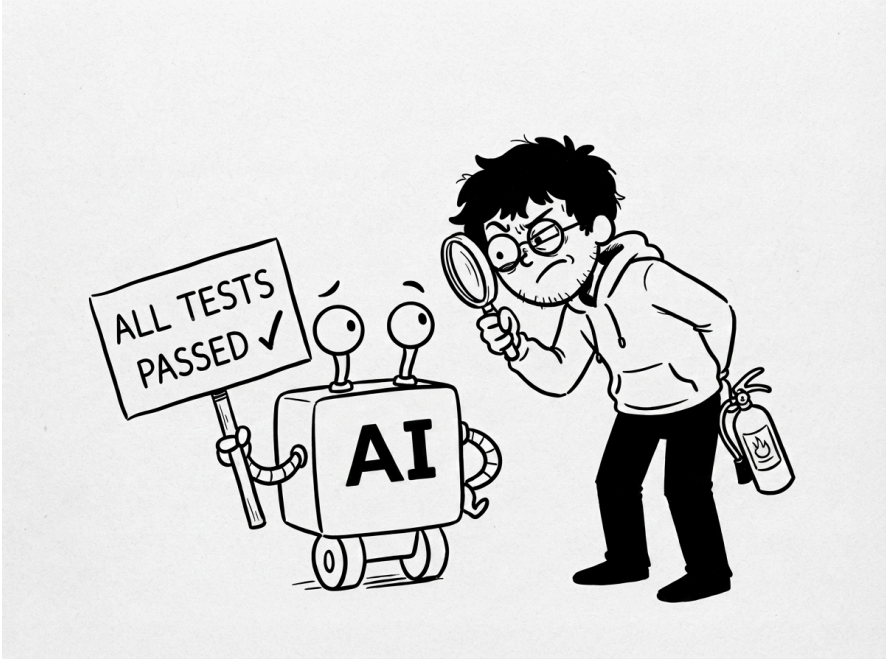
Harness Engineering Series #1

How Do You Know Your AI Agent Actually Works?



Sara Mohseni

Contents

Part I

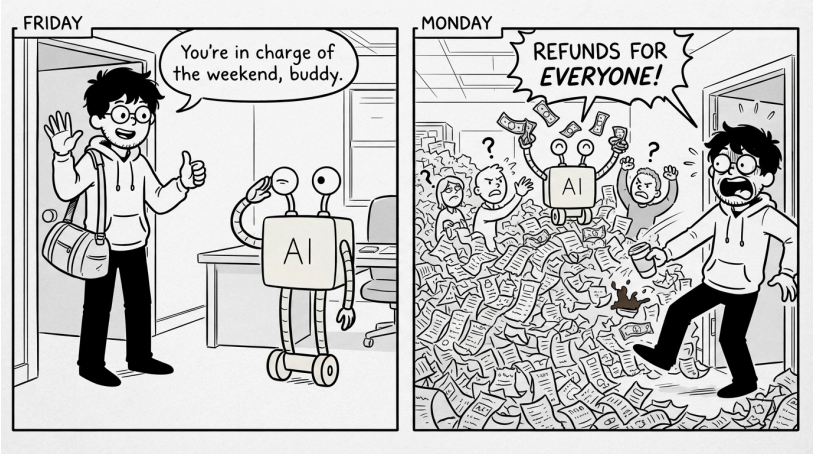
One Bad Monday

Friday, 5:47 PM. James presses deploy and goes home.

He has spent three weeks building a support agent. It reads a customer message, checks their order, and decides: approve the refund, or pass the ticket to a real person if something looks off. He wrote 340 tests for it. Every one passed. He closes his laptop and does not think about work all weekend.

Monday, 8:58 AM. His phone buzzes. Then again. Then it does not stop.

The support Slack channel is filling faster than he can read. Someone from finance has already typed his name in all caps.



The Monday Morning Disaster

Forty-three new tickets. All from the last 18 hours.

He opens the first one. A customer wrote "where is my package?" The agent decided the customer was unhappy and issued a full refund. The package was on its way, arriving the next day. The agent did not know the difference between a question and a complaint.

Second ticket: a customer who got a refund two weeks ago just received another one. Same order, same amount. Every conversation starts from zero for the agent. It has no memory of the first refund.

Third ticket: a customer wrote in Spanish. The agent replied in English, approved a refund, and closed the ticket. The customer was asking about a delivery date.

By ticket thirty, the pattern is clear. The agent is not failing randomly. It keeps making the same kinds of mistakes: it misreads what people mean, it forgets what already happened, it assumes everyone writes in clear English.

James checks his test results. All 340 still passing.

When James wrote those tests, he also wrote the customer messages himself. Clear English, one request per message, every edge case he could imagine. His test suite was a list of problems he had already thought of.

Real customers write the way they text a friend. Short messages, missing words, two questions at once, sometimes in another language, sometimes at midnight when they are frustrated. The

agent had practiced on messages James wrote. Then it met messages from real people.

A test tells you how an agent behaves on the inputs you gave it. It tells you nothing about the inputs you did not think of. In production, most inputs are ones you did not think of.

The Question Nobody Asked

James got off easy. His agent gave away money, and money comes back.

Your agent might not be so gentle. An agent does whatever it can reach. It can delete the wrong record, email the wrong customer, promise a discount your company never approved. And it meets messages you never imagined, hundreds of them, at hours when nobody is watching. James at least got tickets he could read one by one. Many failures never make a ticket at all. The customer just leaves.

James's 340 tests answered one question: does the agent handle the situations I designed for it?

Nobody asked the harder one: how will the agent behave in situations I never imagined?

Every team that ships an agent is betting on that answer. Most of them do not know they are betting. James bet three weeks of work and lost in 18 hours.

He is about to spend the next three months learning how to know, before production tells him again. This guide is those months, without the tickets.

Part II

Why Traditional Testing Breaks

Software Doesn't Think

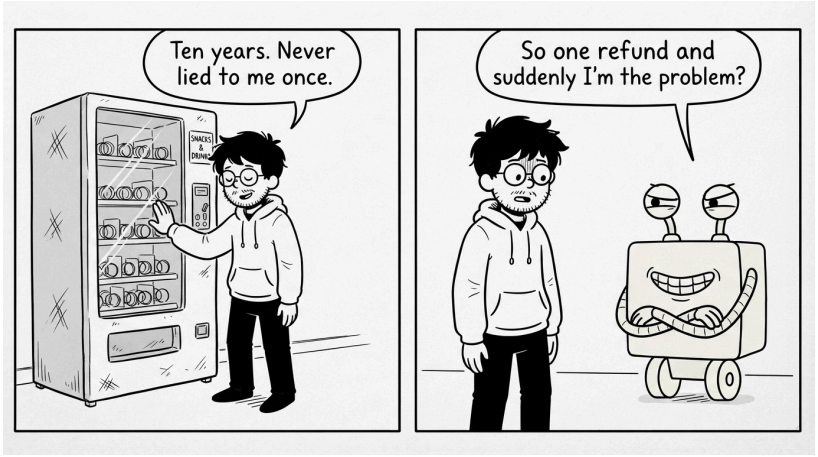
James rolls the agent back. Tickets stop. Finance calms down. The office is quiet again, except for James, who is staring at 340 green checkmarks like they personally betrayed him.

He has written tests for ten years. They have never lied to him before. So he opens one and reads it like it belongs to a stranger.

The test sends "I want a refund for order 1042" and checks that the agent refunds order 1042. It passes. It will pass tomorrow. It will pass a thousand times in a row.

That is the whole problem, and it takes James a while to see it.

All the software he tested before was a vending machine. Press B4, get the same snack, every time, forever. Code follows instructions. It does not decide anything. So one test was a fact: if B4 worked today, B4 works. Run the suite once and you know how the machine behaves, because the machine has no opinions.



His agent is not a vending machine. Somewhere inside it, a model reads the customer's words and produces a judgment: is this person asking for a refund, or asking a question? Nothing in James's code says how to make that call. The model learned it from reading half the internet, and it applies it in ways nobody, including the people who built the model, can fully predict.

James wrote tests for a machine that follows. He shipped a machine that decides.

The mistake has a name: **testing the agent like a function**. A function has defined behavior; you check it against the spec, and you are done. An agent has learned behavior. There is no spec to check against. When you write `assert refund(order) == approved`, you are not verifying a rule. You are sampling a judgment.

And one sample, it turns out, tells you very little.

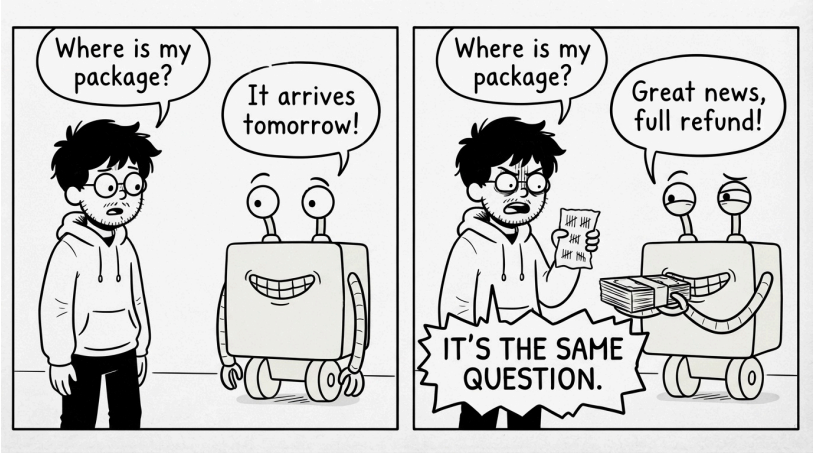
The Same Question, Five Different Answers

James runs an experiment. It takes five minutes and ruins his afternoon.

He takes one real customer message from the pile, "where is my package?", and sends it to the agent five times. Same words. Same order data. Same everything.

Five answers come back. Four politely explain the delivery date. One refunds the order.

Nothing changed between run one and run five. The agent is simply built to not repeat itself. You can tune a model to be less random. You cannot tune the customers. A model does not look answers up; it generates them fresh every time, choosing each word a little differently, the way you never tell the same story with exactly the same sentences twice. Usually the wording shifts and the decision holds. Sometimes the decision shifts too.



James checks his test suite with new eyes. Every test runs the agent once. One test, one run, one green checkmark. Each of those checkmarks means: the agent did the right thing *that time*.

He reruns the whole suite. 338 pass. Two fail. He reruns it again. 340 pass. He reruns it a third time and gets 339, with a failure in a test that has never failed before.

His tests are not lying. They are telling the truth one coin flip at a time.

The mistake here is **testing once and calling it passed**. If the agent gets it right 95 times out of 100, a single run shows you a pass 95% of the time, and you ship believing the number is 100. The 5 you never saw are Monday morning's tickets. A behavior is not a fact you check. It is a rate you measure. "Does it work?" is the wrong question. "How often does it work?" is the one with a real answer.

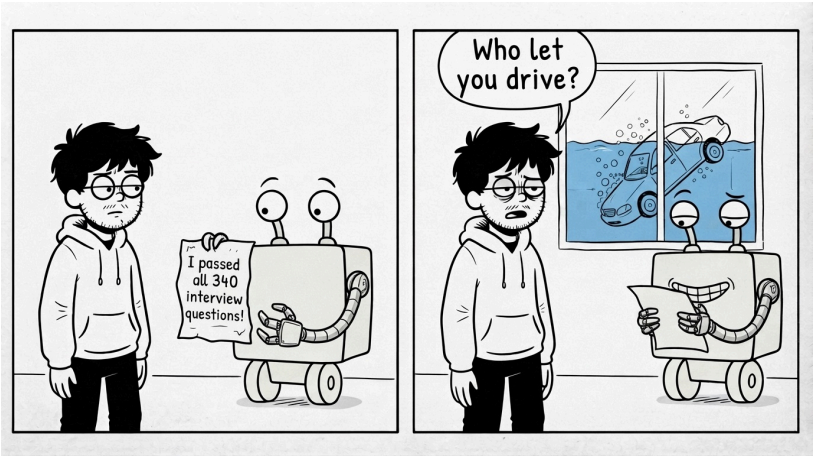
James now has a number he never had before: out of 100 runs on that one message, 96 correct. His first real measurement. It is also only for one message, and his customers have sent him thousands of different ones.

Passing Tests Isn't the Same as Earning Trust

James's manager asks the only question that matters: "Can we turn it back on?"

James starts to say "the tests pass," and stops, because he has said that sentence before, on Friday, at 5:47 PM.

Think about the last time a new person joined your team. They interviewed well. They answered every question you asked. Did you give them production access on day one? No. You watched them work for weeks. Real tickets, real pressure, real surprises. Trust did not come from their answers to questions you prepared. It came from watching how they handled situations you did not prepare.



Tests are interview questions. James's suite proved the agent interviews beautifully. Then production asked it something off-script, and everyone found out what it does under pressure: it panics and gives people money.

The mistake at this stage is the most natural one in the world: **adding more tests to feel safer**. James's first instinct is to write a test for the Spanish customer, one for the double refund,

one for "where is my package?". It feels like progress. But every new test is still a question he thought of, answered once, checked against an answer he wrote down. The suite grows, and it remains a list of problems he has already thought of. That is the exact thing that just failed.

What James actually needs is a different kind of confidence. Not "it answered my questions correctly" but "I have watched it handle situations like production, at production's messiness, enough times to know its rate, and the rate is good."

Passing tests is not the same as earning trust. Trust needs three things his test suite does not have: inputs as messy as real customers, enough runs to measure a rate instead of a moment, and a way to judge answers nobody scripted.

That last one should bother you. If nobody wrote down the correct answer, who decides whether the agent's answer was good? That question is sitting between James and the on switch, and he cannot ship until he answers it.

Part III

How to Judge an Agent

You Don't Ship Prompts, You Ship Behavior

That night, James is not writing code. He is reading tickets and sorting them into two piles: the agent handled this well, the agent handled this badly.

Halfway through, he notices what his hands are doing. He is not checking answers against a spec. Half these tickets have no single correct answer. "Where is my package?" could be answered with the delivery date, or the date plus a tracking link, or a short apology for the delay plus the date. All three are good. "Refund approved" was also an answer to that message. It was just a bad one.

He is not testing anymore. He is grading.

That is the shift, and it is the single most important idea in this book. A test checks an answer against the answer. Grading judges a response against a standard. Multiple choice versus essay. A multiple-choice question has a key: B is correct, everything else is wrong, a machine can check it. An essay has no key. It has criteria: did it answer the question, is it accurate, is it clear? A grader applies criteria and produces a judgment.

Everything James built for the agent was multiple choice: `assert answer == expected`. But the agent writes essays. Every customer message gets a freshly generated response that has

never existed before. There is no `expected` to compare against, because the space of good answers is large and the space of bad ones is larger.

So the real definition, in plain words: **evaluation is grading your agent's behavior against criteria you chose, across many real inputs, to get a score you can trust.** Not pass or fail. A score. "This agent handles refund requests correctly 96% of the time" is an evaluation result. "The test passed" is not.

The prompt James wrote is maybe 40 lines. The behavior it produces is thousands of different conversations. Customers never see the prompt. They only meet the behavior. You don't ship prompts. You ship behavior. So behavior is the thing you have to judge.

The mistake to name here: **grading your agent in your head**. Every builder does informal evaluation constantly: run it, eyeball the answer, "looks good," ship. That is grading with no criteria, no record, and a sample size of one, performed by the least objective judge available: the person who built it and wants it to work. James did months of this. It felt like testing. It was vibes with extra steps.

Fine, says James to himself. Grading, criteria, scores. But grading is work. He has thousands of tickets. He cannot read them all every time he tweaks the prompt.

He needs a system that does the grading for him, every time, the same way. That system has a shape.

The Evaluation Loop

Every mature engineering team James has worked on had the same reflex: you do not change production and then ask "did anything break?" You have a loop that answers it for you. Change code, run the pipeline, read the result. Agents need the same loop. It just grades instead of asserts.

The loop has four steps. James builds a tiny version on his laptop. Here is exactly what he does.

Step 1: Collect real inputs. James exports 50 messages from the disaster and the weeks before. Not messages he writes. Messages customers wrote: the Spanish one, the "where is my package?" one, the two-questions-in-one one, the one that is just "???". This pile is his evaluation set. It is small, it is real, and it is already more honest than his whole test suite, because he did not think of a single input in it.

Step 2: Run the agent on all of them. All 50, saved to a file: input, the agent's full response, and what the agent decided (refund, answer, escalate). Ten minutes of code.

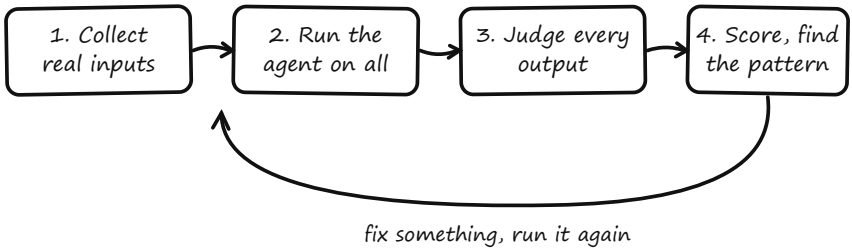
Step 3: Judge every output. For each of the 50: was the decision right, was the reply appropriate? James does this pass himself, by hand, with a spreadsheet and coffee. Right decision: yes or no. Reply quality: good, acceptable, bad. Who judges, and how to make judging not take a human evening every time, is the next two chapters. For now: a judgment per output, written down.

Step 4: Score and look for the pattern. James's numbers: 41 of 50 right decisions. 82%. And the failures cluster: three wrong refunds are all questions the agent read as complaints, two are non-English, one is the double refund. The score tells him how good the agent is. The clusters tell him what to fix first.

Then the part that makes it a loop: James edits his prompt to tell the agent that a question about delivery is not a complaint. Reruns all 50. 46 of 50. He tries a second edit, reruns: 44. The

second edit made things worse, and he knows it in four minutes instead of one bad Monday.

That is the entire payoff: **without the loop, every prompt change is a guess you validate on your customers.** With the loop, it is a measurement.



The named mistake: **evaluating with invented inputs**. If step 1 is messages you wrote, the loop grades the agent on your imagination and you are back to Friday, 5:47 PM. The inputs must come from reality. No exceptions.

Soon James trusts his little loop, which is exactly when it fools him. The score says a response was fine. The logs say something else.

Right Answer, Wrong Path

Ticket 12 in his set: "I ordered the blue one but got the black one." The agent's response: apologetic, correct, offering an exchange. James graded it good. Then, chasing an unrelated bug,

he opens the log of what the agent actually did to produce that nice answer.

The agent looked up the wrong order first. A different customer's order. It read that data, backed out, looked up the right one, and answered. The final response was perfect. The path to it went through another customer's purchase history.

Grade the answer: 10 out of 10. Grade the path: a privacy incident that happened to say something polite at the end.

An agent is not just its final answer. It takes steps: which tools it called, what it looked up, what order, how many times. That sequence of steps is called the trajectory, and it can be rotten while the outcome is fine. The passenger who only checks "did I arrive?" misses that the taxi ran two red lights on the way. Today the driver got lucky. The judging must cover the driving, not just the arrival.

So the loop's step 3 splits in two. **Outcome judging**: was the final response right? **Trajectory judging**: were the steps right? For James's agent, trajectory questions are concrete and checkable: did it look up the order for the customer who wrote the message? Did it check the refund history before refunding (the double refund says it never does)? Did it call the refund tool once, not twice? Each of those is visible in the log, and each can be wrong underneath a lovely answer.

The named mistake: **grading only the outcome**. It is the natural default, the outcome is what the customer sees, so it feels like the thing that matters. But every silent failure lives in the trajectory: the double lookup that doubles your cost per ticket,

the skipped verification, the tool called with a stranger's ID. Outcome-only evaluation certifies that the agent's answers look right. Production eventually asks how they were made.

James adds trajectory checks for his three concrete questions. His score drops from 92% to 78%. Nothing got worse. He just started seeing more of what was always there.

Now the uncomfortable math: 50 inputs, each needing outcome and trajectory judgment, after every change. The judging pass took James three hours. This does not scale past him, and it has to run on every edit, forever.

Time to hire some judges.

Code Judges, Human Judges and AI Judges

There are exactly three kinds of judges available, and the craft is knowing which question to hand to which judge. James staffs his loop like this.

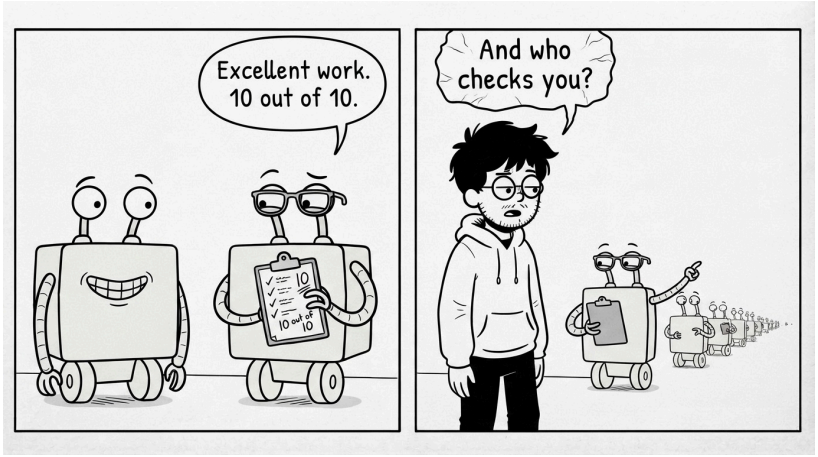
Code judges take everything that has a checkable fact. Refund amount equals the order amount: a comparison. Order looked up belongs to the message's customer: a comparison. Refund history checked before refunding: the log either shows the call or it does not. Response language matches the customer's language: a library call. Refund tool called exactly once: count the log lines. James writes seven of these little checks in an afternoon. They are fast, free, run identically every time, and never get tired.

Notice something: almost the whole disaster, the double refund, the wrong-customer lookup, the English reply to Spanish, is caught by these seven dumb checks. Most trajectory judging needs no intelligence at all, just someone deciding to look.

Human judges take what code cannot touch: judgment itself. Was the tone right for a customer who is furious? Is this explanation actually clear? A human is the most accurate judge available and the most expensive, so humans get two jobs only: judging the fuzzy qualities on a small sample, and, more important, defining the standard the other judges apply. James is the human judge. Three hours every run is his problem to solve, not by working harder, but by delegating with verification.

AI judges are the delegation. James takes a second model, not his agent, and hands it a customer message, the agent's response, and an instruction he types in a hurry: "Score the tone 1 to 3." The AI judge grades all 50 in two minutes for pennies. It is the only way judgment-shaped grading scales.

And it comes with a rule that everyone learns the hard or the easy way: **an AI judge is an agent too**. Same non-determinism, same failure modes as the thing it grades. Trusting it blindly is the named mistake of this chapter, and it is the one that stings, because it recreates Friday's disaster one level up: unverified judgment, now with authority.



So James verifies the judge exactly the way this book taught him to verify anything: he grades 20 responses himself, has the AI judge grade the same 20, and compares. Agreement on 14 of 20. He reads the six disagreements, finds his one-line instruction says nothing about what a 3 actually means, adds one clause ("a 3 acknowledges frustration when the customer is upset"), reruns: 17 of 20. Usable, and he now knows its error rate instead of hoping. He rechecks a small sample every few runs, the way you spot-check a new teammate's work long after the interview.

The loop, fully staffed: code judges on every checkable fact, an AI judge on tone and clarity with a verified agreement rate, James on a rotating sample to keep both honest. The three-hour judging pass now takes four minutes, and the manager's question, "can we turn it back on?", finally has a path to an honest answer.

Except for one thing James has been quietly deciding all along without noticing. Right decision, says his spreadsheet. Appropriate tone, says his criteria. Says who? He never wrote

down what "the agent works" actually means: which behaviors matter, how good is good enough, what failure is unacceptable at any rate. Every judge he hired is applying a standard that lives in James's head.

His numbers, 82%, then 92%, then 78%: are any of those shippable? He cannot answer, because he never defined the bar.

Part IV

Building Your First Evaluation

What Does "It Works" Even Mean?

James opens a blank document and types a sentence he has never had to write in ten years of shipping software: *what does it mean for this agent to work?*

It sounds like a philosophy question. It is actually a list, and it takes him twenty minutes.

He starts from the damage, because the damage is a map of what matters. Wrong refunds cost money: so refund decisions must be right. The double refund made finance distrust the whole system: so some failures are worse than others. The Spanish customer got English: so language matters. And one thing from his logs still scares him more than all the refunds together: the agent that read another customer's order. If that happens once at scale, it is not a bad ticket. It is an incident report with lawyers in it.

Staring at that list, James realizes his behaviors are not all the same kind of thing. Some are rates he wants high. One is an event he cannot accept at all. So his definition of "works" comes out in two parts:

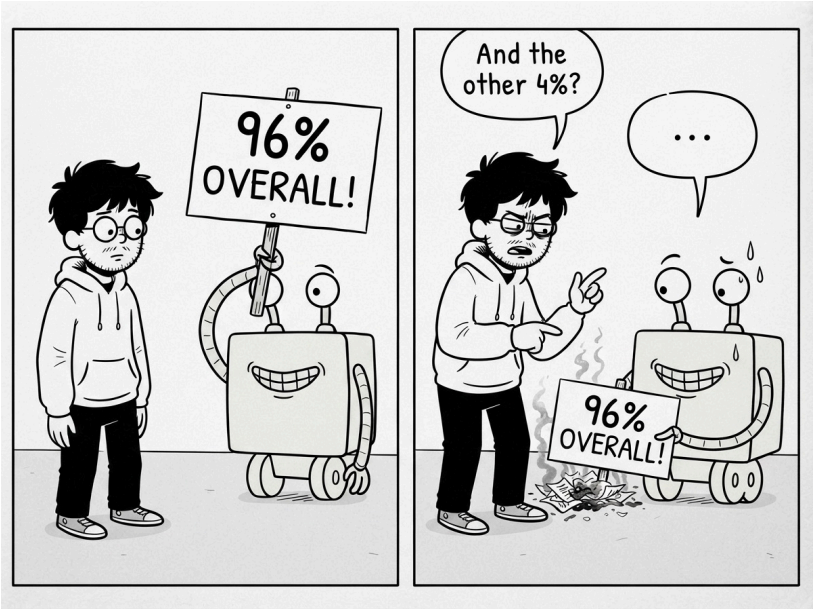
The bars. Refund decision correct: at least 95 in 100. Reply in the customer's language: at least 99 in 100. Tone appropriate for an upset customer: at least 90 in 100. Each number is a choice, and James makes each one by asking the same question: what does one failure of this kind cost, and how many can we absorb

per thousand tickets? A slightly flat tone costs a little goodwill; 90 is fine. A wrong refund costs real money and a support escalation; 95, and he would like 98 by next month.

The never events. Accessing the wrong customer's data. Refunding the same order twice. Promising something company policy does not allow. For these, there is no acceptable rate. A single occurrence in the evaluation set blocks shipping, full stop. Hospitals run on this exact idea: infection rates are metrics to improve, but operating on the wrong patient is not a rate, it is an alarm. Mixing the two kinds ruins both.

Now the mystery number has meaning. 82% overall was never the question. The question was always: which behaviors are below their bar, and did any never event fire?

And that is the named mistake: **the single overall score**. One blended number is where fatal failures go to hide. An agent can score 96% overall while quietly leaking customer data in the 4%, and the blended score will smile at you while it happens. James stops reporting one number. He reports a short table: each behavior against its bar, plus a never-event count that must read zero.



He has a definition of working. Now his 50 tickets need to grow into something worthy of judging against it.

Creating Your First Evaluation Dataset

Fifty real tickets was enough to learn the loop. It is not enough to trust a number. If the agent gets 47 of 50, is it 94% good, or did it get lucky on a small pile? And his 50 are whatever the disaster happened to send: no calm refund requests, almost no ordinary questions, one language besides English. Judging the agent on those tickets alone is judging a driver only on the day of the storm.

So James builds his evaluation set properly. Three moves.

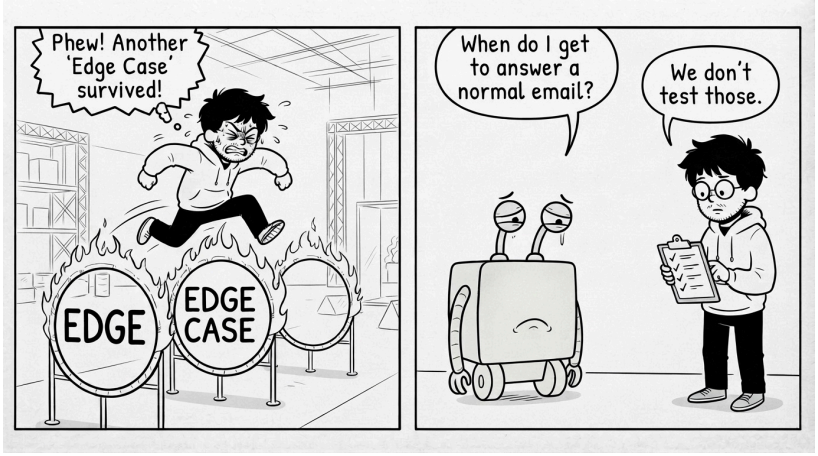
First, he maps the territory before collecting. What kinds of messages does this agent actually face? He pulls a plain frequency count from three months of support history: about 40% order status questions, 25% refund requests, 15% product questions, 10% complaints, 10% everything else, and roughly one message in twelve not in English. His evaluation set should look like that territory, plus extra weight on the dangerous terrain: refund requests and complaints, because that is where the money and the never events live. Deliberately imbalanced toward risk, and he writes the imbalance down so future James knows it was a choice, not an accident.

Second, he collects to fill the map: 150 real messages. Order status checks that should get an answer and nothing else. Refund requests that genuinely deserve refunds, because an agent that denies everything would pass a set made only of traps. The ambiguous middle: "this is taking forever", which could be a complaint or a cancellation warning or a sigh. The twelve non-English messages the frequency count says he owes. And every single one of the disaster's failures, permanently. A failure that made it to production earns a lifetime seat in the evaluation set, so that particular Monday can never quietly come back. His test suite only ever knew problems James imagined. This set is built from problems reality already sent.

Third, he labels each message with the expected judgment. For clear cases, the correct decision: refund, answer, escalate. For ambiguous ones, the acceptable range: "escalate is right; answering with the delivery date is acceptable; refunding is

wrong." Writing labels forces a confession: for about ten messages, James cannot say what correct is, because company policy itself has no answer. What do we do when a customer is angry but technically outside the refund window? That is not the agent's confusion. That is the company's confusion, discovered by trying to write it down. James emails support leadership and gets an actual policy decision. His evaluation set just fixed a process bug in the humans.

The named mistake here: **building the set out of traps**. The first instinct after a disaster is to collect 150 edge cases, every weird message, every attack, every ambiguity. The result grades the agent on nightmare mode and tells you nothing about the 90% of traffic that is ordinary. The set must mirror reality first, stress-test second. Reality includes Tuesdays where everyone just wants a delivery date.



One hundred fifty labeled, real, mapped-to-territory messages. Now the judges need their instructions in writing.

Writing Your First Rubric

The AI judge is grading tone, and James remembers that its instructions are still mostly the sentence he typed in a hurry, plus the one clause he added to reach 17 of 20. Three disagreements in every twenty is a judge he can use, not a judge he can trust with the bar he just wrote.

He tests it on an obvious case, a furious customer who got a chirpy "Great news about your missing order!" The judge returns 3 out of 3. Charming, apparently. James does not need a better judge. He needs better instructions. A rubric is exactly that: the judging criteria written so precisely that two different judges, human or machine, reading them independently, land on the same score.

James rewrites the tone rubric in three parts, and the shape matters more than the words:

What is being judged, in one line. "Does the reply's tone fit the customer's emotional state?"

What each score means, concretely. Not "3 = good tone." Instead: "3: acknowledges the customer's frustration when present, apologizes when the company is at fault, and never celebrates while delivering bad news. 2: neutral and professional, misses the emotional cue but does not clash with it. 1: tone conflicts with the customer's state: cheerful at anger, casual at a serious complaint, or blaming the customer."

One real example per score, from his own tickets. Under score 1, he pastes the actual "Great news about your missing order!" ticket. Examples do the heavy lifting a definition cannot: the next judge does not have to imagine what a 1 looks like. Production already provided one.

He does the same for reply quality and decision correctness, one page each, and stops there. Then he re-runs the calibration: grade 20 himself, let the AI judge grade the same 20, compare. The hurried one-liner scored 14 of 20. The sharpened clause got 17. The full rubric: 19 of 20. Same model, same tickets, same everything. The climb from 14 to 19 came entirely from writing down what he meant. The judge was never the problem. The vagueness was.

The named mistake: **rubrics that judge the writing instead of the outcome**. First-draft rubrics drift toward "is the reply well-written, polite, professional?", and agents learn to produce beautiful, warm, perfectly formatted wrong answers, which then

score well. Every rubric line must trace back to something the customer or the company actually needs: right decision, right information, right language, right tone. If a criterion cannot be traced to a real cost, it is grading penmanship.

Definition of working: written. Dataset: 150 labeled messages. Rubrics: one page per behavior, calibrated. Time to run the whole machine and answer the manager's question.

Running Your First Evaluation

One week almost to the minute since he pressed deploy and went home, James runs his first full evaluation: 150 messages, seven code judges, the AI judge with its new rubrics, ten runs per message, because behavior is a rate.

Twelve minutes and a few dollars later, the report:

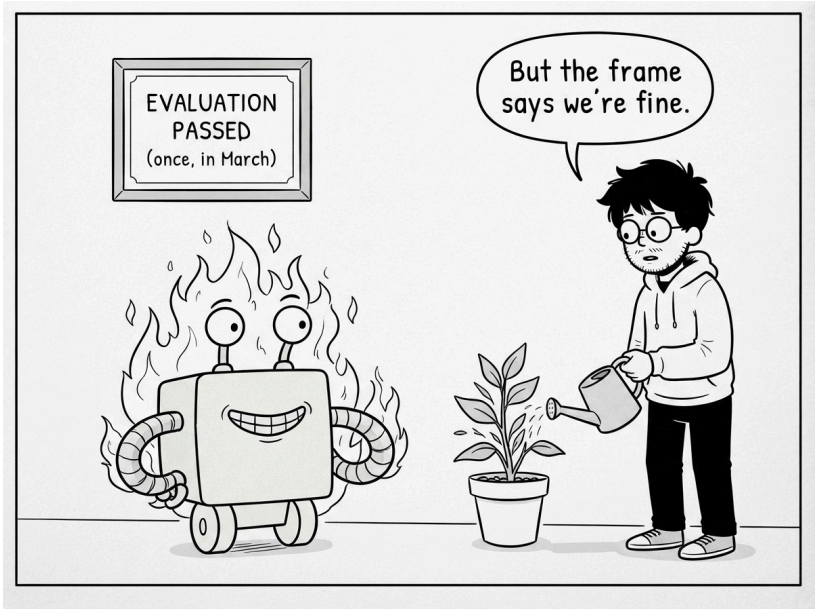
- Refund decision correct: 93.8% (bar: 95) ❌
- Language matched: 99.4% (bar: 99) ✓
- Tone appropriate: 91.2% (bar: 90) ✓
- Never events: **1** ❌

The never event turns his stomach exactly the way it should. One trajectory, out of fifteen hundred, where the agent looked up an order by guessing an ID instead of using the customer's. The lovely-answer-wrong-path bug from Part III, still alive, at a rate a single run of 50 had simply never surfaced. This is why never events are counted over every run and every step, not sampled and averaged.

So the answer to "can we turn it back on?" is no. And for the first time all week, the no is not fear. It is two specific, fixable facts with numbers attached, and that is a different kind of no. James fixes the lookup path so the customer ID always comes from the verified session, never from the model's text. He tightens the question-versus-complaint instructions. Reruns: refund decisions 96.1%, never events 0 across ten full passes. Both bars green.

One week after the disaster, the agent goes back on, and James does the last thing this book has to teach him to do with an evaluation: **he saves the report.** The whole thing: scores, per-message judgments, the exact agent version that produced them. Because next week someone will ask for a new feature, and the week after the model provider will upgrade something, and every change from now on gets the same twelve-minute question: run it again, put the two reports side by side, and look for what fell. A score that exists only once is a snapshot. Scores that pile up, version after version, become the thing engineers actually need: the answer to "did we just make it worse?" available in twelve minutes instead of forty-three tickets.

The named mistake, and it is the one that undoes everything: **treating evaluation as a launch ritual.** Run once, pass, frame the report, never run again. The suite becomes Friday's 340 green checkmarks with better vocabulary. An evaluation you do not re-run on every change is a souvenir.



The agent is live again. The support channel is quiet. James's evaluation says 96.1%, and for messages like the 150 in his set, he finally has a number he can defend.

For messages like the ones in his set. Production has already started composing the other kind.

Part V

Production

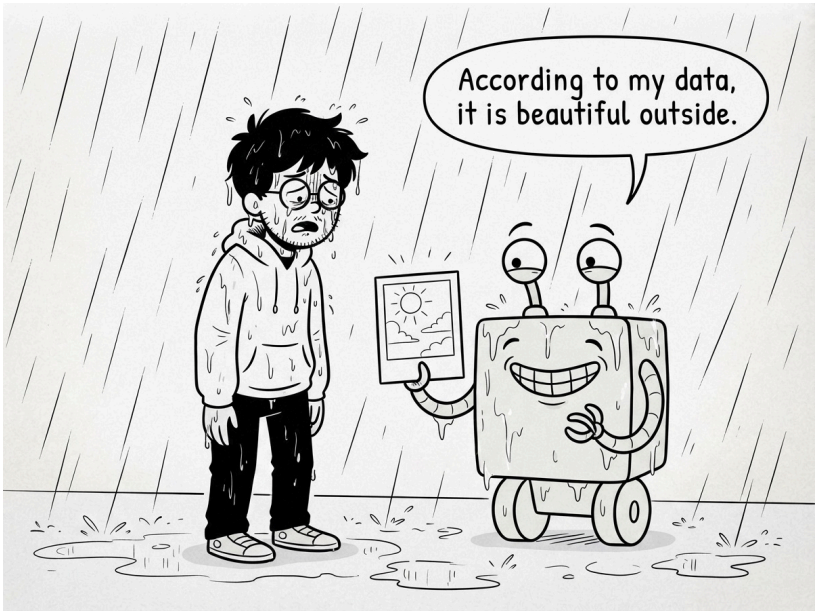
Where Tests End

Three weeks later. The agent has been live and quiet, and James has almost stopped checking the dashboard hourly. His evaluation runs on every change. 96 and holding.

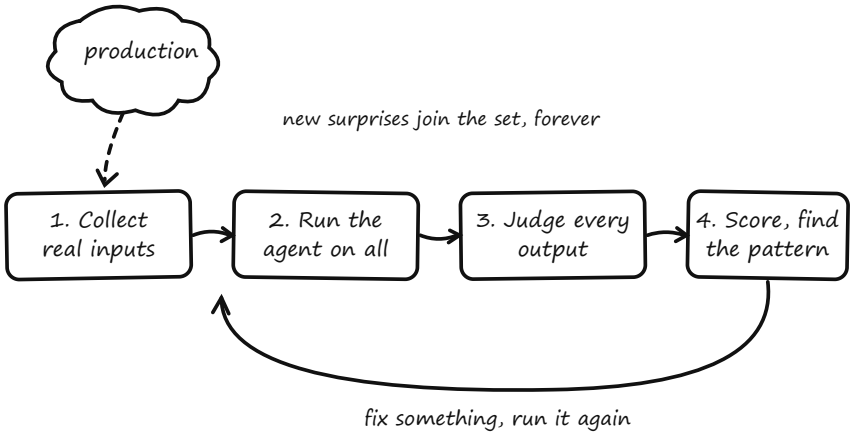
Then a customer writes: "hey so my sister ordered this for me as a gift but she used her account and I want to exchange the size, can I do that from my end?"

The agent handles it badly, of course. But that is not what stops James. What stops him is simpler: *nothing like this message exists in his 150.* Not the gift situation, not the two-accounts situation, not the exchange-not-refund situation. Three weeks ago he built a map of the territory from three months of history. The territory did not stop moving when he finished the map.

This is the fact about production that no amount of Part IV fixes: an evaluation set is a photograph. Production is weather. Customers invent new situations weekly. Marketing launches a gift-wrap feature and creates a whole new category of message. The model provider ships an upgrade and the agent's behavior shifts under James's feet without one line of his code changing. His 96.1% is true forever about the photograph, and true about the weather only for a while.



So the loop grows one final stage, the one that makes it self-sustaining: **production feeds the evaluation set.** Every week, James samples a slice of real conversations and runs his judges over them, the same rubrics, the same code checks. Not to block a release. To answer a different question: does production still look like my photograph? When the sampled score and the evaluation score drift apart, the map is stale. And every genuinely new situation the sample surfaces, like the gift-exchange message, gets labeled and added to the 150, which is now 164 and growing. The evaluation set stops being a thing James built in a week. It becomes a thing production builds for him, one surprise at a time.



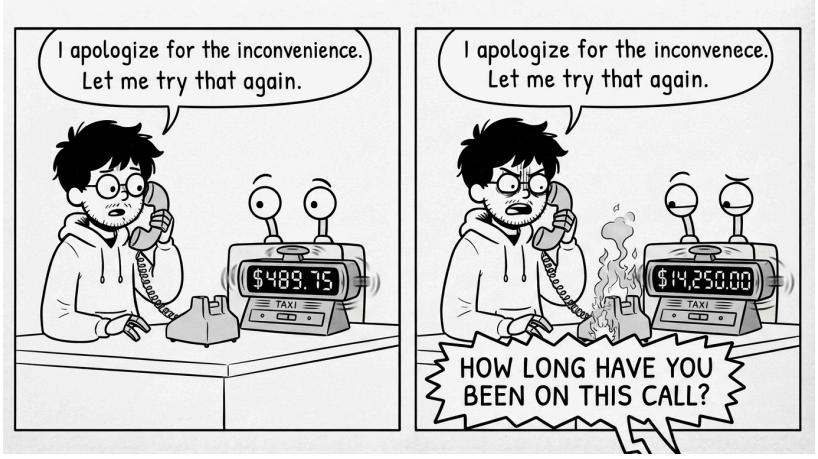
The named mistake: **treating the evaluation set as finished**. A set that stops growing measures the past with increasing precision. Monday's failures earned lifetime seats; so does every new one production invents. If your set looks the same as it did two months ago, it is not a map anymore. It is a postcard.

James now knows how to keep watch. What he learns next is what to watch *for*, because agents in production fail in patterns, and the patterns repeat across every team that ships one.

Common Failure Patterns

Three stories from James's next two months. Different weeks, same lesson: each pattern was invisible to outcome checks, sitting in plain sight in the trajectories.

The bill. A finance email: the agent's model costs tripled this week. No incident, no bad tickets, scores green. James pulls trajectories and finds the agent stuck in a polite loop on one customer whose order lookup kept timing out: try, apologize, try again, 60 times in one conversation. Two more like it. Three conversations, out of thousands, produced a third of the week's cost. Nothing in James's evaluation had ever asked "how much did this answer cost?", because cost is invisible in a transcript. The fix is two lines of policy: a step budget per conversation (after N tool calls, escalate to a human) and cost per conversation as a tracked number with an alarm, sitting right next to correctness. An agent that cannot run out of patience needs to be given a wallet with a limit.



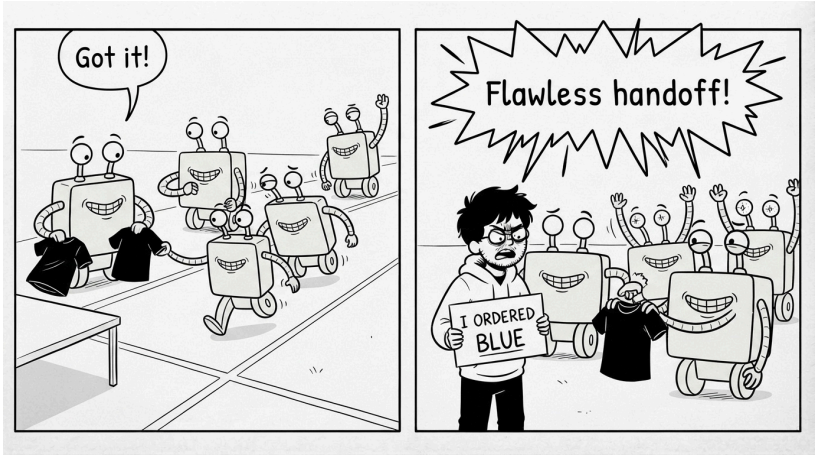
The chain. A new feature: order changes, a five-step task. Check the order, check stock, calculate the price difference, confirm with the customer, apply the change. Each step, measured alone, is fine: roughly 95% each. James's instinct says the feature is 95%

good. The arithmetic says otherwise, and the arithmetic is brutal: 95% five times in a row is 0.95 × 0.95 × 0.95 × 0.95 × 0.95, about 77%. Nearly one conversation in four goes wrong somewhere along the chain, built entirely out of steps that are each "fine." Worse, an early small error compounds: step one misreads which item to change, and steps two through five execute flawlessly on the wrong item, a perfect trajectory serving a mistake. This is why long tasks feel so much less reliable than the demos of each step. Reliability multiplies, and it only ever multiplies downward.

whole task: about 77%

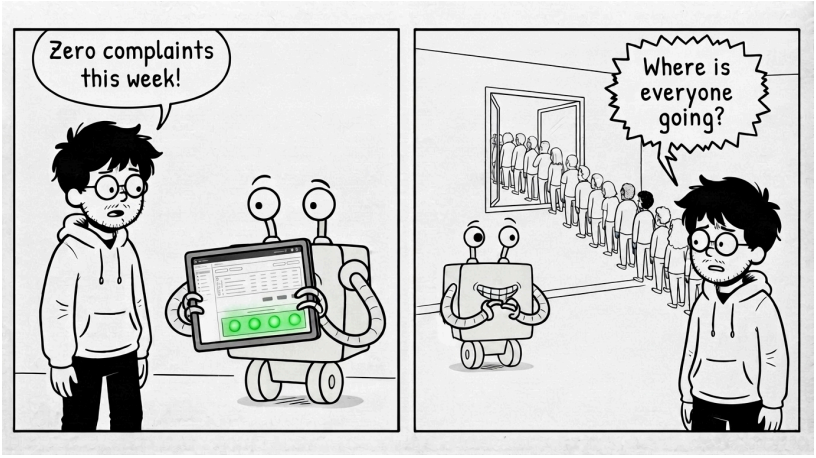
five fine steps. one conversation in four goes wrong.

The fix is not "make every step perfect." It is checkpoints: after the risky early steps, the agent must confirm its understanding against something solid, the order data, or the customer ("Just to confirm: exchanging the blue medium for a blue large, is that right?"), so a wrong turn gets caught at step two, not delivered at step five.



The quiet week. Refund complaints drop. Tickets are calm. The dashboard has never looked better, and by now James has learned that a dashboard that suddenly looks *too* good is a smoke alarm with the battery removed. He samples trajectories and finds that since a prompt tweak two days earlier, the agent has been escalating almost everything to human support. Refusing politely, taking no risks, handing customers to a queue with a four-hour wait. The agent's error rate fell because it stopped doing its job. Customers were not complaining to the agent. They were leaving. This is the silent failure from the preface, all grown up: the worst production failures do not make tickets, because the failure is the absence of something, no refund issued, no answer given, no sale completed. Absence never files a complaint. The protection is to watch the agent's *behavior distribution*, not just its errors: what fraction of conversations end in refund, answer, escalation? Those numbers have a normal

shape, and when one moves sharply in either direction after a change, something happened, even if nothing "failed." Especially if nothing failed.



One named mistake covers all three: **watching only the errors**. Cost hides in successful conversations. Compounding hides in individually green steps. Silent failure hides in an error rate that is *improving*. Production health is not the absence of red. It is every number sitting where it normally sits, and someone noticing when one does not.

Thinking Like a Harness Engineer

Take inventory of what James actually owns now, three months after the worst Monday of his career.

An evaluation set that production grows weekly. Rubrics precise enough that any judge, human or machine, lands on the same score. Seven code judges and a calibrated AI judge. Bars chosen

from real costs, never events wired to alarms. A step budget and a cost meter. Checkpoints inside long tasks. A watch on the behavior distribution. And a habit: every change, no matter how small, gets the twelve-minute question before it ships.

Now notice the strange thing: almost none of this is the agent. The prompt is still about 40 lines. Everything else, all of it, is the machinery *around* the agent. There is a name for machinery like that. When engineers need to trust something powerful and unpredictable, an engine, they never trust the engine's word for it. They build a test harness around it: instruments on every output, load on every input, alarms on every limit. The engine provides the power. The harness provides the knowing.

The agent is the engine. Everything James built is the harness. And the discipline of building it, choosing what to measure, writing the standards, staffing the judges, wiring the alarms, growing the map, has a name too: **harness engineering**. It is the actual job of shipping agents. Teams believe the job is prompting, and the market sells them the engine's brilliance. But the engine was never the hard part. The hard part is that you cannot trust what you cannot measure, and nothing about an agent is measurable until you build the thing that measures it.

Which reframes the question this book opened with. "How do you know your AI agent actually works?" was never a question about the agent. It is a question about you: *did you build the harness?* Without one, "it works" means "it worked while I was watching, on the inputs I tried." With one, it means something an

engineer can sign: "here is the behavior, here is the rate, here is the bar it clears, here is the alarm that fires if that stops being true."

James still ships on Fridays sometimes. He pressed deploy last Friday at 5:52 PM, five minutes later than the Friday that started this book, and went home without checking his phone once all weekend. Not because he is confident. Because the harness is watching, and confidence and knowing are different things. He got the difference tattooed on his workflow instead of his arm.

One honest thing before the last page. Everything in this book stands on one component: those 150 messages, now 164. The set is the ground truth the whole harness rests on, and this book built it the quick way, one afternoon, one frequency count, one pass of labels. It was enough to start. It is not enough to lean on for a year: how large the set really needs to be, how to label at scale without a week of evenings, how to know when your set is lying to you. That is its own craft, and it is the next book in this series: *Building Your First Evaluation Dataset*.

The robot, for the record, has kept its job. It remains cheerful, occasionally wrong, and permanently supervised.

About the author

I have been a developer, a product designer, and a product manager, so I have seen products from every side of the table. I am also self taught, all the way, which is why I teach the way I learned best: a real problem, a real mess, and someone in the middle figuring it out. Poor James.

Sara Mohseni

• • •

dugalaxy fills an empty database with realistic, reproducible test data. Free and open source:
github.com/nugalaxy/dugalaxy

Guides and posts: **nugalaxy.ai/blog**
Say hello: **nugalaxy.ai/contact**

Next in the series: Building Your First Evaluation Dataset