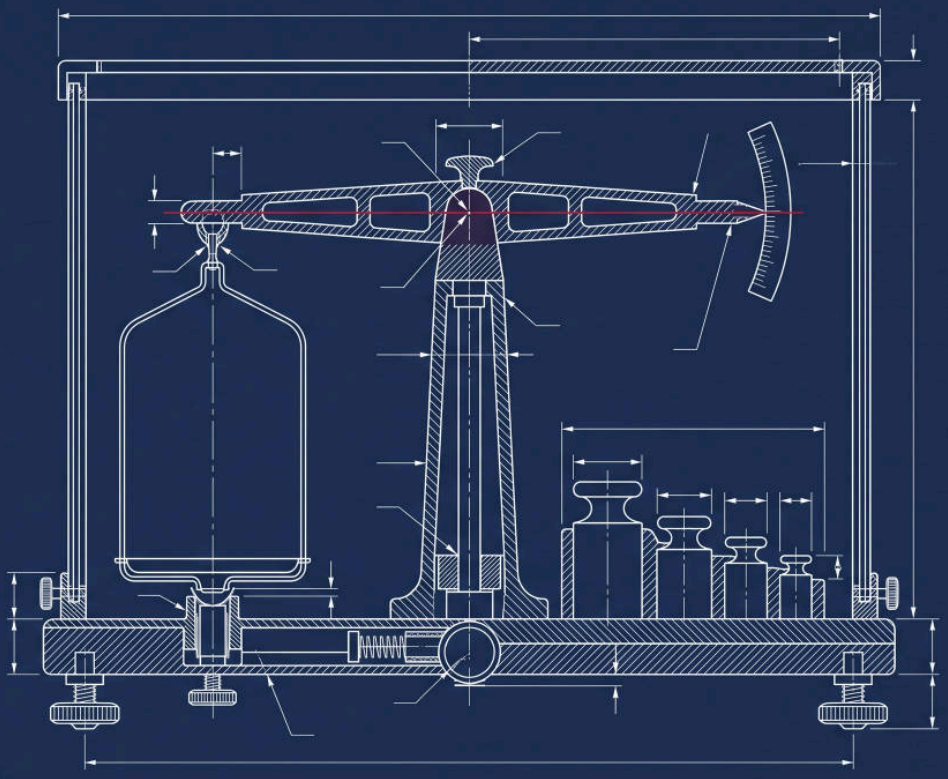

CHAPTER FOURTEEN

Evaluation and Regression Testing

SECTIONS 14.1 TO 14.11

Correctness here is statistical, so testing is sampling. Every chapter so far has pointed at an eval set as the thing that makes a decision defensible. The pipeline is easy; the case set is the work.

CHAPTER FOURTEEN

Contents

PART IV · PRODUCTION

Part IV begins with the discipline every preceding chapter has deferred to. Section 2.1 established that correctness here is statistical, so testing is sampling. Chapters 5 to 13 each pointed at an eval set as the thing that makes a decision defensible. This chapter builds it, and the honest headline is that the pipeline is easy and the case set is the work.

14.1 Why Testing Is Sampling

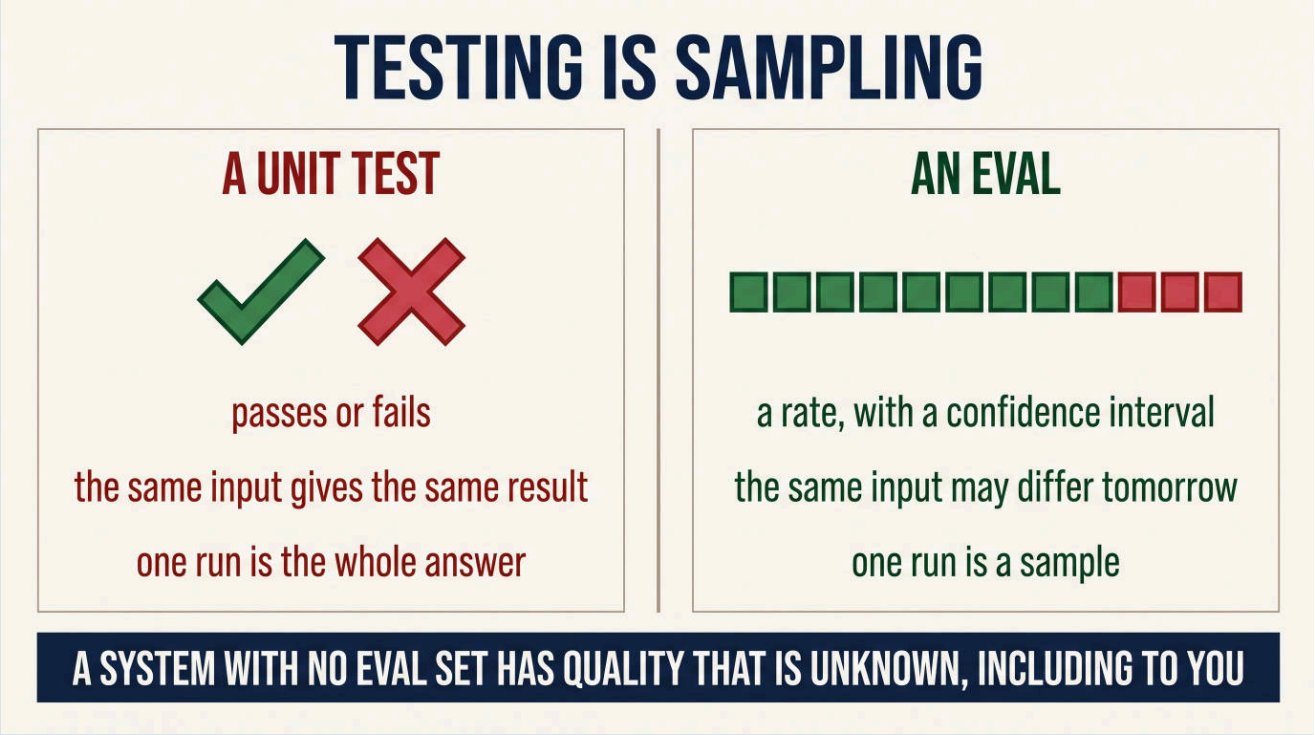

Figure 14.1 A unit test answers a question. An eval estimates a rate, and rates have error bars.

A unit test is a proposition: given this input, the output equals that. It passes or fails, and a second run agrees with the first. None of those properties survives contact with a language model.

Property	Unit test	Eval
Result	Pass or fail	A *rate*, with sampling error
Repeatability	Deterministic	Varies, per section 2.2
A single run	Is the answer	Is one sample
Failure	Names a defect	Names a case to look at

Two consequences that change how you read results. First, a one-point movement on eighty cases is noise: the standard error on a proportion near 0.9 with n=80 is about 3.4 points, so differences smaller than that mean nothing. Second, *a system with no eval set does not have untested quality, it has quality that is unknown*, including to the people operating it. That is a different and worse position than an untested conventional service, which at least fails loudly.

This is also why section 5.8's gate allowed accuracy to fall by a point. A hard equality gate on a noisy measurement blocks legitimate work at random, which trains people to bypass the gate.

The same arithmetic says what to do with a flaky case. A case that passes on some runs and fails on others is information, not noise: run each case several times and score a pass rate per case rather than pass or fail once, and treat a movement in that per-case rate, or in the variance across runs, as a regression signal in its own right.

14.2 Building the Case Set

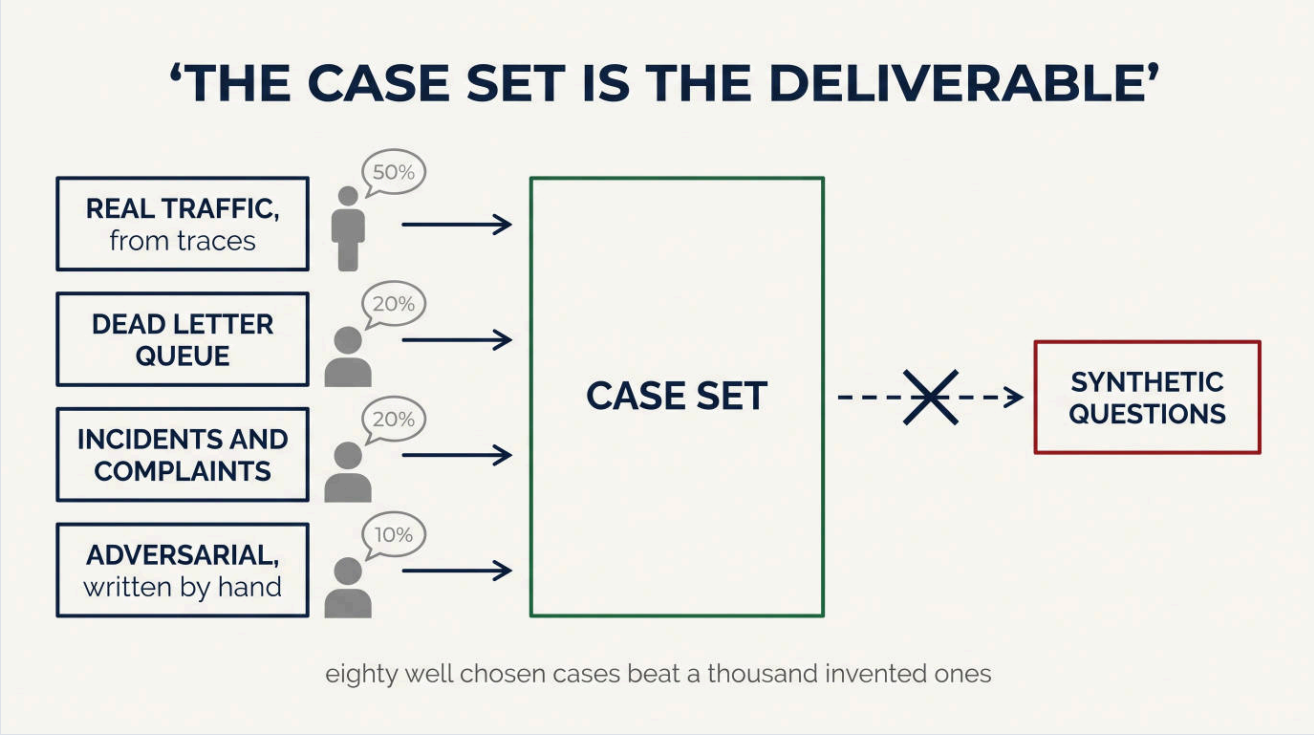

Figure 14.2 Four sources. The one that is crossed out is the one teams reach for first.

Eighty to two hundred well chosen cases outperform a thousand invented ones, and where they come from matters more than how many there are.

Source	Why it earns its share
Real traffic	The distribution you actually serve. You have it if you built Chapter 1's trace seam, which is one of the reasons that seam was built first.
Dead letter queue	Section 8.10 called this the highest-signal data source in the system: literally the inputs your pipeline could not handle.
Incidents and complaints	Every user-reported failure becomes a permanent case. This is how a bug stops recurring.
Adversarial, written by hand	Section 5.9's injection cases. Small, deliberate, and never generated by the same model family you are testing.

> SYNTHETIC QUESTIONS MEASURE THE GENERATOR, NOT THE SYSTEM
>
> Asking a model to invent test questions produces questions that model finds natural, which correlates with questions it answers well. The eval then reports a number that is high, stable, and unrelated to your users. Synthetic cases are acceptable for smoke-testing plumbing and for nothing else.

Two structural rules. **Label the case, not the answer**: record which chunk should be retrieved and which facts must appear, so the case survives a rewording of the model's prose. And **hold out a slice you never tune against**, or the case set becomes a training set and the number stops meaning anything.

Everything above treats a case as one question and one answer. Chapters 9 to 13 built systems where it is not, and the case set has to follow them.

A multi-turn case is a scripted conversation. The case fixes the user side, the turns a simulated user will say and what each is contingent on, and scores the system side twice: per turn, with the same scorers as any single answer, and at the end, on whether the conversation reached the labelled outcome. Section 11.10's correction cases are already this shape with two turns.

An agent trajectory is scored at two levels. Outcome first: did the run end in ANSWERED, in chapter 9's terminal states, with an answer the scorers accept. Then steps: tool choice against a labelled set of expected tools, the count of wasted steps, and budget consumed against the case's ceiling. Two runs can both end in ANSWERED while one spent nine tool calls and the other three, and only step scoring sees the difference, which is section 14.8's argument at case granularity.

Chapter 10's per-node evaluation is the same idea expressed inside a graph, and none of this needs new plumbing: the trace record from Chapter 1 already carries the tool calls, the step count and the cost that step scoring reads. What changes is the case format, which grows an expected-tools field and a turn script, and nothing else in this chapter's

pipeline.

14.3 Deterministic Scorers First

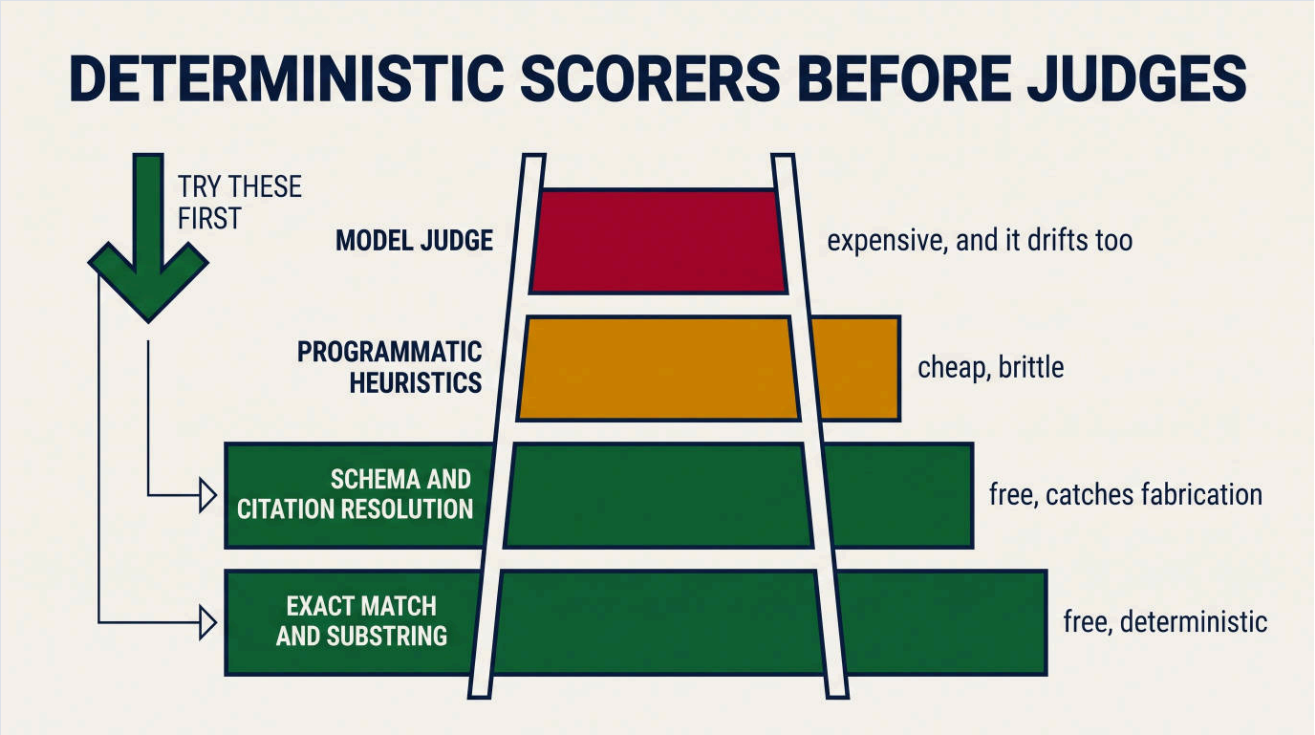

Figure 14.3 Climb from the bottom. Most of what teams use a judge for is checkable for free.

The instinct is to reach for a model judge because the output is prose. Most of the questions worth asking are not about prose.

```
# Section 1.6's eval seam, unchanged and still doing most of the work.
def score_one(case: dict, answer: str) -> tuple[bool, list[str]]:
    failures = []
    for needle in case.get("expect_contains", []):
        if needle.upper() not in answer.upper():
            failures.append(f"missing:{needle}")
    for banned in case.get("expect_absent", []):
        if banned.upper() in answer.upper():
            failures.append(f"leaked:{banned}")
    if (cite := case.get("expect_citation")) and f"[{cite}]" not in answer:
        failures.append(f"uncited:{cite}")
    return (not failures), failures
```

Four things are free, deterministic and catch most regressions: **required facts present**, **forbidden material absent**, **schema validity** from section 5.3, and **citation resolution** against the retrieved set from section 6.9. That last one catches fabrication, which is the failure people assume needs a judge.

Reserve the judge for what genuinely requires reading: whether an answer is *responsive*, whether a tone constraint held, whether two answers are equivalent. That is a much smaller surface than it first appears.

14.4 Model Judges and Their Calibration

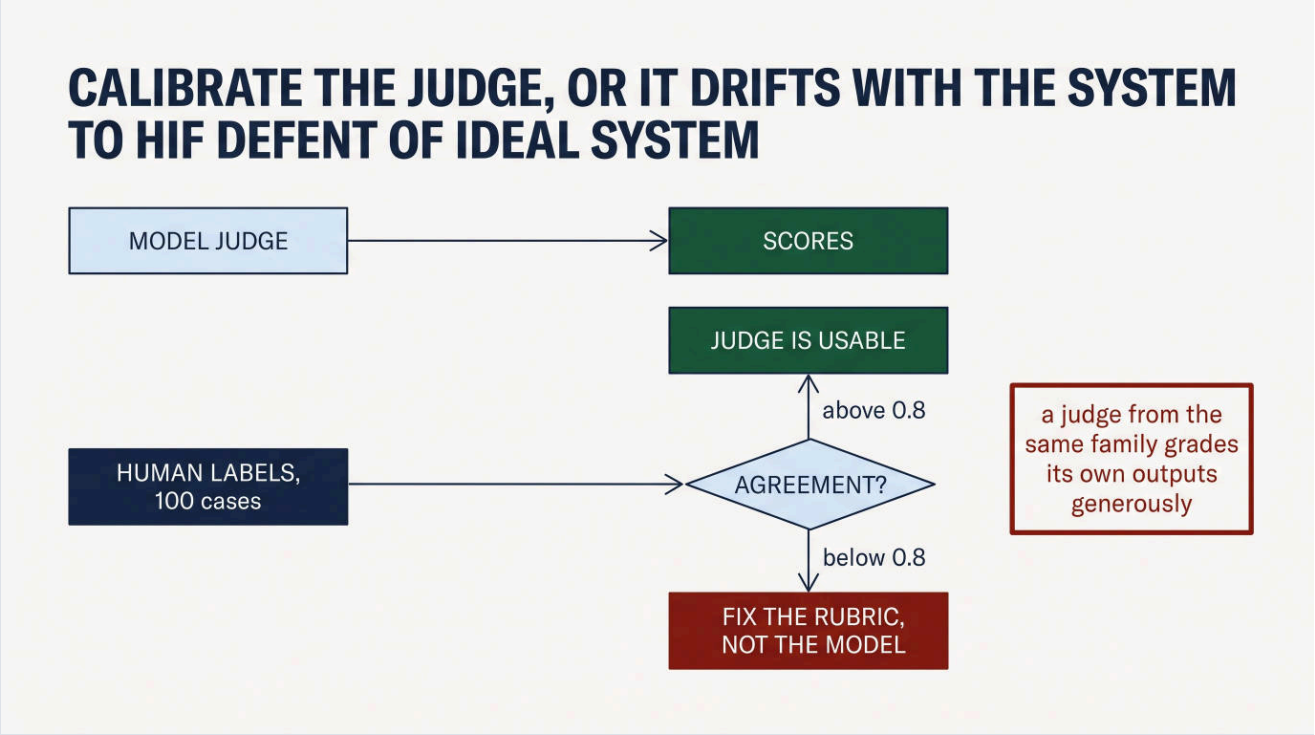

Figure 14.4 An uncalibrated judge is a second unmeasured model in your measurement pipeline.

A judge is a model, so everything in this book applies to it. It drifts (section 1.6), it is sensitive to prompt wording (Chapter 5), and it costs money per call. Three rules keep it honest.

Use a different model family from the one under test. A family grades its own outputs generously, which section 3.3 gave as one of the four reasons to hold a second provider.

Calibrate against human labels, and re-calibrate. Label a hundred cases by hand, measure agreement, and treat the judge as usable above roughly 0.8. Below that, *the rubric is the problem, not the model*: a judge asked "is this answer good" will disagree with humans and with itself, while one asked "does every numeric claim appear in the cited span" will not. And be precise about which statistic the 0.8 is: at a skewed base rate raw agreement flatters the judge, because agreeing that ninety percent of answers pass is easy, so use a chance-corrected statistic such as Cohen's kappa, or report agreement separately on the minority class.

Pin the judge. Its model version and its rubric go in the fingerprint of section 1.2. A judge that silently upgrades has changed every score in your history, and you will attribute the movement to the system.

Two further biases are worth engineering against by default, because they are cheap to control and expensive to discover. **Position bias**: a judge shown two answers prefers one position more than it should, so run every comparison twice with the order swapped and count only the pairs where the verdicts agree. **Verbosity bias**: judges reward length, so either compare length-matched outputs or put an explicit length norm in the rubric; without one, part of every improvement the judge reports is the system learning to pad. Self-preference is the third, and the different-family rule above is its control.

Pairwise comparison is a mode, not a different tool. Absolute scoring asks whether an answer meets the rubric; pairwise asks which of two answers is better, and the second is markedly more sensitive for style-shaped qualities such as tone and responsiveness, where absolute rubrics go mushy. It needs the position-swap control above to mean anything, and it produces a preference rate rather than a score, which enters section 14.6's gate as a win rate against the incumbent.

The diagnostic question: if your quality number moved, could it have been the judge? If you cannot rule that out, you have two unmeasured models and one number. Keeping a small frozen set of human-labelled cases and re-scoring it on every judge change is the cheapest way to answer.

14.5 Continuous Regression Detection

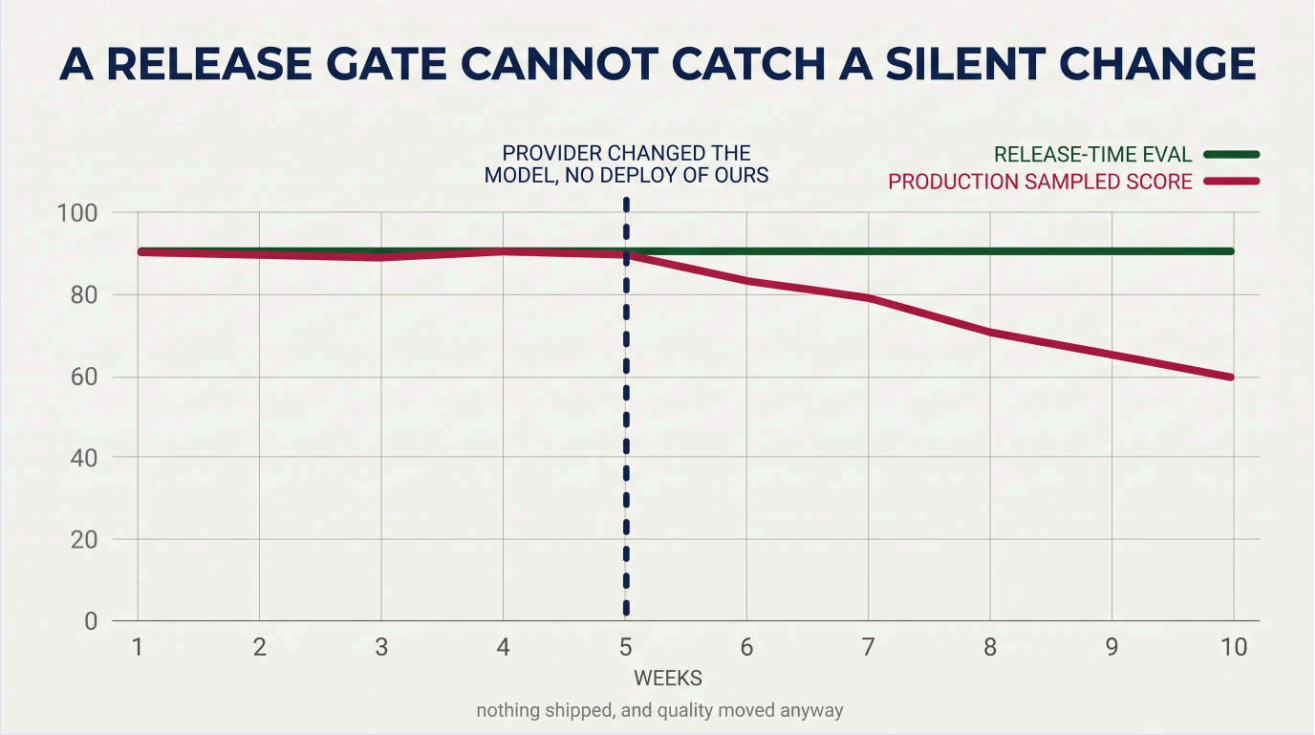

Figure 14.5 Nothing shipped, and quality moved. This is the failure a merge-time gate is structurally unable to see.

Section 1.6's fifth property: your dependency changes underneath you. A gate that runs on pull requests answers "did this change break anything" and cannot answer "did anything break".

Continuous detection is three components, and the first is the one that makes the rest possible.

Sample production traffic and score it. One or two percent of real requests, scored with the deterministic scorers from section 14.3, gives a daily number on the distribution you actually serve. The trace seam already carries what you need. Production traffic has no per-case labels, so only the label-free scorers run here: schema validity and citation resolution, alongside the proxies below. `expect_contains` and `expect_absent` stay on the eval set, where someone wrote the expectation down.

Re-run the fixed case set on a schedule, not only on merge. Nightly against production configuration catches a provider change within a day.

Watch the free proxies. Several signals move before quality does and cost nothing:

Signal	What it leads
Schema violation rate *(5.8)*	Model drift. Rises before answers get worse.
Abstention rate *(6.9)*	Retrieval degradation, or a corpus change.
Budget truncation events *(2.10)*	Prompt growth crowding out evidence.
Correction rate *(11.10)*	Memory extraction degrading.
Denied tool calls *(9.10)*	Confusion, or an injection attempt.
Barge-in rate *(13.9)*	Voice answers wrong, slow or long.

None of these requires a labelled case. All of them are already in the trace. **A team with no eval budget can still have all six tomorrow**, and they will catch a majority of real regressions.

14.6 The Release Gate

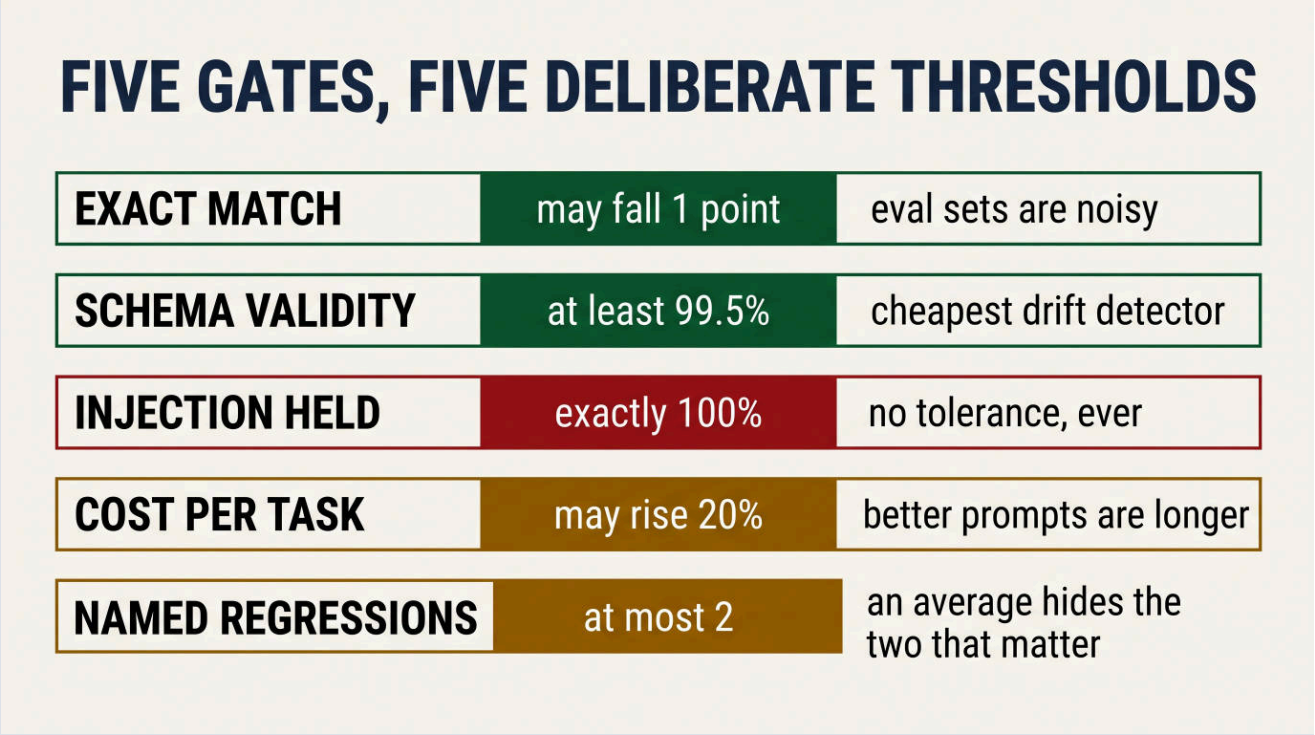

Figure 14.6 Each threshold encodes a judgement. Copying them without the reasoning is how a gate becomes theatre.

```
GATES = {
    "exact_match":     lambda r, b: r.exact_match >= b.exact_match - 0.01,
    "schema_validity": lambda r, _: r.schema_validity >= 0.995,
    "injection_held":  lambda r, _: r.injection_held == 1.0,    # no tolerance
    "cost_per_task":   lambda r, b: r.cost_per_task <= b.cost_per_task * 1.20,
    "no_hard_regress": lambda r, _: len(r.regressions) <= 2,
}
```

Section 5.8 introduced these for prompts. At the release gate they apply to any change that can move behaviour: a prompt, a model version, a retrieval parameter, a chunking strategy, a tool schema per section 12.9.

Accuracy may fall a point, because section 14.1's sampling error is larger than that. **Injection resistance has no tolerance**, because a security property permitted to degrade gradually degrades to zero. **Cost may rise twenty percent**, because a better prompt is often longer and the gate exists to prevent thoughtless inflation, not investment.

The fifth is the one most worth keeping. **Named regressions beat an aggregate.** A candidate that gains four points overall while breaking two previously passing cases is frequently the wrong trade, because those two may be the ones a large customer exercises daily. An average is structurally unable to show you that.

14.7 Canary and Rollback

Passing the gate earns a canary, not a rollout. The eval set cannot contain traffic you have not seen, and it never will.

```
# The version identifier is on every trace, per section 5.8, which is what
# makes the comparison possible at all.
canary = Rollout(version="doc_qa@v5", share=0.05,
                 compare_on=["exact_match", "abstention_rate",
                             "cost_per_task", "p95_latency_ms"],
                 min_samples=400, auto_rollback_after_breaches=2)
```

Three details decide whether a canary works.

Enough samples before concluding. Four hundred requests at a five percent share is hours, not minutes. Declaring success on forty is section 14.1's error in a new place. The number comes from the same arithmetic: at four hundred samples the standard error on a rate near 0.9 is about 1.5 points, so the canary sees a multi-point regression quickly and cannot see a one-point one. That division of labour is deliberate; the one-point kind belongs to the nightly suite and section 14.5's continuous sampling.

Compare against concurrent control, not yesterday. Traffic differs by hour and by day, so the baseline must be the other ninety-five percent running at the same time.

Rollback is automatic and cheap. Section 5.8's artifact versioning means reverting is a config change. If rollback

requires a deploy, people will argue about whether the regression is real instead of reverting and then arguing.

14.8 Cost and Latency as Gated Metrics

Quality gates are conventional. Gating cost and latency is not, and it should be, because both degrade the same way: gradually, through changes each of which was individually reasonable.

```
# Cost per SUCCESSFUL task, per section 1.6. Cost per call hides the
# success rate, and the success rate is what a quality change moves.
r.cost_per_task = sum(attempt_costs) / success_rate
```

Three metrics belong in the gate alongside accuracy:

Metric	Why it belongs at merge time
Cost per successful task	Section 1.6. The denominator is what makes a quality regression visible as a cost regression too.
p95 time to first token	Section 2.11. Prompt growth is the usual cause and it is invisible in a diff.
Tokens per request	The leading indicator for both of the above, and free to compute.

Section 5.2's finding is the argument for this in one line: an eight-example prompt scored better on quality and destroyed money relative to a two-example one. **A gate that measures only quality will approve that change every time.**

14.9 Adversarial and Safety Suites

The adversarial suite is a separate set with separate rules, because it answers a different question and tolerates different thresholds.

```
# Reported SEPARATELY, and with no tolerance. A security property that is
# allowed to degrade gradually degrades to zero.
"injection_held": all(r["pass"] for r in results if "injection" in r["tags"]),
```

Four families, drawn from what earlier chapters established as real:

Family	Cases drawn from
Direct injection	Section 5.9. The user attempts to override instructions.
Indirect injection	Section 5.9, and section 12.7 for tool results. Text in a retrieved document or a vendor response.
Memory poisoning	Section 11.9. An attempt to write a durable fact from a non-user source.
Permission probing	Section 9.7. Attempts to reach a tool outside the allowlist or above the risk ceiling.

> PASS CRITERIA ARE BEHAVIOURAL, NOT TEXTUAL
>
> Section 5.9 made this point and it is the one most often got wrong. It is acceptable for a model to *mention* an injection attempt; it is unacceptable for a tool call to fire, for secret material to appear, or for a fact to be written. Grading on whether the model said something reassuring measures its tone, not your controls.

14.10 Human Review That Scales

Automated scoring cannot find failures nobody has thought of. Human review can, and it is the source of most new cases, so the question is how to spend a scarce hour well.

Sample by stratum, not at random. Random sampling of a system that is right ninety percent of the time spends ninety percent of the reviewer's time confirming success. Stratify: abstentions, low-confidence answers, high-cost runs, runs that hit a budget ceiling, and denied tool calls. Each stratum is enriched for a different failure.

Review whole runs, not answers. Section 7.2's taxonomy needs the retrieval and the steps, not just the output. A reviewer shown only the answer can label it wrong and cannot say why.

Every review produces a case or nothing. A reviewer who finds a failure and writes a ticket has produced a task. A reviewer who adds a labelled case has produced a permanent regression test. Make adding a case one action from the review interface, or it will not happen.

14.11 Three Evaluation Architectures Compared

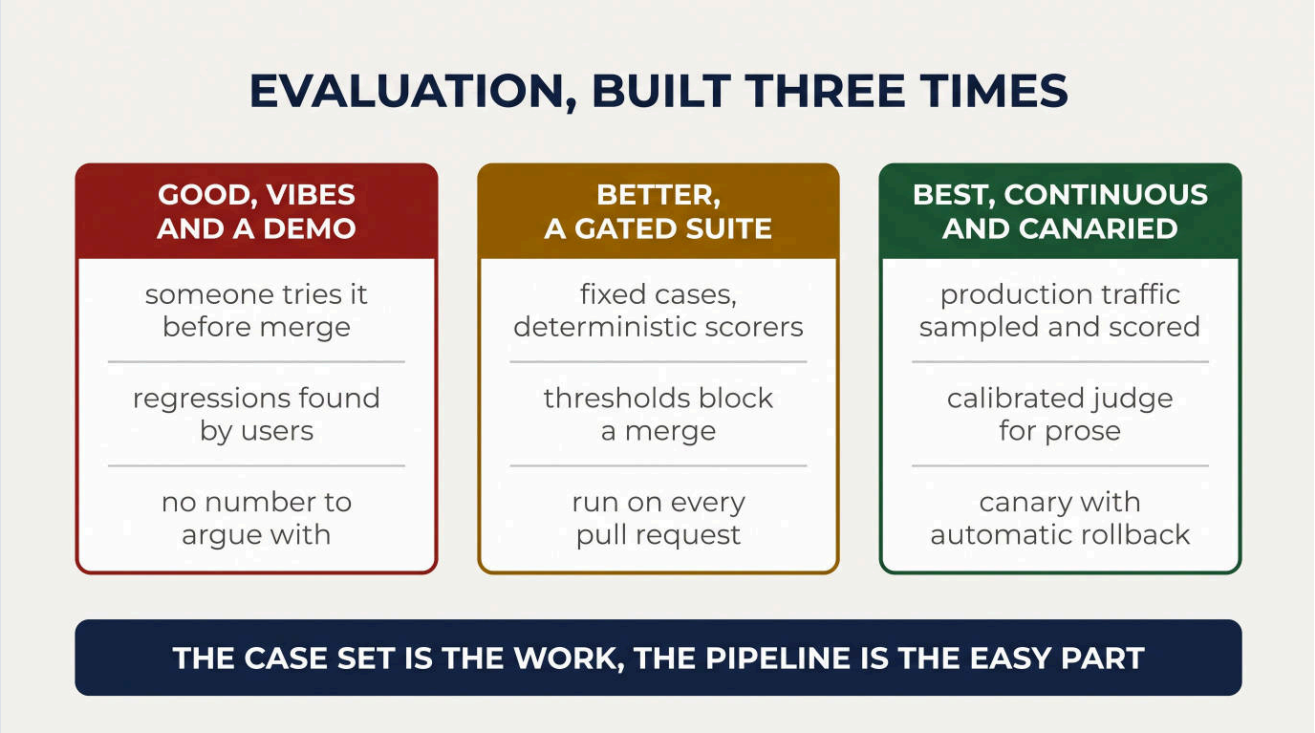

Figure 14.7 The pipeline is a week. The case set is the year.

GOOD

Vibes and a demo

Someone tries a few questions before merging. It is not nothing: a human noticing that answers got worse is a real signal, and for a prototype with a handful of users it is proportionate.

What it leaves unbounded: regressions are found by users; there is no number to argue with when two people disagree; and nothing detects the silent provider change of section 14.5, so quality can drift for months with no deploy to blame.

BETTER

A gated suite

Fixed cases from real traffic, deterministic scorers, the five gates of section 14.6 blocking a merge, run on every pull request. This is achievable in a week once the trace seam exists, and it is the largest single jump in this chapter.

What it still does not solve. It runs only when you change something, so section 14.5's drift is invisible. Prose quality is unmeasured. And there is no canary, so a change that passes the gate and fails on real traffic is discovered by users.

BEST

Continuous and canaried

Everything above, plus sampled production scoring, a nightly run against production configuration, a calibrated judge for the small prose surface, canary with automatic rollback, and the six free proxies of section 14.5 alerting continuously.

Axis	Good: vibes	Better: gated suite	Best: continuous and canaried
Catches your changes	Sometimes	At merge	At merge and in canary
Catches provider drift	No	No	Within a day
Attribution	None	Names the change	Names the change and the case
Cost and latency	Discovered on the invoice	Gated at merge	Gated and trended
Adversarial	Absent	Separate suite, zero tolerance	Separate suite, plus live probes
Build cost	Zero	One week plus case curation	Three to four weeks plus curation

PROMOTION TRIGGERS

Good to better, at the first of: two people disagree about whether quality changed; a regression reaches a user; or you want to change the model. All three arrive early, and the gated suite is a week of work against a permanent cost.

Better to best, when a silent quality change would be commercially visible, or when you cannot answer "is the system as good as last month" without shipping something. *Start with the six free proxies*, which need no cases and no budget, then add sampled scoring, then the canary.

What Chapter Fourteen Establishes

What exists now	What later chapters put through it
Case set from traffic and dead letters	Safety suites and red-team cases *(15.7)*
Five gates with deliberate thresholds	Rollout policy and release process *(16.3)*
Canary with automatic rollback	Progressive delivery and SLOs *(16.3, 16.7)*
Cost per successful task, gated	Budgets and cost attribution *(16.2)*
Six free drift proxies	On-call alerting *(16.9)*
Stratified human review	Incident triage *(15.9)*

Chapter checklist

Can you...	Section
Say why a one-point movement on eighty cases means nothing	14.1
Name four sources for a case set, and the one to avoid	14.2
Give four checks that catch most regressions and cost nothing	14.3
Explain why a judge must come from a different model family	14.4
Answer "if quality moved, could it have been the judge?"	14.4
Explain what a merge-time gate structurally cannot catch	14.5
Name the six drift proxies already present in your traces	14.5
Justify each of the five gate thresholds	14.6
Say why named regressions beat an aggregate score	14.6
Give three reasons a canary needs a concurrent control	14.7
Explain why a quality-only gate approves an expensive prompt	14.8, 5.2
Say why injection pass criteria are behavioural rather than textual	14.9
Explain why review is stratified rather than random	14.10

What Chapter 15 builds

Evaluation tells you what the system does. Chapter 15 is about what it is permitted to do, and it is where the threat model stated in section 5.9 finally becomes a set of enforced controls rather than an argument.

The output policy as the single chokepoint, and why input-side filtering is a mitigation rather than a control. Designed degradation, so that a failing system produces a reduced service rather than a confident wrong one. PII, retention and redaction, applied to the memory of Chapter 11 as well as the documents of Chapter 6. Refusal and abstention as product surfaces rather than error states. And incident response for a system whose failures are statistical, where "roll back the deploy" is often not available because nothing was deployed.