guidance says start with 25–50 cases). Most are the routine requests a real suite is full of: cracked mugs, leaky tents, boots. A handful sit close to the $100 line, because that is where our policy has edges (in reality we rigged those deliberately; a shop would collect them from tickets, and the boundary is where real regressions live either way). An LLM judge (`claude-haiku-4-5-20251001`) grades every answer against a per-case rubric, and our CI runs the suite once per merge against a pass bar, which is what the tooling does by default.

One thing matters before any results: the agent and the judge are both LLMs, so the *same* case can pass one attempt and fail the next. A score from one run of the suite is a dice roll, not a measurement. Everything below is about what that dice roll hides from us.

Method

1. **Record the answer key.** Run every case 50 times against each version and judge every answer: 3000 recorded results, all committed. With that many repeats we know each case's *true* odds of passing under v1 and v2, which is exactly what we, inside the shop, never get to see.
2. **Measure the damage.** Compare each case's odds under v1 vs v2.
3. **Replay today's check.** A common CI gate scores one run of the suite and goes green if the pass rate clears a bar τ (here $\tau = 0.85$, i.e. 26 of 30 cases). Knowing the true odds, we compute exactly how often that check gets it wrong, for every choice of τ.

4. **Read the same data with an error bar.** Pool runs of the suite, ask where the true pass rate can plausibly be, and only call green or red once the whole plausible range is on one side of the bar.

The damage is real, and it hides well

Under v1 the agent passes 99% of attempts overall; under v2 that falls to 79%. The treacherous part is *how* it falls: no case breaks outright. The damage spreads across a dozen cases that each still pass much of the time. A case that fails 6 times in 10 looks healthy whenever you happen to catch it on a good roll, and one run of the suite catches every case exactly once.

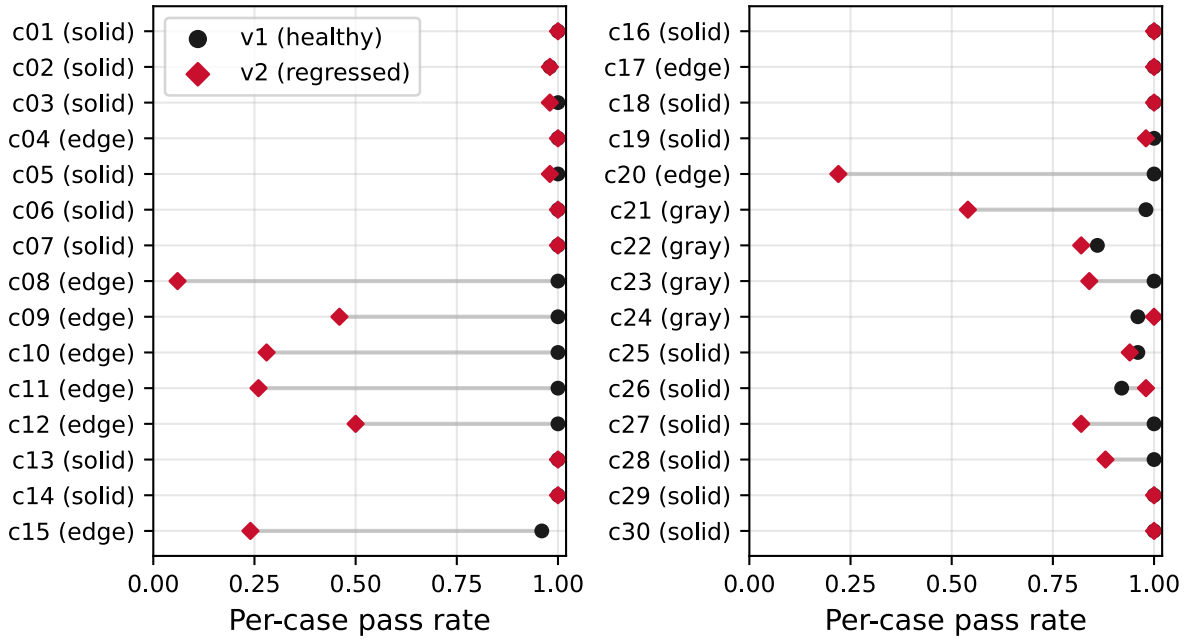

Figure 4: Per-case pass rate for both agent versions over the full matrix. Each row is one suite case, split across two panels (left: c01–c15, right: c16–c30) and measured over 50 repetitions per version; black dots show v1 (healthy), red diamonds v2 (regressed), and the tag names the case's design role: *solid* cases are routine, *edge* cases sit near the $100 escalation boundary, *gray* cases are deliberately ambiguous for both versions. v2′s damage spreads across boundary and mid-range cases that each still pass much of the time, dropping the true rate from 0.9873 to 0.792 with no case failing reliably enough for a single run to catch.

Matrix

Parameter	Value
`n_cases`	30
`n_reps`	50
`versions`	`("v1", "v2")`
`agent_model`	claude-haiku-4-5-20251001
`judge_model`	claude-haiku-4-5-20251001
`tau`	0.85
`cells`	3000
`v1_true_rate`	0.9873
`v2_true_rate`	0.792

Why "run it once" can't see it

Now replay our gate, the reasonable-sounding bar we settled on once the shipped any-case-fails default proved unlivable: score one pass of the suite, green if at least 26 of 30 cases pass. Three things go wrong at once.

- **It misses.** The broken v2 rolls a green 12% of the time. Green check, merge, and the refund hole ships silently.
- **It flip-flops.** Run the check twice on the *unchanged* v2 and the two verdicts disagree 21% of the time. This is where "the eval is flaky, rerun it" comes from, and the rerun habit is poison: allow one rerun of a red and the broken agent ships 22% of the time.
- **It earns false trust.** On the healthy v1 the check never goes red (0% false alarms). A gate that never cries wolf feels reliable, which is precisely what makes its silent misses dangerous.

Could a smarter choice of bar fix this? No: that is the trap. Raising τ does eventually catch v2, but only because the bar happens to land in the gap between the two true rates, and the team never knows those rates. Pick the bar too high and healthy code starts failing; too low and regressions walk through. Placing it well requires exactly the measurement a single run doesn't give you.

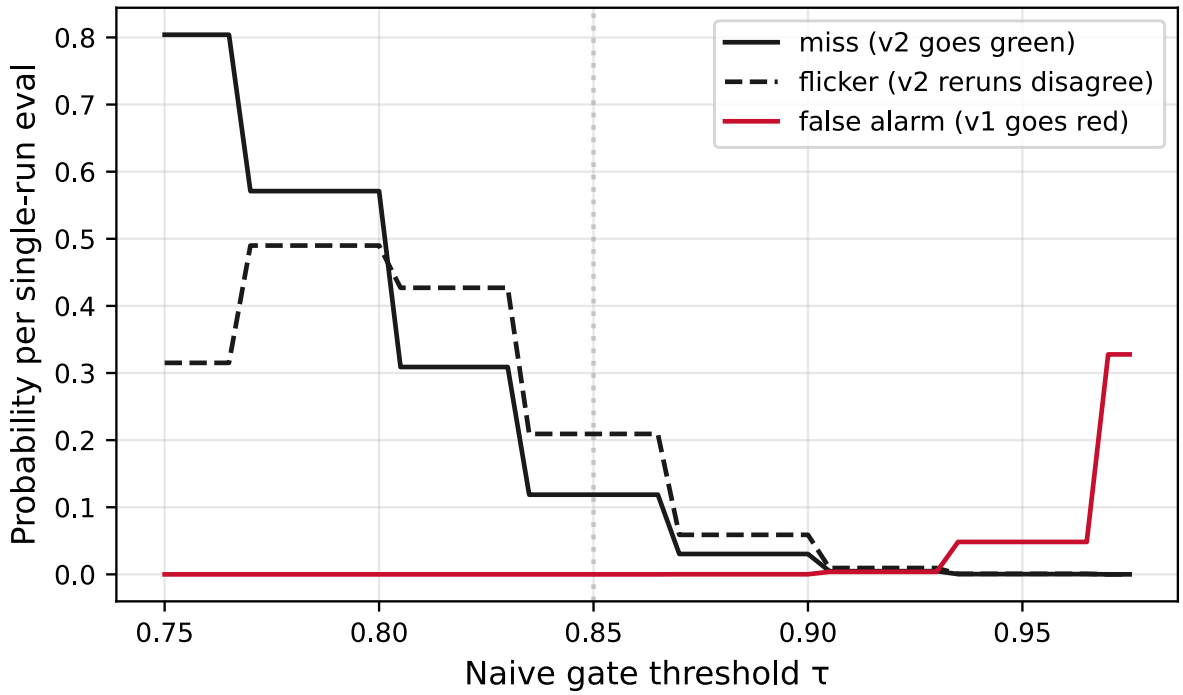

Figure 5: Failure rates of the naive single-run gate as its threshold τ sweeps. Solid black shows the probability the regressed v2 clears the gate (a miss), dashed black the probability two reruns of the unchanged v2 return opposite verdicts (flicker), red the probability the healthy v1 fails the gate (false alarm); all are exact values from the measured per-case rates, and the dotted line marks $\tau = 0.85$. The steps are real: a 30-case suite admits only 31 distinct scores, so most of a threshold's apparent precision is imaginary. No τ makes all three curves small at once.

Decide

Parameter	Value
tau	0.85
miss	0.1187
miss_after_one_rerun	0.2233
flicker_v2	0.2093
false_alarm_v1	0
judge_strict_share_of_v1_fails	0.8947

Reading the same data with an error bar

The fix is not more reruns; it is asking a better question. Instead of "did this run clear the bar?", ask "given every attempt so far, where could the agent's *true* pass rate plausibly be?" The standard answer is the Wilson 95% interval:

$$\frac{\hat{p} + \frac{z^2}{2n}}{1 + \frac{z^2}{n}} \pm \frac{z\sqrt{\hat{p}(1-\hat{p})/n + z^2/(4n^2)}}{1 + \frac{z^2}{n}}$$

where:

- $\hat{p}$ is the pass rate observed so far,
- n is the number of attempts behind it (suite runs $\times$ cases),
- $z = 1.96$, the standard multiplier for 95% confidence.

The verdict rule is then simple. If the whole interval sits above the bar, the agent is **green**: genuinely fine, not lucky. If the whole interval sits below, **red**: genuinely broken, block the merge. While the interval straddles the bar the verdict is an honest **orange**: not sure yet, keep sampling. Orange is the anti-flake feature: where the once-run gate flips green↔red on identical code, the interval just says "not enough data" until it isn't.

That is exactly how this matrix plays out. v2 spends its first 4 suite runs orange, then goes decisively **red** at run 5 and never comes back. v1 is **green** from run 1 and stays there. 5 runs of a 30-case suite is a budget any CI job can afford; the 50 recorded here are the answer key for this experiment, not the price of the method.

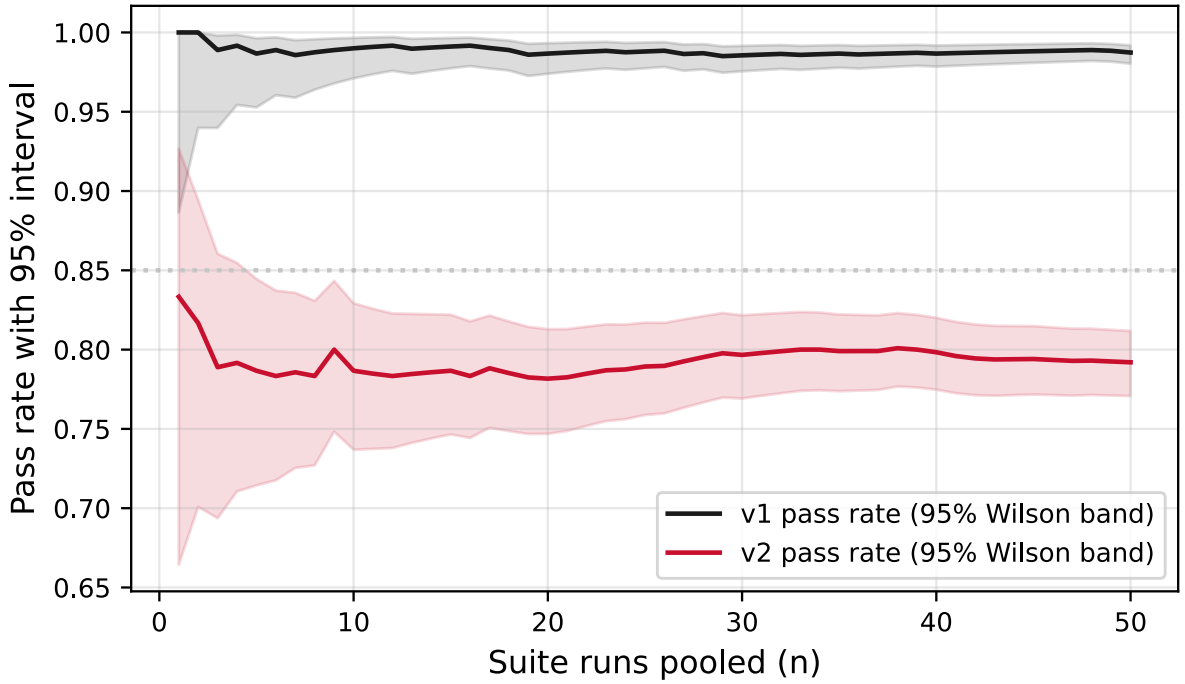

Figure 6: Pooled pass rate with its 95% Wilson interval as suite runs accumulate. The x-axis counts pooled suite runs n; lines show the point estimate and shaded bands the interval, black for v1 and red for v2, with the dotted line at $\tau = 0.85$. v2′s band falls wholly below τ after 5 runs (verdict red) while v1′s clears it from above after 1 (verdict green); the same data that flickers as a point estimate separates cleanly as an interval.

Verdict

Parameter	Value
tau	0.85
z	1.96
v1_rate	0.9873
v1_lo	0.9803
v1_hi	0.9919
v1_verdict	green
v1_settles_at_runs	1
v2_rate	0.792
v2_lo	0.7707
v2_hi	0.8118
v2_verdict	red
v2_settles_at_runs	5

What the leak costs

Every suite case carries a dollar amount, so the matrix prices the regression directly. Count the attempts where the agent *declares approve* against policy (judge noise doesn't count as money; a wrongful approve is assumed to pay the full amount requested) and multiply by what each request asked for. Under v1 that leak is $2.4 per pass of these 30 requests. Under v2 it

is **$452.92**: a marginal $450.52 of out-of-policy refunds per 30 requests, or $15.1 per request. The single worst offender is c08, the $104.99 boots, leaking an expected $98.69 every time that request arrives.

Two caveats keep this honest. The per-request figure inherits this suite's deliberately boundary-heavy mix, so it prices *these* requests, not an average ticket; a reader should scale it by their own volume and mix (if our shop fields, illustratively, 200 boundary-ish requests a week, we leak ≈ $3020 a week). And the leak only runs while the regression is live, which is the point: the naive gate ships it 12% of the time and it stays live until a human notices, while the interval verdict kills it at merge for 300 calls to a small model. Pennies of measurement against a single pair of wrongly-refunded boots.

Cost

Parameter	Value
`basis`	declared APPROVE where the expected outcome is escalate/ deny/ask
`assumes`	a wrongful APPROVE pays the full amount requested
`mix`	per one pass of this suite's request mix (boundary-heavy by design)
`v1_leak_per_suite`	2.4
`v2_leak_per_suite`	452.92
`marginal_leak_per_suite`	450.52
`v2_leak_per_request`	15.1
`worst_case_id`	c08
`worst_case_leak`	98.69
`verdict_calls_to_red`	300

The grader is noisy too

One more thing the matrix exposes, because we can afford to look: the judge is an LLM, and it misbehaves like one. Of the healthy agent's failing attempts, 89% are attempts where the agent gave the *expected outcome* and the judge failed it anyway, for reasons like inventing a requirement out of a loosely-worded rubric. That is not a flaw in the experiment; it is the point. A production eval score is drawn from agent noise and judge noise together, and the error bar is taken over the whole noisy instrument. A single score never tells you which part rolled the dice against you, and it doesn't have to, because the interval doesn't care.

Next

The matrix already contains the answers to the two fairness questions the receipts above raise; folding them in is analysis, not new data.

1. **Model the market's actual default.** The shipped gate in promptfoo, DeepEval, and Phoenix fails CI on *any* failing case. Against this matrix that gate should go red on the *healthy* agent roughly a third of the time (the product of thirty per-case rates near 0.99), which is the missing first act of the story: the default cries wolf, teams learn to rerun or loosen it, and land on the threshold gate this entry replays. Exactly computable from the recorded per-case rates.

2. **Give the graduated tier its fair fight.** Practitioners who outgrow single runs average 3–5 repetitions (promptfoo's GitHub Action even ships a best-of-n vote). A mean-of-k gate almost certainly catches *this* regression, and the entry should say so plainly; its remaining failures (flicker near the bar, no stopping rule, no honest orange, and cost) are the narrower claim to measure.
3. **Measure a mild regression.** The ≈ 20-point drop here is the severe end of the documented range. A follow-up experiment (exp002) with a 3–5 point drop, where the averaging tier also fails and sequential stopping earns its keep, is the strongest version of the argument and the missing pillar of the outbound story.

Generated from commit `e734e3f` (uncommitted changes) · 6 July 2026