LLM EVALUATION AT SCALE

Testing, Regression Detection, and Quality Measurement for Generative AI Systems

AUTHORED BY

ADITI PATODIYA

LLM Evaluation at Scale

Testing, Regression Detection, and Quality Measurement
for Generative AI Systems

Aditi Patodiya

LLM Evaluation at Scale

Testing, Regression Detection, and Quality Measurement for Generative AI Systems

First Edition, 2026

Disclaimer

The information in this book is provided for educational and informational purposes only. It does not constitute professional engineering, legal, clinical, safety, or regulatory advice. The author and publisher make no representations or warranties regarding accuracy, completeness, or applicability. Readers should consult qualified professionals before making decisions based on this content. Views expressed are those of the author and do not necessarily reflect the positions of any organizations mentioned.

Printed in the United States of America.

Contents

Chapter 3: The Full LLM Application Evaluation Stack

- Evaluating input understanding
- Evaluating retrieval quality
- Evaluating context assembly
- Evaluating prompt construction
- Evaluating model output
- Evaluating tool calls and actions
- Evaluating post-processing
- Evaluating streaming and partial responses
- Evaluating persistence and conversation state
- What the map is for

Chapter 4: Building the First Evaluation Dataset

- Why evaluation datasets fail
- Sources of evaluation examples
- Real user queries vs. synthetic queries
- Happy paths, edge cases, and adversarial cases
- Dataset size: starting small but meaningful
- Metadata every test case should include
- Versioning evaluation datasets
- Avoiding overfitting to test cases

Chapter 5: Creating Golden Answers Without Overfitting

- What a golden answer is
- When golden answers work
- When golden answers fail
- Reference answer vs. expected behavior
- Acceptable answer variation
- Rubric-based comparison
- Multiple valid outputs
- Maintaining golden answers over time

Chapter 6: Mining Production Traffic for Evaluation

- Sampling production conversations

- Claim-level verification
- Unsupported inference detection
- Partial hallucinations
- Measuring hallucination severity

Chapter 10: Evaluating Multi-Turn Conversations

- Why single-turn tests are not enough
- Context carryover
- Reference resolution
- User correction handling
- Conversation state updates
- Memory freshness
- Summary quality
- Multi-turn regression tests
- Conversation-level scoring

Chapter 11: Evaluating Tool Use and Agentic Actions

- Tool selection accuracy
- Argument correctness
- Permission and authorization checks
- Confirmation before irreversible actions
- Idempotency and duplicate prevention
- Tool failure handling
- Retry behavior
- Auditability
- Human-in-the-loop evaluation

Chapter 12: LLM-as-Judge Evaluation

- What LLM-as-judge is
- When it works well
- When it fails
- Pairwise comparison vs. absolute scoring
- Judge prompt design
- Bias and position effects

Chapter 19: Human Feedback Loops

- Human review vs. user feedback
- Expert review vs. crowd review
- Reviewer training
- Calibration exercises
- Measuring inter-rater agreement
- Disagreement resolution
- Feedback quality control
- Converting feedback into product changes
- Avoiding reviewer fatigue

Chapter 20: Online Evaluation and Experimentation

- Offline vs. online evaluation
- A/B testing for LLM products
- Canary launches
- Guardrail monitoring
- User satisfaction metrics
- Task completion metrics
- Business metrics vs. quality metrics
- Interpreting noisy user feedback
- When to rollback

Chapter 21: Building an Evaluation Platform

- Why spreadsheets do not scale
- Dataset versioning
- Evaluation execution engine
- Scoring service
- Comparison reports
- Human review workflow
- Integration with CI/CD
- Governance and access control

Chapter 22: Cost, Latency, and Quality Tradeoffs

- Quality is not the only metric

Introduction

The dashboards were green. All of them. Latency was inside budget, error rates were flat, the deployment pipeline had gone through without a single failed check, and the on-call engineer had already gone to bed. It was a little after two in the morning, and by every measure the engineering organization had agreed to care about, the release was a success.

Four days later, a product manager forwarded a screenshot. A customer had asked the assistant to compare three products. The assistant had listed them cleanly. The customer had then typed a short follow-up, six words, asking about the second one. The assistant answered about the first. The customer corrected it. The assistant apologized and answered about the third. The screenshot ended there, because that is where the customer had ended the conversation and, most likely, ended their patience with the product.

Nothing in our monitoring had fired. No exception was thrown. No timeout occurred. The service returned a well-formed response in under a second, with a correct HTTP status code and a payload that passed schema validation. Every test we had written was still passing. The system was, in the precise engineering sense of the word, working.

That screenshot has stayed with me longer than any outage I have been paged for. Outages are honest. They tell you they happened. This kind of failure does not announce itself. It sits quietly inside a healthy-looking system, degrading the product one conversation at a time, and it will keep doing so for as long as your instruments have nothing to say about it.

What Traditional Testing Was Built For

Software testing as most of us practice it descends from a specific set of assumptions. The system under test is deterministic. The same input produces the same output. Correctness is a property you can state in advance, encode as an assertion, and check mechanically. When the assertion fails, the failure is unambiguous, reproducible, and attributable to a change you can find in version control.

Those assumptions have held up remarkably well. They gave us unit tests, integration tests, continuous integration, and the whole apparatus of modern release engineering that Jez Humble and David Farley described in Continuous Delivery. They gave us the confidence to deploy to production

dozens of times a day. They are the reason a mid-sized engineering team can ship software that runs on a hundred million devices without anyone losing sleep about it most nights.

Generative systems break every one of those assumptions at once.

The same input does not produce the same output. Set the temperature to zero and you narrow the variance, but you do not eliminate it, because the system around the model is also moving. The retrieval index was refreshed. A tokenizer boundary shifted. A provider silently updated the serving stack. Two requests that look identical to you can traverse different context, different evidence, and different truncation boundaries, and the model will faithfully produce two different answers from two different inputs that you believed were the same.

Correctness is often not a property you can state in advance. If a user asks whether a product is a good fit for a small apartment, there is no assertion you can write that captures the right answer. There are many right answers. There are also many wrong ones, and the difference between them is a matter of judgment that a reasonable person could reasonably dispute.

Failures are frequently not reproducible. The conversation that broke was six turns deep, drew on a document that has since been reindexed, and depended on a context window that filled up in a way it will not fill up again. You have the final answer and a vague complaint. You do not have the failure.

And the failure is often not attributable to a change in version control at all, because nobody changed anything. The model provider changed something. Or the data changed. Or the users changed, which happens constantly and without warning, because a marketing campaign brought in a cohort that asks questions your evaluation set has never seen.

The Shape of an LLM Failure

Once you start looking at generative failures closely, a pattern shows up. They are layered, and the layer where the damage is visible is almost never the layer where the damage was done.

A user gets a confidently wrong answer about a return policy. The obvious diagnosis is hallucination, and if hallucination is your diagnosis then the model is your defendant. But trace it back. The retrieval layer returned three

documents, two of which were about a different product line. The context builder truncated the conversation at a fixed token limit, cutting the turn in which the user specified which product they owned. The prompt template ordered the evidence by recency rather than relevance, so the most relevant passage sat in the middle of a long context, which is exactly the position from which language models retrieve least reliably, a result Nelson Liu and colleagues at Stanford documented carefully in the paper they titled Lost in the Middle.

By the time the model produced its answer, the answer it produced was arguably the best available answer to the question it was actually asked. The model did not hallucinate so much as it was set up to fail. If your evaluation only looks at the final text, you will keep filing bugs against the model and keep shipping fixes that do not work.

This is why I have come to think of evaluation as a system property rather than a model property. The model is one component. It happens to be the loudest one, and the most fun to talk about, and the one that gets the press coverage. It is also the component you have the least control over. The parts you control are the parts around it, and those parts are where most of your quality lives.

There is one more property of these systems that makes the whole problem harder, and it is the one that catches experienced engineers off guard. The system changes when you have changed nothing.

A conventional service is stable between deployments. If nothing shipped, nothing moved. A generative application has at least four moving parts you do not control. The model provider updates a serving stack. The retrieval index is rebuilt from a corpus that other teams edit. The traffic shifts because a campaign brought in a new population of users. And the world changes, which means the answers that were correct last month are wrong this month while your evidence still says otherwise.

That means you have two questions to answer rather than one. Whether your change made things worse, and whether things got worse while you were not looking. No release process is designed to ask the second one, and only continuous measurement can answer it.

Why Demos Are the Enemy

I want to say something uncomfortable about demos, because demos are how most generative AI features get approved and they are the single most misleading artifact in the entire discipline.

A demo is a curated sample of size one, run by the person with the strongest possible incentive for it to succeed, on inputs they chose, in a session they can restart. It is an existence proof. It shows that the system can produce a good answer. It says nothing whatsoever about how often it does, on which inputs it fails, how badly it fails when it does, or whether the failures cluster in a way that will make a specific group of users conclude the product is broken.

The gap between the demo and the deployment is where careers go to die. I have watched teams present a flawless five-minute walkthrough to leadership, get funding, launch, and then spend the next two quarters discovering that the product works beautifully for the twelve queries they happened to try and unevenly for everything else. Nobody lied. The demo was real. It just measured the wrong thing, which is to say it measured almost nothing.

The antidote is a measurement discipline that produces a number you can argue with, defend, and watch move. More demos will not get you there.

Evaluation Is an Engineering Problem

There is a tendency, especially in organizations where the research function and the engineering function sit far apart, to treat evaluation as a science activity. Somebody runs a benchmark. A slide appears with a score on it. The score is higher than the last score. The feature ships.

I understand why this happens, and I want to argue against it directly. Benchmarks measure general capability. Products need specific capability. A model that tops a public leaderboard may still be worse for your application than the model you already run, because your application asks a narrow set of questions in a particular voice against a particular corpus with particular constraints on latency and cost, and none of that was on the leaderboard.

Stanford's Center for Research on Foundation Models made this point structurally when they published HELM, the multi-scenario benchmark work led by Percy Liang and his collaborators. Their argument was that a single accuracy number hides more than it reveals, and that any honest

evaluation has to report multiple metrics across multiple scenarios simultaneously, including the ones where the model does badly. That is the right instinct. The version of it that applies to a production team is narrower and more demanding: report the metrics that describe your product, on the traffic your product actually receives, and report them every time anything changes.

Treating evaluation as engineering means it gets the things engineering work gets. It gets a codebase. It gets version control. It gets code review. It gets a service level objective. It gets an owner whose name is on a page somewhere. It gets a budget. And it gets wired into the release pipeline so that a change which degrades quality is blocked automatically, the same way a change which fails a unit test is blocked automatically, without anybody having to notice, argue, or remember.

The EVAL-OPS Framework

This book is organized around a framework I have found useful for thinking about the whole problem at once. I call it EVAL-OPS. It has seven parts, and each letter names a body of work that a serious production team eventually has to do.

E is Evaluation Dataset Design. Before you can measure anything you need something to measure against, and the something has to look like your product rather than like a research benchmark. Representative cases drawn from real use, synthetic cases built to cover the scenarios real use has not produced yet, edge cases, adversarial cases, and a growing library of the specific failures you have already seen. Versioned, so you can tell what changed. This is Part Two.

V is Validation Rubrics. Measurement requires criteria, and criteria for generative output have to be written down with enough precision that two different reviewers, or a reviewer and a machine, will reach the same verdict on the same output most of the time. Correctness, groundedness, completeness, safety, tone, latency, usefulness. Each needs a definition sharp enough to score. This is Part Three.

A is Automated and Human Review Loops. Some checks are deterministic and cheap. Some require judgment. Some require expertise you cannot automate at any price. The work is deciding which is which, then building the loop that combines them without letting the cheap signal quietly replace

the expensive one. Part Three covers this too, including the use of models as judges, which is powerful and dangerous in roughly equal measure.

L is Live Regression Monitoring. Quality degrades. It degrades when you change the prompt, when you change the model, when you change the retrieval index, when you change nothing at all and the world changes around you. Detection has to be continuous, and it has to be sensitive enough to catch a three percent drop before your users do. This is Part Four.

O is Observability and Failure Taxonomy. When quality drops you need to find out why, and the answer lives somewhere in a pipeline with a dozen stages. That means logging the intermediate state, tracing a request end to end, and having a shared vocabulary for failure types so that two engineers looking at the same incident describe it the same way. This is Part Five.

P is Production Release Gates. Everything above is decoration if a bad change can still ship. Evaluation has to be a gate, with a threshold, wired into continuous integration and rollout, with an automatic rollback behind it. This is Part Four as well, because gating and regression detection are two halves of the same mechanism.

S is Scale, Governance, and Continuous Improvement. One team can run evaluation with discipline and a spreadsheet. Ten teams cannot. At organizational scale you need a platform, an operating model, an owner, a review process, and a way of reporting quality to people who will never read a rubric. This is Part Six, and Part Seven turns the whole thing into templates you can copy.

What This Book Assumes

I have written this for engineers and engineering leaders who are shipping generative features into production systems that real people use. I assume you know what an API is, what a deployment pipeline does, and roughly how a language model gets from a prompt to a token. I do not assume you have trained one, and almost nothing in this book requires you to.

I assume your constraints are real. You have a latency budget. You have a cost ceiling. You have a legal team, a security review, and a launch date. You cannot solve quality problems by throwing an arbitrarily large model at them, and you cannot solve them by hiring a hundred annotators either. The techniques here are chosen because they work under constraint.

I also assume good faith and limited time, which is why every chapter ends with something concrete. A schema. A rubric. A checklist. A scorecard. You should be able to read a chapter on a flight and have something you can put in a repository when you land.

What This Book Will Not Do

It will not give you a universal metric. There is no single number that captures the quality of a generative system, and anyone selling you one is selling you a proxy that will be gamed within a quarter. Charles Goodhart's observation about economic indicators applies with unusual force here: a measure that becomes a target stops being a good measure. You will need several numbers, and you will need to keep an eye on the ones you are not optimizing.

It will not tell you which model to use. That answer expires. What will not expire is the method you use to decide, and Chapter Fourteen gives you that.

It will not pretend that evaluation is solved. Large parts of it are genuinely open. Measuring the quality of a long, multi-turn, tool-using conversation is a problem that the field has not settled, and I have tried to be honest throughout about where the ground is firm and where it is still moving. Where I am giving you a technique that works but has known failure modes, I say so.

How to Read This Book

Read Part One in order. It is short and it sets up the vocabulary that everything else uses.

After that, the parts are close to independent, and you should go where your pain is. If you have no evaluation dataset, go to Part Two and build one this week. If you have a dataset and no idea how to score it, go to Part Three. If your problem is that a prompt change broke production last month and nobody caught it, go to Part Four and start with release gates. If your problem is that quality drops and you cannot find out why, go to Part Five. If your problem is that six teams are each doing evaluation differently and none of the numbers can be compared, go to Part Six, and then go and have a difficult conversation with your leadership.

Part Seven is a reference. It contains an end-to-end architecture, three worked case studies in different domains, and a set of templates. If you like to see the finished thing before you understand how it was built, start there and work backward.

One Last Thing About That Screenshot

I keep coming back to the six-word follow-up, the one that said tell me more about the second one, because of how small it is. It is the least remarkable thing a person can say in a conversation. It is the kind of sentence a system either handles or does not, and there is no partial credit for it.

Every test we had passed. Every dashboard was green. And the product was quietly failing in the only place that matters, which is inside the head of a person who wanted help and did not get it, and who will not file a bug report, and who will simply stop coming back.

The whole point of an evaluation discipline is to make that failure visible before the user finds it. Not to prevent every failure, which is impossible. To find them, name them, count them, rank them, and put a number on the wall that goes down when you fix them and up when you break them. That is the job. Everything in the rest of this book is a technique for doing it.

I want to set an expectation about what that job feels like, because the first honest measurement is a shock and I would rather you were braced for it.

Your number will be worse than you think. It will be worse than the demo suggested, worse than your team believes, and worse than whatever you have most recently told a leader. When we first scored a real dataset properly, against criteria we had written down before we looked at any outputs, the pass rate came back roughly twenty points below what everyone in the room would have guessed, and the room included people who had built the thing and used it every day.

Nothing had gotten worse. We had simply, for the first time, looked.

That number is the most valuable artifact you will produce all year, and within about a week there will be pressure to soften it. Somebody will point out that several of the failing cases are unfair. Somebody will suggest that the rubric is too strict. Somebody will propose blending in an easier set of cases to make the metric more representative, and they will mean it sincerely, and they will occasionally even be right. Whether you hold at that moment determines everything that follows, because a measurement that bends under pressure is a measurement that will bend every time, and a metric that only ever moves in the direction people want has stopped being a metric. It has become a mirror.

Let us start by being precise about why the tools we already have are not enough.

Part One

Why LLM Evaluation Is Different

Chapter 1: The Testing Gap in Generative AI

The pull request was eleven lines long. Ten of them were whitespace. The eleventh changed a single sentence in a prompt template, moving an instruction about citation format from the end of the system message to the beginning, where somebody had read that models pay more attention.

It had two approvals. It had passed continuous integration, which took ninety seconds and ran four hundred and twelve tests, every one of which was green. The author merged it on a Thursday afternoon and went to a meeting.

What that change did, and what nobody discovered for nineteen days, was reduce the rate at which the assistant included a citation at all. Not the format of the citations. The presence of them. Moving the instruction to the front of the system message had put it further from the evidence block, and the model, faced with a long context and an instruction it had seen a thousand tokens earlier, started dropping the citation about one time in seven. Every test passed because no test in that suite of four hundred and twelve asserted anything about whether an answer cited its source. They asserted that the response parsed, that the latency was under budget, that the API returned two hundred, and that a handful of known queries produced output containing certain keywords. All of that remained true. The product got measurably worse anyway.

I have told this story to a lot of engineers and the reaction is almost always the same. First, a defensive one: we would have caught that. Then, about ten seconds later, a quieter one: how would we have caught that.

The difference between software correctness and AI usefulness

Traditional software has a property that we rely on so completely that we forget it is a property. Its correctness is decidable. You can look at a function, look at its specification, and determine whether the function meets the specification. If it does not, there is a defect, and the defect has a location.

This decidability is what makes the unit test possible. A unit test is an executable statement of intent. It says: given this input, the output must equal this value. It is a proposition that is either true or false, and a machine can check it in microseconds. Stack a few thousand of those together and you have a safety net that catches almost everything a careless human might do to a codebase.

Now try to write one for a language model. Given the input describe the return policy for this item, the output must equal what, exactly. There is no string you can put on the right-hand side of that assertion. Any string you choose is one of a very large number of acceptable answers, and the model will produce a different one on Tuesday than it did on Monday, and both will be fine.

You can weaken the assertion. You can check that the output contains the word thirty, because the return window is thirty days. This works right up until the model writes the number as a numeral, or says a month, or says within thirty days of delivery in a sentence that also says unless the item was marked final sale, which is correct and which your keyword check happily accepts even if the model had gotten the whole thing backward.

The deeper issue concerns the property you actually care about. Software correctness is the wrong frame. What you need to know is whether the answer was useful, which asks something

else entirely: did a person who wanted help get help. Those are different properties, they are measured in different ways, and one of them has no boolean answer.

A response can be entirely factually correct and completely useless. It can answer the question that was asked while missing the question that was meant. It can be accurate, well-sourced, appropriately hedged, and so long that nobody reads past the second paragraph. Traditional testing has nothing to say about any of that, because traditional testing was built for a world where the specification was the whole truth.

Why 'it looks good in a demo' is not an evaluation strategy

The demo problem is a sampling problem, and it helps to state it in those terms because then you can see how bad it is.

Suppose your assistant handles a query correctly eighty-five percent of the time. That is a rate most teams would consider respectable for a first launch. Now suppose you run a demo with eight carefully chosen queries. The probability that all eight succeed is roughly twenty-seven percent, which sounds low until you remember that the person running the demo tried the queries beforehand, discarded the ones that failed, and is now showing you the survivors. Conditioned on that selection, the probability of a clean demo approaches one. You will see a perfect demo of a system that is wrong one time in seven.

Push the failure rate to a genuinely alarming thirty percent and the demo still looks fine, because the demo was never a sample. It was a search. Somebody searched the input space for inputs that work and then showed you those. Call this dishonesty and you have missed the point. It is the completely natural behavior of a human being who wants their project funded, and it happens even when the person is trying hard to be fair, because the queries you think of first are the queries you designed the system to handle.

There is a second problem, which is that a demo has no denominator. When the demo works, you learn that the success set is not empty. You learn nothing about its size. And the size is the only thing that matters, because a product used a million times a day will encounter its failure modes a hundred and fifty thousand times a day if the failure rate is fifteen percent, and every one of those is a person.

The replacement for a demo is a fixed set of inputs, chosen before you look at the outputs, scored against criteria written before you look at the outputs, producing a rate. A better demo will not get you there, and a longer one will only take more of your afternoon. The rate can be bad. That is fine. A bad rate you can see is worth more than a good demo you cannot generalize from.

A second-order effect of the demo problem deserves its own warning, because it shapes what gets built rather than only what gets shipped.

When the evidence for a feature is a demo, the incentive is to optimize the demo. Teams tune the prompt against the twelve queries they show in the review. They fix the specific failures that embarrassed them in front of leadership. They add an instruction to handle the case that a vice president happened to try. Every one of these is a rational local action and the sum of them is a system that performs beautifully on a set of inputs nobody outside the building will ever type.

The correct response is to make the evaluation set the artifact of record, and to make it larger than any one person can hold in their head. A number computed over five hundred cases cannot be gamed by memorizing twelve of them, which is precisely why the number is worth having.

The problem of subjective correctness

Here is where the discipline gets genuinely hard, and I want to be careful not to wave it away.

For a large fraction of the questions a production assistant receives, there is no ground truth. A user asks whether a laptop is good for video editing. The honest answer depends on what they edit, at what resolution, on what budget, with what tolerance for render times, and none of that is in the question. Several different answers are defensible. Several more are indefensible. The line between them is drawn by judgment, and judgment varies.

Engineers hate this, and their first instinct is to make it go away by picking a proxy that is objective. Answer length. Reading level. Presence of a citation. Cosine similarity to a reference answer. These are all measurable, and measuring them feels like progress, and every one of them can be satisfied perfectly by an answer that is wrong.

The similarity metrics deserve a specific warning, because they have a long and mostly discouraging history. BLEU, introduced by Kishore Papineni and colleagues at IBM in 2002 for machine translation, and ROUGE, introduced by Chin-Yew Lin in 2004 for summarization, both measure overlap between a candidate text and a reference text. They were useful for the tasks they were designed for, where the space of acceptable outputs is narrow. Applied to open-ended generation, they punish the model for saying the right thing in different words and reward it for saying the wrong thing in familiar ones. BERTScore, from Tianyi Zhang and colleagues in 2020, improved matters by comparing embeddings rather than tokens, but it inherits the same basic assumption: that there is a reference, and that closeness to the reference is goodness. For a product where the reference is one of many valid answers, that assumption does not hold.

What does work is to stop trying to compare the answer to a reference and start scoring it against criteria. Rather than asking whether the answer matches, ask whether it did the specific things a good answer must do and avoided the specific things a good answer must not do. Did it address the question that was asked. Did every factual claim trace to available evidence. Did it disclose the relevant limitation. Did it avoid recommending an action the policy prohibits. Those questions have answers, and two trained reviewers will mostly agree on them, and that mostly is what makes the whole enterprise possible. Chapter Eight is about how to write criteria sharp enough to survive contact with a second reviewer.

Multi-turn quality and conversational drift

Single-turn evaluation is the version of this problem that everyone builds first, because it is the version you can build in an afternoon. You take a query, you get an answer, you score the answer. It is tractable and it is a real signal and it will catch real bugs.

It will also miss the entire class of failures that customers complain about most.

Consider an enterprise support assistant that answers account, billing, and troubleshooting questions. Test it with a hundred single-turn questions and it does well. Its answers about billing cycles are correct. Its troubleshooting steps are in the right order. Its policy statements match the policy documents. You would ship it, and you would be right to feel good about it.

Now put it in front of a person. The person asks about three subscription tiers. The assistant lists them. The person types: what about the second option, does that include the API. And the assistant answers about the third option, or about the first, or asks the person to clarify which option they mean, which is the most infuriating outcome of all because the option is right there on the screen and any human being would know.

The system did not get worse at billing. It failed at something the single-turn suite never asked it to do, which is to carry the state of a conversation forward and resolve a reference against it. The phrase the second option is a pointer. Resolving it requires knowing what was said, in what order, and which items were enumerated. If the conversation history was truncated, if it was serialized in a format the model could not read, if the enumeration was rendered as a UI component whose contents never made it into the text sent back to the model, then the pointer dangles, and the model does what models do with a dangling pointer, which is guess.

This is the failure that motivated most of my thinking about this book. It is a category error to call it a model problem. The model was never given the information it needed. The failure is in context construction, and it is invisible to any evaluation that looks at one turn at a time.

Drift is the slower cousin of this failure. Over six or eight turns, a conversation accumulates constraints. The user mentioned a budget. Then they mentioned they have a small apartment. Then they said they already own one of the accessories. A good assistant holds all of that and applies it. A drifting assistant applies the most recent constraint and quietly forgets the earlier ones, usually because the earlier ones fell off the front of a truncated context window. Every individual response looks fine. The conversation, taken as a whole, is a slow-motion failure, and the user experiences it as the assistant not listening.

There is a reproduction technique that costs nothing and that almost nobody uses, and it deserves to be standard practice.

Run the failing case ten times. Not once. The single most common triage error in this discipline is a developer trying a reported failure once, seeing a good answer, and closing the ticket as not reproducible. A failure that occurs one time in five will look fixed on the first attempt eighty percent of the time, and the ticket will be closed, and the failure will continue, and the customer who reported it will conclude that reporting things is pointless.

Ten runs and a count is the minimum. What you want out of triage is a rate rather than a verdict, and a rate of two in ten is a bug with a size attached, which is a thing an engineer can work on and a product manager can prioritize.

Why production AI failures are hard to reproduce

In conventional debugging there is a moment of relief when you get a reliable reproduction. From there it is craft. You bisect, you instrument, you narrow, you find it.

Generative failures resist reproduction in a way that is worth spelling out, because underestimating this is how teams end up with a bug tracker full of tickets nobody can close.

Sampling is the obvious source. Unless you have pinned the temperature to zero, which most production systems do not because it makes responses feel mechanical, the same input yields different outputs. A failure that occurs one time in ten will not show up when you try it once, and a developer who tries it once and sees a good answer will close the ticket as not reproducible, which is both true and useless.

Then there is state. The failing conversation had a history. Reproducing it requires reconstructing that history exactly, and if your logging captured the final user query but not the assembled context, you have thrown away the input. This is the single most common gap I see. Teams log the request and the response. The request is not the input to the model. The input to the model is the request plus the retrieved evidence plus the conversation summary plus the system prompt plus whatever the tool layer injected, and if you did not log that, you cannot reproduce anything.

Then there is the world. Retrieval indices are rebuilt. Documents are updated. Product catalogs change hourly. The evidence the model saw last Tuesday no longer exists, and the query that failed last Tuesday now retrieves something different and works fine. The bug is real, it is gone, and it will come back the next time the index is in that state.

And then there is the provider. If you call a hosted model, the thing behind the endpoint is not a constant. Serving stacks are updated, quantization changes, safety layers are tuned. You may be told about the significant changes. You will not be told about all of them. I have watched a quality metric shift by several points across a weekend during which our team merged nothing at all.

The response to all of this is to log enough that reproduction becomes possible, and to measure in aggregate so that the unreproducible individual case shows up as a rate that moves. Despair is available, and it is not useful. A failure you can measure at a rate of eight percent is a failure you can work on, even if you can never reliably trigger it on your laptop. Chapter Seventeen is about what to log to make this possible, and I would argue it is the single highest-return investment in the entire book.

There is a version of this failure that is specific to the way generative products get built, and I want to name it because it is common and it is expensive.

A team prototypes in a notebook. The prototype works, in the sense that the outputs look good when a person reads them. The prototype becomes a service. The service becomes a product. At no point in that progression does anybody introduce a measurement, because at no point does the thing visibly break, and the prototype's evidence, that the outputs look good when a person reads them, silently remains the strongest evidence anyone has for the entire product, right up to the launch.

The moment to introduce measurement is at the prototype, and it costs almost nothing there, and it will feel absurd because you have twelve examples and they all work. Write the criteria down anyway. The team that has a rubric before it has a service is the team that never has to retrofit one onto a product that is already failing.

One more asymmetry, and it is the one that determines how much of your budget this discipline deserves.

In conventional software, a defect costs you once. The bug fires, somebody notices, you fix it, and the cost is bounded by the time it took. In a generative product, a defect costs you every time it fires, silently, for as long as it goes undetected, and detection is the hard part rather than the fix. The citation defect from the opening of this chapter ran for nineteen days and the fix took an afternoon.

That ratio, nineteen days of damage against an afternoon of repair, is the argument for spending on detection rather than on repair. Almost every generative failure I have worked on has had that shape. The fix is cheap. Knowing is expensive. Build for knowing.

The need for evaluation as an engineering discipline

Everything above points in one direction. The gap between what traditional testing gives us and what generative systems require is not going to be closed by writing more unit tests. It requires a parallel discipline, with its own artifacts, its own tooling, its own place in the release pipeline, and its own owner.

That discipline needs a dataset, because you cannot measure without a fixed set of inputs. It needs rubrics, because you cannot score without criteria. It needs automation, because a process that requires a human to run it will not be run. It needs a gate, because a measurement that does not block a bad release is a measurement that will be ignored under deadline. And it needs continuous refresh, because the traffic changes and a dataset that reflects last year's users measures nothing.

There is precedent for this. Machine learning teams learned an expensive version of the same lesson a decade ago. In 2015, D. Sculley and colleagues at Google published a paper about hidden technical debt in machine learning systems, and their central observation was that the model is a small box inside a very large diagram, and that almost all the risk lives in the boxes around it. Configuration, data dependencies, feature extraction, monitoring, the analysis debt of not knowing why the system does what it does. The same diagram applies to generative applications with the labels changed.

The version of the lesson I would add is about the data itself. Nithya Sambasivan and her coauthors at Google Research published work at the 2021 CHI conference on what they called data cascades, the compounding downstream damage caused by upstream data problems that nobody thought were important enough to fix. Their summary of the culture they observed has stayed with me: everyone wants to do the model work, nobody wants to do the data work. Evaluation is data work. It is unglamorous, it is slow, and it is where the quality of your product is actually determined.

The table below sets the two disciplines side by side. It is the compact version of the argument in this chapter, and it is worth putting somewhere your team can see it, because the distinctions in it will come up in every design review you have for the next year.

Dimension	Traditional Software Testing	LLM Evaluation
Output	Deterministic; same input, same output	Probabilistic; same input, varying output
Correctness	Decidable against a specification	Judged against criteria; often several valid answers
Assertion	Exact equality or an explicit condition	Rubric score, threshold, or pairwise preference
Pass signal	Binary pass or fail per test	A rate across a population of cases
Failure visibility	Exception, timeout, failed assertion	Silent; response is well-formed and wrong
Reproduction	Reliable given the same input	Often impossible; depends on sampling, state, index, provider
Unit of test	Function, endpoint, component	Turn, conversation, session, and pipeline stage
Regression trigger	A code change in version control	Prompt, model, retrieval, policy, data, or nothing at all
Coverage	Lines, branches, paths	Intents, personas, evidence conditions, failure classes
Cost per run	Effectively free	Real money per case; scales with dataset size
Oracle	The specification	Human judgment, calibrated and sampled
Time to feedback	Seconds in CI	Minutes offline; days or weeks online

Look at the last two rows for a moment, because they are the ones that reshape how a team works. Cost per run means you cannot simply run everything on every commit; you will have to build a tiered strategy, with a fast cheap suite on every pull request and an expensive comprehensive suite on a schedule. Time to feedback means the loop is slower than engineers are used to, which changes how you plan, how you batch changes, and how much you can safely ship at once.

None of this is a reason to skip the work. It is a reason to design the work properly, which starts with a question I have watched a surprising number of teams fail to answer clearly. Before you can measure quality, you have to say what quality means for your product. Not in the abstract. In a list, with definitions, that your team agrees on.

That is the next chapter.

Chapter 2: What Should We Evaluate?

There were seven of us in the room, and I asked everyone to write down, on an index card, what a good response from our assistant looked like. One sentence. No conferring.

The product manager wrote that a good response answers the customer's question. A scientist wrote that a good response is faithful to the retrieved evidence. A designer wrote that a good response is short enough to read on a phone without scrolling. The engineer sitting next to me wrote that a good response arrives in under eight hundred milliseconds. Someone from the policy team wrote that a good response does not make a claim the company cannot stand behind. And the person who had joined the team two weeks earlier, who I suspect was the only one thinking clearly, wrote that a good response makes the customer feel like the thing understood them.

Every one of those is right. They are also mutually constraining, and in some cases directly opposed. Faithfulness to evidence makes responses longer. Brevity makes them less complete. Under eight hundred milliseconds forbids the extra retrieval pass that would have made the answer faithful. The policy constraint sometimes requires the assistant to decline, which will not make the customer feel understood at all.

The meeting had been called to discuss why our quality numbers were not moving. The index cards answered the question. We were not measuring quality. We were each measuring the one dimension we personally cared about, calling it quality, and then arguing.

Evaluation begins with a decomposition. You have to break the vague word into a set of dimensions that can be defined, scored, and weighed against each other, and you have to write the definitions down before anybody starts scoring. This chapter is that decomposition. It is a menu, not a mandate. Most products need six or seven of these. Almost none need all ten equally.

Response correctness

Correctness is whether the factual content of the answer is true. It is the dimension everyone starts with and the one people most often confuse with all the others.

The subtlety is that correctness has to be scoped. True according to what. For a question about a physical fact, correctness is truth in the world, and you can check it. For a question about a company policy, correctness is agreement with the current authoritative policy document, which may itself be out of date relative to what a support agent would actually do, and which changes without telling you. For a question about a product, correctness is agreement with the catalog, which is a moving target updated by a system nobody on your team controls.

So define your oracle. Write down, for each category of question, the source of truth against which correctness is judged. Then accept that the oracle can be wrong, and that when a reviewer marks an answer incorrect because it disagrees with the policy document, you have learned either that the model was wrong or that the policy document is stale, and both of those are findings worth having.

Score correctness at the level of the claim, not the response. A five-sentence answer with four true sentences and one false one is not eighty percent correct in any way that matters, because the false one is the one the customer will act on. Chapter Nine develops claim-level scoring in detail. For now, the rule of thumb is that a response containing a material false claim is incorrect, full stop, and the severity of the falsehood is a separate axis.

Groundedness and faithfulness

Groundedness asks a different question from correctness, and the difference is one of the most useful distinctions in this book.

Correctness asks: is this true. Groundedness asks: did the system have a basis for saying it.

An answer can be true and ungrounded. The model states a fact that happens to be correct, drawn from what it memorized during training, with no support from anything your retrieval layer returned. That is a lucky guess, and lucky guesses are dangerous in production because they are indistinguishable from unlucky ones until a customer gets hurt. An answer can also be grounded and false, when the retrieved document is wrong and the model faithfully repeats it. That is a data problem with a different owner and a different fix.

Separating these two dimensions changes what you do about a failure. Ungrounded but true means your retrieval is weak and you are getting away with it for now. Grounded but false means your corpus is polluted. Ungrounded and false is the classic hallucination. Grounded and true is the target.

The measurement technique that has worked best for me comes out of the factuality research literature. Sewon Min and colleagues, in the FActScore work published in 2023, proposed decomposing a long-form generation into atomic claims and checking each one independently against a source. It is slower than scoring the response as a whole, and it is far more informative, because it gives you a numerator and a denominator: how many claims did this answer make, and how many of them can be traced to evidence. Shipping a product with a claim-support rate you can quote is a different posture from shipping one with a vague sense that it is mostly fine.

Relevance to user intent

The user typed a string. The string was a proxy for something they wanted. Relevance is whether the answer addressed the want.

This is where a lot of technically excellent systems quietly fail. Someone asks whether a particular jacket is waterproof, and the assistant produces four hundred words about the fabric technology, the manufacturing process, and the care instructions, every word of it true and grounded, without ever saying yes or no. The customer wanted a yes or no. They got a brochure.

Intent has depth. There is the literal question, and there is the goal behind it, and sometimes there is a goal behind that. Someone asking about the return window is often not curious about the return window. They are deciding whether to take a risk on a purchase, and the answer that serves them says thirty days and adds that returns on this category are free, which is the fact that actually changes their decision.

Scoring relevance requires you to record, alongside each test case, what the user was trying to accomplish. This is annotation work and it is tedious and it is the difference between an evaluation set that measures answering and one that measures helping. When you mine production traffic in Chapter Six, capturing intent is the most valuable and most expensive thing you will do to each conversation you sample.

Watch also for the opposite failure, which is answering more than was asked. An assistant that responds to a narrow question with a comprehensive briefing has quietly offloaded the work of finding the answer back onto the person who asked. Thoroughness is the wrong word for it.

Completeness and specificity

Completeness is whether the answer contains everything the person needed. Specificity is whether it committed to anything.

Incompleteness is easy to detect once you look for it. The answer explains three of the four steps. It states the price but not that the price excludes shipping. It recommends an action but omits the precondition that makes the action legal. A rubric catches this by listing required elements per test case, which is one of the reasons Chapter Five argues for expected-behavior specifications over golden answers. A required-elements list is checkable. A golden paragraph is not.

Vagueness is harder, because it hides. Models trained with heavy safety and helpfulness tuning develop a habit of producing answers that are technically responsive and operationally worthless. It depends on your situation. You may want to consult a professional. Several factors are relevant. Each of those sentences is unfalsifiable, and a system that produces them has learned to satisfy a helpfulness objective without taking any risk, which is a rational strategy for a model and a terrible outcome for a product.

I have found it useful to score specificity explicitly, with a criterion that reads roughly: does the answer commit to a position, a number, or an action that could be wrong. If nothing in the answer could be wrong, nothing in the answer is worth much. This one criterion has surfaced more product-quality problems for me than any other single line in a rubric.

Safety and policy compliance

Safety is a broad word and it is worth splitting it. There is harm, there is policy, and there is legal exposure, and they have different owners and different thresholds.

Harm covers the categories any responsible deployment worries about: content that could hurt someone, content that targets a group, content that assists a dangerous act. Providers do a great deal of work here and the base model will refuse most of it, but your application composes the model with retrieval and tools, and composition creates surface area. A model that would never volunteer a piece of dangerous information may repeat it faithfully if your retrieval layer hands it a document that contains it.

Policy is where most production teams actually spend their time. These are the rules your company has, which the model has never heard of. Do not promise a refund. Do not give medical advice. Do not compare against a named competitor. Do not state a delivery date. Each rule is an evaluation criterion, and each one needs test cases designed to tempt the system into breaking it. A policy nobody has tried to break is a policy you have no evidence for.

Refusal quality belongs here too, and it is underrated. A system that refuses correctly but rudely, or refuses correctly without explaining why, or refuses things it should have answered, is failing even though the safety metric is green. Over-refusal is a real cost, it degrades the product, and it is invisible to any metric that only counts unsafe outputs. Chapter Twenty-Three treats this at length.

Tone and communication quality

Tone gets dismissed as cosmetic. I would push back on that. Tone is how the system signals its own reliability, and users read it constantly.

An assistant that is breezy about a serious question reads as untrustworthy. An assistant that is ponderous about a trivial one reads as broken. An assistant that apologizes three times in a

paragraph reads as if it does not know what it is doing, which, when it is apologizing for the third time, it probably does not.

There is also a specific failure that the research community has started to name. Mrinank Sharma and colleagues at Anthropic published work in 2023 on sycophancy in language models, the tendency of a model to agree with a user's stated position rather than maintain a correct one under pushback. In a product this is a quality catastrophe dressed as politeness. The user says that is wrong, and the assistant, which was right, folds. Any evaluation that scores agreeableness will reward this behavior. Any evaluation that scores correctness under disagreement will catch it, and I would put at least a few such cases in every dataset.

Tone is scoreable. Write down the three or four adjectives your product is supposed to embody, define what violates each, and have reviewers mark violations. It will feel soft. It will also catch the release where somebody changed the persona instruction and made the assistant chatty in a domain where chattiness reads as incompetence.

One dimension deserves a warning because it is the one teams most often measure without noticing what they have done.

Confidence is not a quality dimension and it correlates with almost all of them. A model that hedges everything will score badly on specificity and well on safety. A model that commits to everything will score the reverse. If your rubric does not separate the two, you will end up optimizing the system's rhetorical register rather than its accuracy, and the direction you optimize in will be whichever one your reviewers happen to prefer.

So score the claim and the confidence separately. Was the assertion correct, and was the degree of certainty it expressed appropriate to the evidence behind it. A system that says probably, and is right, has behaved better than one that says definitely, and is right, if the evidence supported only the first.

Multi-turn consistency

This dimension asks whether the system behaves like it has been in the conversation.

The concrete checks are narrow and testable. Does it resolve references, so that the second one points at the second one. Does it carry constraints forward, so that a budget mentioned in turn two still applies in turn seven. Does it accept a correction, so that when the user says no, I meant the blue one, the blue one stays selected. Does it avoid re-asking for information it already has, which is the single most reliable way to make a person feel like they are talking to a machine. Does it stay consistent, so that the recommendation in turn eight does not contradict the recommendation in turn four without acknowledging that something changed.

There is also a freshness question that cuts the other way. State should persist, but not forever. If a user finishes shopping for a gift and starts shopping for themselves, the assistant that is still applying the gift constraints is being consistent and being wrong. Knowing when to drop state is as much a design decision as knowing when to keep it, and your evaluation should have cases for both.

Chapter Ten is entirely about this dimension, because it is the hardest one to measure and the one where the industry's tooling is thinnest.

Tool-use correctness

Once the model can act, the stakes change shape. A wrong answer is a wrong answer. A wrong action is a refund issued to the wrong account, an email sent to a customer who did not want it, a

ticket closed that should have escalated.

The dimensions here are structural. Did the system choose the right tool, or any tool at all when it should have used one. Were the arguments correct, including the ones the model had to infer rather than read. Did it have permission, and did the check actually happen rather than being assumed. Did it confirm before doing something irreversible. Did it avoid doing the same thing twice when the first attempt timed out but actually succeeded, which is the idempotency failure and it will bite you.

And then the recovery dimension, which teams almost always skip. The tool failed. What did the system do. Did it retry sensibly, tell the truth about what happened, and stop before it made things worse. Chapter Eleven is a full treatment. The reason it is worth a chapter is that agentic failures are the only ones in this book that can cost real money in a single request.

Latency and cost

Quality that arrives too late does not count as quality. Treat that as the finding rather than as a grudging compromise, because any evaluation that files speed under somebody else's problem is measuring an artifact rather than a product.

For streaming interfaces, the number that matters most is time to first token, because that is when the user stops wondering whether the thing is broken. Total generation time matters too, but a fast first token with a steady stream reads as responsive even when the full answer takes six seconds. A slow first token reads as failure even when the full answer takes three.

Cost is the constraint that determines what you are allowed to build. Every technique in this book that improves quality does so by spending something: more retrieval, longer context, more reasoning tokens, a second pass, a judge. Each of those has a price per request, and the price times your request volume is a number that will appear on a slide in front of someone who can cancel your project.

So put them in the rubric. Every evaluation run should report quality, latency, and cost together, as a triple, because a change that improves quality by two points and doubles cost is a decision, and decisions should be made by people who can see all three numbers. Chapter Twenty-Two builds the decision matrix.

A dimension that belongs on this list and that I have never seen on anyone's rubric: whether the system did the same thing twice.

Ask a production assistant the same question ten times and read the ten answers side by side. In most systems, two or three of them will differ on something material, and the team will be surprised, because nobody has ever looked. Customers do look. They ask a friend, the friend asks the assistant, and the two of them compare notes, and the product has just told two people different things about the same policy.

Self-consistency is measurable and cheap. Run each case several times and report the variance alongside the mean. A case with a mean score of four and no spread is under control. A case with a mean of four and a spread from one to five is a coin flip with a good average, and the people it fails are not comforted by the average.

User satisfaction and task completion

The last dimension is the only one that is not a proxy, and it is the hardest to get.

Did the person get what they came for. Not did the answer look good. Did they buy the thing, resolve the issue, close the ticket, stop asking. This is the outcome, and every other metric in this chapter is an attempt to predict it from evidence available at generation time.

The signals are noisy. Explicit feedback, the thumbs up and thumbs down, has a response rate low enough that the population who click are not the population who use, and they skew angry. Implicit signals are richer and more ambiguous: the user rephrased, which might mean the answer failed or might mean they thought of something else. The user abandoned, which might mean frustration or might mean they got what they needed in the first sentence and left. The user escalated to a human, which is the clearest negative signal you will get and the one worth building a pipeline around.

The right posture is to treat offline quality metrics as a leading indicator and outcome metrics as the ground truth, then check periodically whether the leading indicator still leads. When your rubric score goes up and your escalation rate does not go down, your rubric has drifted away from the thing you care about, and it is time to fix the rubric rather than celebrate the score. Chapter Twenty gives the machinery.

The quality dimensions checklist

Here is the whole decomposition in one place. Use it as a starting menu when you set up a new evaluation program, and be deliberate about what you drop. Every dimension you leave out is a category of failure you have decided not to see.

Dimension	The question it answers	Typical scoring approach
Correctness	Is the factual content true, against a defined oracle?	Claim-level check against source of truth
Groundedness	Did the system have evidence for what it said?	Claim extraction and source matching
Relevance	Did it address the actual intent behind the question?	Rubric score against recorded intent
Completeness	Did it include everything the person needed?	Required-elements checklist per case
Specificity	Did it commit to anything falsifiable?	Binary criterion; vagueness is a failure
Safety	Could this content cause harm?	Category classifiers plus adversarial cases
Policy compliance	Did it obey the rules the company set?	Per-rule assertion with tempting test cases
Refusal quality	Did it decline correctly, and only when it should?	Two rates: unsafe passes, and over-refusals
Tone	Does it sound like the product it belongs to?	Violation marking against named adjectives
Multi-turn consistency	Does it behave like it was in the conversation?	Conversation-level scripted tests
Tool-use correctness	Right tool, right arguments, right permissions?	Structured assertions on the call, not the prose
Recovery	Did it handle tool failure without making it worse?	Fault-injection cases
Latency	Did it arrive in time to be useful?	Time to first token and end-to-end, at p50 and p99
Cost	What did this answer cost to produce?	Tokens in, tokens out, retrieval and judge calls

Dimension	The question it answers	Typical scoring approach
Task completion	Did the person get what they came for?	Online outcome metrics; escalation and abandonment

One warning before we move on. Having fifteen dimensions does not mean reporting fifteen numbers to your leadership every week; that is how evaluation programs die. Pick two or three headline metrics that your organization will actually watch, keep the rest as diagnostics that you look at when a headline moves, and be explicit about which is which.

And now the part that most teams skip. Every dimension in this chapter has been described as a property of the response. Almost none of them are caused by the response. They are caused upstream, in a pipeline with a dozen stages, each of which can be evaluated on its own terms. That pipeline is the subject of the next chapter.

Chapter 3: The Full LLM Application Evaluation Stack

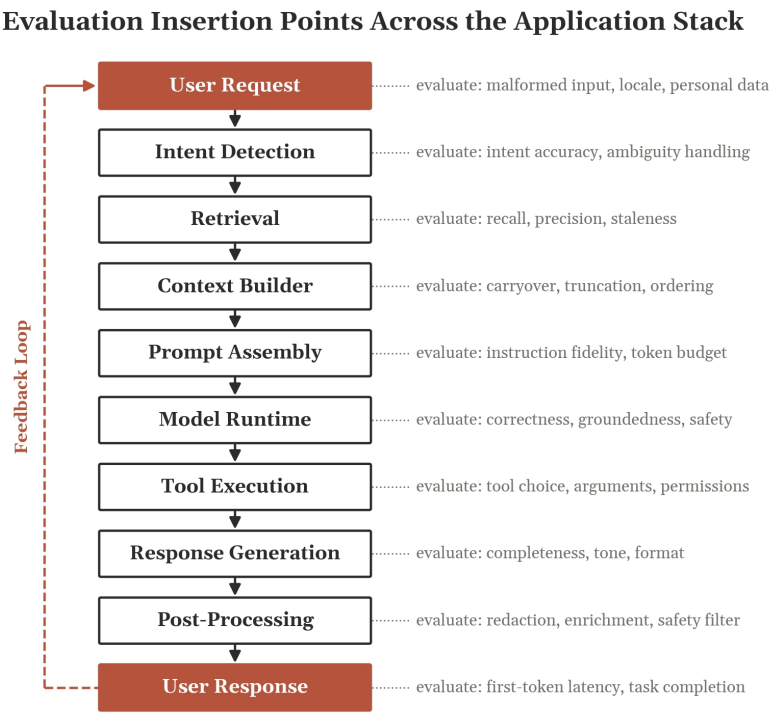
Somebody had written the word HALLUCINATION at the top of the whiteboard, in capitals, with a box around it. Underneath were twelve examples, pulled from a week of complaints, each one a case where the assistant had stated something that was flatly untrue.

We spent three hours on those twelve examples. By the end, four of them had nothing to do with the model.

In two cases, the retrieval layer had returned documents about a discontinued version of the product, because the index still contained them and the ranker had no notion of recency. The model read what it was given and reported it accurately. In one case, the conversation summary had compressed a customer's stated constraint into a phrase that inverted its meaning, so the model was answering a question the customer had not asked and had, in fact, explicitly ruled out. In one case, the answer had been correct when generated and had been mangled by a post-processing step that stripped a qualifying clause because it looked like boilerplate.

Four out of twelve. A third of what we had confidently labeled a model problem turned out to be a problem in a component that had never been evaluated by anyone, because nobody owned evaluating it, because we had all silently agreed that evaluation meant looking at the final text.

The rest of this chapter is an argument against that agreement. A generative application is a pipeline. Every stage of the pipeline can fail. Every stage of the pipeline can be evaluated. And the evaluation you do at each stage is cheaper, faster, and more diagnostic than the evaluation you do at the end, because at the end everything has been mixed together and you cannot get it apart again.



Evaluating input understanding

The first thing that happens to a user request is that something tries to figure out what it is. Sometimes this is an explicit intent classifier. Sometimes it is a router that picks between a cheap path and an expensive one. Sometimes it is a lightweight model deciding whether retrieval is needed at all. Whatever it is, it makes a decision, and a wrong decision here is unrecoverable downstream.

The good news is that this stage is the most conventional part of the pipeline. It produces a label, or a small structured object, and labels can be evaluated with the classification machinery the field has had for thirty years. Precision, recall, a confusion matrix. Nothing exotic.

The failures worth building cases for are the ones that look small. Ambiguity, where the request could reasonably be two things and the router silently picks one. Compound requests, where the user asks two questions in one sentence and the classifier assigns a single label, discarding half the

request before anything else has run. Language and locale, where an input in a language your classifier was not trained on gets routed to a default that produces a confident answer in the wrong context. And the injection case, where the input contains text designed to look like an instruction, which matters more once tools are in play.

Look hard at your confusion matrix rather than your accuracy number. An intent router that is ninety-four percent accurate can still be sending every one of a specific high-value intent to the wrong place, and the aggregate number will happily hide that.

Evaluating retrieval quality

If your system retrieves, then retrieval is where most of your quality is decided, and I would go further: in most retrieval-augmented products I have looked at, a majority of the answer failures are retrieval failures wearing a costume.

The model cannot state a fact that was never put in front of it. If the right passage was not retrieved, the model has three options, all bad. It can say it does not know, which is honest and unhelpful. It can answer from parametric memory, which is ungrounded and might be stale. Or it can construct something plausible from the wrong passages it was given, which is the failure people call hallucination and which is really a failure of supply.

Retrieval has its own metrics and they predate all of this. Recall at k asks whether the passage you needed appeared anywhere in the top k. Precision asks how much of what you retrieved was worth retrieving. Mean reciprocal rank asks how far down the list the good passage was, which matters because position affects how much attention the model pays. Measuring these requires labeled data: for a set of queries, which passages are actually relevant. Building those labels is the tedious work that teams skip, and skipping it means you will spend the next year debugging the model instead of the index.

Beyond the classic metrics, three things break retrieval in production and none of them show up in recall at k. Chunking, where the passage that answers the question got split across a boundary and neither half is retrievable. Staleness, where the index contains the truth as of last month. And conflict, where two retrieved documents disagree and nothing in the

pipeline decides which one wins, so the model does, arbitrarily. Chapter Fifteen is devoted to all three.

Evaluating context assembly

Now you have some retrieved passages, a conversation history, a user profile perhaps, some system state, and a fixed token budget. Something has to decide what goes in, in what order, and what gets dropped. That something is the context builder, and in my experience it is the single most under-evaluated component in the entire stack.

It is under-evaluated because it does not feel like a component. It is often a hundred lines of string manipulation in the middle of a service, written by whoever was there at the time, modified by everyone since, and covered by no tests beyond the ones that check it does not throw.

It is also where the multi-turn failures come from. The truncation policy is the whole ballgame. If the builder drops the oldest turns when the budget is exceeded, it will drop exactly the turn where the user stated the constraint that everything since has depended on. If it summarizes instead, the summary is a lossy compression that some other model produced and nobody checked. If it drops evidence to make room for history, the answer becomes ungrounded. If it drops history to make room for evidence, the answer forgets the conversation. There is no free choice here, and the choice is being made, right now, in your codebase, probably by a magic number.

The evaluation is direct and nobody does it. Take the assembled context, before it goes to the model, and score it. Is the information required to answer this question present in this context. That is a yes-or-no question, it can be answered by a human in ten seconds or by a cheap model in a hundred milliseconds, and it splits your entire failure population into two piles: the model had what it needed and got it wrong, or the model never had a chance. Those two piles have completely different owners and completely different fixes. Any team that cannot separate them is debugging blind.

Position matters too. Nelson Liu and colleagues at Stanford showed in Lost in the Middle that models retrieve information most reliably from the beginning and end of a long context and least reliably from the middle. If your builder appends the most relevant evidence after a long conversation

history, you have put the most important thing in the worst position. That is testable. Permute the order, hold everything else fixed, and measure.

Evaluating prompt construction

The prompt is code. It is code with no type system, no compiler, and no static analysis, which means it deserves more testing than your code, not less.

What can go wrong is more mechanical than people expect. Template variables that render empty and leave a dangling colon followed by nothing, which the model will interpret as an empty instruction and handle unpredictably. Instructions that contradict each other, usually because two teams added rules six months apart and nobody read the whole prompt end to end. Escaping failures, where retrieved content containing the same delimiter you use for your sections silently reshapes the structure of the prompt. Formatting drift, where a schema you asked for in the instruction no longer matches the schema your parser expects, because someone updated one and not the other.

Assert on the rendered prompt, not the template. Snapshot it in tests. Check that every section is present, that no placeholder survived, that the token count is inside budget, and that the total length has not silently grown by forty percent since last quarter, which it has, because prompts only ever grow.

And put the instruction near the thing it governs. The eleven-line pull request from the last chapter is the whole lesson: a rule about citations, placed a thousand tokens away from the evidence it applied to, stopped working. You will never find that bug by reading the diff. You find it by running a suite that checks whether answers cite their sources, which is to say, by having built the discipline this book is about.

Evaluating model output

This is the stage everyone means when they say evaluation, and by the time we get here I hope it looks smaller than it did at the start of the chapter.

The model takes a fully assembled context and produces text. Given that input, the questions are the ones from Chapter Two. Is it correct. Is it grounded in what was supplied. Does it follow the instructions it was given. Is it safe. Is it in the right register.

The essential discipline here is to hold the input fixed. If you are evaluating the model, you have to compare it against itself, or against another model, on identical assembled contexts. The moment you allow the retrieval to differ between two runs, you are no longer measuring the model. You are measuring the system, which is a valid thing to measure and a different thing, and confusing the two is how a team spends six weeks migrating to a better model and gets a worse product.

Run the model stage with cached contexts. Store the exact context that produced each logged answer, replay it against a candidate model, and compare. This is the offline comparison that makes Chapter Fourteen possible, and it is worth the storage.

Evaluating tool calls and actions

When the model emits a tool call, the artifact you are evaluating is structured, and that is a gift. You do not have to judge prose. You have a function name and a dictionary of arguments, and you can assert on them exactly the way you would assert on any other function call in a test.

Did it call the right function. Did it call one at all, in a case where the correct behavior was to answer from context without touching anything. Are the argument values right, particularly the ones the model had to derive rather than copy, like a date computed from next Tuesday or an account identifier resolved from an earlier turn. Are the types right. Is anything present that should have been redacted.

The critical property that people forget: authorization has to be checked by the system, not by the model. A model instructed not to issue refunds above a threshold will usually comply and will occasionally not, and once in a while a user will discover the phrasing that gets it to. The check belongs in the tool layer, where it is deterministic, and your evaluation should include cases specifically designed to talk the model into crossing a line, so you can confirm that crossing it does nothing because the layer below refused.

Then there is the counting problem. If the tool call times out and the system retries, and the first call actually succeeded, the customer gets charged twice. Idempotency keys solve this and evaluation confirms it, with fault-injection cases that simulate a timeout on a call that succeeded server-side. Almost nobody writes those cases. Chapter Eleven insists on them.

Evaluating post-processing

The model finished. The answer was good. And then your code got hold of it.

Post-processing does a lot of work in a mature system, and every step of it is a place where a good answer becomes a bad one. Safety filters redact. Enrichment layers inject product cards, images, links, and prices. Formatters convert markup. Truncators cut a response that exceeded a display limit. Rankers reorder. Each of these is a transformation, each transformation has a failure mode, and none of them are covered by an evaluation that scores the model's raw output.

The truncation failure is the one I see most: an answer that ends mid-sentence because a UI constraint cut it at a character count, leaving a conditional statement with its condition and no consequence. The enrichment failure is the most damaging: a product card injected next to a paragraph that recommended a different product, so the text says one thing and the card shows another, and the customer believes the card.

The evaluation is a diff. Score the model output, score the post-processed output, and look at every case where the score dropped. If your pipeline is healthy that set is empty. In my experience it is never empty the first time anyone looks.

A stage that is missing from the diagram because it does not exist in most architectures, and that I would argue should.

A pre-generation check on the assembled context. Before the expensive model call, a cheap model reads the context and answers one question: does this contain what is needed to answer this question. If the answer is no, the system already knows the response will be bad, and it has options. It can retrieve again with a different strategy. It can ask the user a clarifying question. It can say it does not know.

What it usually does instead is call the model anyway, produce a confident answer built from insufficient evidence, and discover the problem when a customer complains. The check costs a fraction of a cent and it converts a hallucination into an admission, which is the best trade available anywhere in this pipeline.

Evaluating streaming and partial responses

Streaming is not a delivery detail. It changes what the user sees, and therefore it changes what quality means.

In a streamed response, the user reads the beginning before the end exists. That means a correction the model makes at the end arrives after the reader has already formed a belief. It means an answer that starts confident and ends with a caveat will be received as confident. It means a safety filter that operates on the complete text has already lost, because the text was on the screen before the filter saw it.

So evaluate the stream. Time to first token, because that is the number the user feels. The stability of the prefix, meaning whether the beginning of the answer would have been the same beginning if the model had known how the answer ended. Partial safety, meaning whether any prefix of the stream is harmful even if the completed response is fine. And termination behavior, which is where the ugliest bugs live: the connection drops at token four hundred, and what does your system do with the four hundred tokens the user already read. Does it log a truncated answer as if it were complete, poisoning your metrics with a failure that was actually a network event.

Early termination is worth a category of its own in your failure taxonomy, and Chapter Eighteen gives it one, because it is common, it is invisible, and it corrupts every other number you collect.

A note about where these checks live, because the natural instinct is to build them all into one large evaluation service and that instinct is wrong.

The stage-level checks belong to the stage. The assertion that a retrieved passage was relevant belongs in the retrieval team's test suite, running on their pull requests, on their schedule. The assertion that the assembled context contains the required evidence belongs to whoever owns the context builder. Centralizing every check into a single evaluation pipeline produces a queue, and the queue is owned by one team, and that team becomes the bottleneck for everybody else's quality work.

What belongs in the central platform is the standard: the schema, the rubric library, the judge calibration, and the gate. What belongs to each component team is the coverage of their own component. This division is the difference between an evaluation program that scales and one that turns into a service desk.

Evaluating persistence and conversation state

The last stage is the one that runs after the response is delivered, and it determines what the next turn will be able to see.

Something has to write the turn to storage. Something has to decide what gets summarized and what gets kept verbatim. Something has to decide how long state lives, when a conversation is considered over, and what happens when the same user opens a new session on a different device an hour later.

These decisions produce failures that are diabolical to debug, because they manifest one turn after the mistake, or ten. A write that silently fails means the next turn has no memory of this one, and the user experiences an assistant that forgot something it just said. A summarizer that drops a negation means every subsequent turn operates on an inverted constraint. A session boundary drawn in the wrong place means a user returning to finish a task starts over.

Evaluate this by round-tripping. After a turn, read the persisted state back and assert against it. Does it contain the entities that were established. Does it contain the constraints. Does the summary, if there is one, preserve every negation and every quantity from the source. Do this in the test suite, on every change to the persistence layer, and you will catch a class of bug that would otherwise be reported to you as the model being stupid.

A last observation about this map that took me longer to appreciate than it should have.

The stages are not equally instrumented in any organization I have seen, and the pattern of what is missing is remarkably consistent. Retrieval is well instrumented, because search teams have been measuring recall for thirty years and they brought their tooling with them. The model is well instrumented, because that is where the attention is. Everything between them is dark.

That gap has nothing to do with any particular company. It is a consequence of how these systems get built: retrieval and generation are recognizable components with owners and prior art, and the code that joins them is written by whoever is joining them, under deadline, as glue. Glue does not get a test suite. And the glue is where the conversation state lives, where the truncation happens, and where the ordering is decided, which is to say it is where a large share of your quality is being determined by a decision nobody remembers making.

What the map is for

The stage-by-stage view does three things for a team, and it is worth naming them because they are the reason to do the extra work.

It assigns ownership. When a failure is attributed to the context builder, there is a person who owns the context builder, and they can fix it. When every failure is attributed to the model, there is nobody, and the bug sits in the backlog under a label that means nothing.

It makes evaluation cheaper. Scoring an assembled context for information sufficiency is faster and less expensive than generating a full response and having a human read it. Catching the failure at the retrieval stage saves you the cost of everything downstream.

And it makes the numbers honest. A single end-to-end quality score, moving up and down, tells you that something changed. The stage-level breakdown tells you what. Without it you are running a system whose only instrument is a light that turns red, and you will spend your incidents guessing.

That is the theory. The rest of this book is the practice, and the practice starts where all measurement starts, which is with the question of what you are going to measure against. You need a dataset. Almost certainly you do not have one, and the one you think you have is a text file somebody made during the prototype that nobody has looked at since.

Part Two is about building one properly.

Part Two

Designing Evaluation Datasets

Chapter 4: Building the First Evaluation Dataset

I want to tell you about a file called eval_queries.txt, because a version of it exists inside almost every company that has shipped a generative feature.

It was created during the prototype, in week two, by an engineer who needed something to try the model on. It has forty-one lines. Thirty-eight of them are questions somebody thought of at their desk. Three of them are questions a real customer asked, pasted in from a Slack thread. It has no answers, no metadata, no owner, and no history, because it lives in a repository nobody remembers to update and it has not been touched in eleven months.

And it is still, today, the thing the team runs before a launch. Somebody opens it, pipes it through the assistant, reads the outputs, and says the responses look reasonable. That sentence, the responses look reasonable, is the sum total of the evidence supporting a feature used by several million people.

The uncomfortable part is that this is an improvement over having nothing. The file is not useless. It catches catastrophic failures. It just cannot catch anything else, and everyone involved knows it, and nobody wants to say so out loud because the alternative is a great deal of unglamorous work that no promotion committee will ever ask about.

This chapter is about that work. It is the least exciting thing in this book and it pays back more than anything else here. If you build a real evaluation dataset and do nothing else I recommend, you will still be better off than most teams shipping generative AI today.

Why evaluation datasets fail

Before the recipe, the failure modes, because a dataset can exist and still be worthless in four distinct ways.

It is unrepresentative. Somebody sat down and imagined what users would ask. What users actually ask is stranger, shorter, more misspelled, more ambiguous, and much more concerned with edge cases in your policy than anything an engineer would think to write. A dataset of imagined queries measures how well your system handles the queries your team can imagine, which was never the question.

It is too easy. Test cases drift toward the cases the system already handles, because those are the ones that produce a clean pass and a good feeling. Over time, an evaluation suite that starts at seventy percent will climb to ninety-five percent, and every point of that climb will be earned partly by improving the system and partly by quietly removing or rewording the cases that kept failing. Nobody decides to do this. It happens.

It is stale. The traffic changed. A new product category launched, a new policy was written, a competitor did something that changed how people phrase their questions. The dataset reflects the world of eight months ago and it will keep reporting excellent scores while the product degrades in the parts of the world it cannot see.

And it is unlabeled. This is the most common of all. Somebody collected a thousand real queries and stopped, because collecting is easy and labeling is not. A thousand queries with no statement of what a correct answer requires is a thousand things you can run and nothing you can score. You will end up reading outputs and forming impressions, which is a demo with extra steps.

Sources of evaluation examples

Cases come from five places, and a healthy dataset draws from all of them in deliberate proportions.

Production traffic gives you reality. It is the only source that tells you what users actually do, and it should be the largest single component of any dataset for a system that has launched. Chapter Six is entirely about mining it, because doing it well involves sampling, privacy, and labeling decisions that deserve their own treatment.

Support and escalation records give you the failures reality has already produced. Every ticket a human had to resolve is a case your system did not close, and every one of those is a test case with the answer already written by the agent who fixed it. This is the highest-quality source most companies have and the one they most reliably forget they own.

Domain experts give you the cases that matter and have not happened yet. Sit a policy specialist down and ask them what would worry them if a machine answered questions in their area. They will produce twenty scenarios in an hour, and those twenty will be sharper than two hundred sampled queries, because the expert is selecting for consequence rather than frequency.

Synthetic generation gives you coverage. It is fast, it is cheap, and it fills the holes the other sources leave, particularly in the long tail of rare intents and adversarial phrasings. It also produces confident nonsense if you let it, which is why Chapter Seven spends as much time on validating synthetic cases as on generating them.

And incidents give you the cases you cannot afford to fail twice. Every production failure that got escalated, argued about, and fixed should end its life as a permanent test case. This is the regression suite in the classical sense, and it is the one part of the dataset that never gets pruned.

Real user queries vs. synthetic queries

There is a temptation to build a dataset entirely out of synthetic cases, because you can produce ten thousand of them before lunch, and there is an opposing temptation to insist on real traffic only, because synthetic data feels like cheating. Both positions are wrong and it is worth being precise about why.

Real queries have distributional truth. They tell you what people ask and in what proportions, which means a score computed over them estimates the experience of an actual user. They are messy in ways nobody would invent: a question that is three words long with two of them misspelled, a question that is four sentences of context followed by no question at all, a question typed by someone who is angry. That mess is the product.

Real queries are also badly distributed for testing. The head of the distribution is enormous and boring. If you sample a thousand conversations at random you will get eight hundred variations on a handful of common intents and almost nothing from the tail where the interesting failures live. Sampling uniformly from production is an efficient way to measure the thing you already know and learn nothing.

Synthetic queries have coverage without truth. You can generate every combination of intent, phrasing, and edge condition you can specify. You cannot generate the ones you did not think of, which is the same limitation as writing them by hand, only faster. And synthetic queries drift toward the register of the model that generated them: grammatical, complete, politely phrased, and unlike anything a real person types into a box on their phone while walking.

So combine them, and be explicit about the split. My default is a stratified sample from production for the score you report, plus a synthetic and expert-authored suite for the failures

you want to catch before they reach production. The first tells you how the product is doing. The second tells you what is about to go wrong. Reporting them as a single blended number destroys both signals.

Happy paths, edge cases, and adversarial cases

Every dataset should be stratified along a difficulty axis, and the strata should be labeled, because the moment they are blended you lose the ability to see a regression that is confined to one of them.

Happy paths are the cases the product exists to handle. They should pass at a very high rate and they should be boring, and their job in your suite is to catch catastrophes: a deployment that broke retrieval entirely, a prompt change that turned the assistant into something unrecognizable. If your happy-path pass rate drops from ninety-eight to eighty, you do not need a rubric to know something is on fire.

Edge cases are where the product is unclear about what it should do. The user asks about a product that was discontinued. The user asks a question that spans two policies which contradict each other. The user asks in a language you support with a term you do not. These are the cases where the correct answer is a design decision, and writing the test case forces you to make the decision, which is a large part of the value.

Adversarial cases are the ones designed to break something. Prompt injection embedded in a retrieved document. A user who insists, politely and repeatedly, that the assistant confirm something false. A request that is harmless in isolation and harmful in the context of the previous three turns. These should fail sometimes, and when they stop failing entirely you should be suspicious that your adversarial cases have gone stale rather than proud that your defenses are perfect.

Track the pass rate on each stratum separately, forever. A blended number that goes from ninety-one to ninety can hide a collapse in the adversarial stratum from seventy to forty, and the adversarial stratum is the one that will end up in a news article.

Dataset size: starting small but meaningful

The question I get asked most often is how many cases, and the honest answer is fewer than you think for the first version and more than you want for the mature one.

Start with fifty. Fifty cases, hand-labeled, covering your top intents and a deliberate scattering of edge and adversarial conditions, will find more real problems in a week than a thousand unlabeled ones will find in a quarter. Jakob Nielsen and Thomas Landauer made a version of this argument for usability testing in 1993, showing that a small number of participants surfaces the large majority of discoverable problems, and while the mathematics does not transfer directly, the intuition does: the first few cases are worth enormously more than the last few, because the failures they find are the common ones.

The statistical constraint arrives when you want to detect a change rather than find a bug. If your pass rate is eighty percent and you want to reliably detect a five-point drop, fifty cases will not do it, because the noise in a fifty-case sample is larger than the effect you are trying to see. The rough arithmetic: the standard error of a proportion around eighty percent with fifty samples is about five and a half points, so a five-point drop is inside the noise. Push to five hundred cases and the standard error falls to under two points, and now the drop is visible. Push to two thousand and you can see the two-point drops that matter for a mature system.

This gives a natural growth path. Fifty to find the obvious failures. Five hundred to gate a release. A few thousand, stratified, to run continuously and detect drift. And a smaller expensive suite of a few dozen cases scored by human experts, run less often, to check that the automated scores still mean what you think they mean.

Cost enters here and it will shape your architecture. Every case costs a model call, and if you score with a judge model it costs two or three. A two-thousand-case suite is real money per run, which means you will not run it on every commit, which means you need a tiered strategy: a fast suite on every pull request, the full suite nightly and before every release. Design for that from the start.

A sizing rule of thumb that has held up across several domains, offered with the caveat that it is a heuristic rather than a result.

Aim for at least twenty cases per intent you care about, and at least fifty per intent that carries real consequence. Below twenty, the pass rate on that intent is too noisy to act on, and you will find yourself investigating swings that are pure sampling variation. Above a hundred for a low-risk intent, you are spending labeling budget on precision you will never use.

Then check the arithmetic against your intent list, because it will produce a number that is larger than the dataset anybody proposed. That gap is the real cost of measuring your product, and knowing it early is better than discovering it in month four when the suite turns out to be too small to gate anything.

Metadata every test case should include

A test case is not a query. A query is a string. A test case is a query plus everything needed to score the response, and the difference between those two things is the difference between a dataset and a text file.

Timnit Gebru and her coauthors argued in Datasheets for Datasets, published in 2018, that every dataset should ship with documentation of its provenance, composition, and intended use. The argument was made for training data and it applies with more force to evaluation data, because an evaluation dataset whose provenance is unknown is a measuring instrument of unknown calibration, and the numbers it produces will be quoted to executives.

The schema below is where I start. Adapt the fields, keep the discipline.

Field	Purpose	Example value
case_id	Stable identifier; survives rewording	eval-2026-0417
query	The user input, verbatim, including errors	does the second one ship free
conversation_history	Prior turns, in full, as the model would see them	[turn objects]
intent	What the user was trying to accomplish	compare_shipping_terms
expected_behavior	What a correct response must do	Resolve 'second one'; state shipping cost
required_facts	Claims that must appear, by meaning	Item 2 ships free above the threshold
disallowed_behavior	What a correct response must not do	Do not quote a delivery date
required_evidence	Sources the answer must be grounded in	policy-doc-shipping-v7, catalog:item2
rubric_id	Which scoring criteria apply	rubric-comparison-v3
severity	Cost of failing this case	high
category	Stratum: happy, edge, adversarial	edge
source	Where the case came from	production-sample-2026-03
owner	Who resolves disputes about this case	policy-team
created	When it entered the suite	2026-03-11
last_reviewed	When a human last confirmed it is still right	2026-06-02
status	active, quarantined, retired	active

Two fields there earn their keep in ways that are not obvious. The last_reviewed date lets you find the cases that have quietly gone wrong, because a policy changed and the expected behavior recorded in your dataset is now the incorrect behavior, and your suite is now penalizing the system for being right. That happens constantly and it is invisible without a date. And the owner field prevents the argument. When a case fails and two engineers disagree about whether the response was acceptable, the case has an owner, and the owner decides, and the decision is recorded, and you move on.

A test of whether your dataset is any good, which takes ten minutes and which most datasets fail.

Hand it to somebody who does not work on the product and ask them to answer twenty of the cases themselves, using only the corpus and the expected-behavior specifications. If they cannot determine what a correct answer requires, from what you wrote down, then neither can a reviewer, and neither can a judge model, and every score you have computed against those cases is noise dressed as a number.

The cases that fail this test are rarely bad cases. They are underspecified cases, and the fix is to write down what you meant. The exercise is uncomfortable precisely because it exposes how much of your dataset's meaning has been living in the heads of the three people who built it.

Versioning evaluation datasets

The dataset is a measuring instrument. Changing it changes what the numbers mean, and a number whose meaning changed silently is worse than no number.

So version it, the way you version code. Every change gets a commit. Every score reports the dataset version it was computed against. When somebody presents a chart showing quality climbing from seventy-two to eighty-one over six months, the first question anyone should ask is which version of the dataset each point was computed on, and if the answer is that the dataset changed nine times, the chart means nothing.

In practice, run two tracks. A frozen track, which changes rarely and only by explicit decision, and against which you report the trend line that goes on the leadership slide. And a living track, which absorbs new cases continuously as production teaches you about new failures, and which you use for development. Merge the living track into the frozen track on a schedule, publish the merge, and re-baseline the trend line at the merge point so the discontinuity is visible rather than hidden.

Store the dataset with the code. It is source. It goes through pull requests, it gets reviewed, and a change to an expected_behavior field gets the same scrutiny as a change to a threshold in production, because it is the same kind of change.

A practical note on ownership, because a dataset with no owner has a predictable life cycle.

In month one it is loved. In month four it is out of date and everybody knows it. In month seven somebody proposes rebuilding it, and the rebuild is scoped as a project, and the project is deprioritized. In month twelve the team is running a suite whose cases describe a product that no longer exists, and reporting the resulting number to leadership, and nobody has done anything wrong at any point.

The fix is that dataset maintenance is a recurring cost rather than a project. Budget a small permanent flow, a few hours a week, and it never becomes a rebuild. This is unglamorous and it is the single most common reason evaluation programs quietly stop meaning anything.

Avoiding overfitting to test cases

The last problem is the one that will eventually get you, and it gets everyone, and there is no complete defense.

You have a suite. It fails on eleven cases. You look at the eleven, you find the pattern, you add an instruction to the prompt addressing the pattern, and now it fails on three. Quality improved. Did it. Or did you teach the system to handle these eleven specific cases in a way that generalizes to nothing?

This is the reason machine learning has held-out test sets, and it is the reason those held-out sets stop working once people start looking at them. Cynthia Dwork and colleagues published work in Science in 2015 on what they called the reusable holdout, and the underlying problem they were addressing is exactly this: every time you make a decision based on the test set, you leak a little information from it into your system, and after enough decisions the test set is no longer measuring generalization. Avrim Blum and Moritz Hardt made a related argument about competition leaderboards in the same period, showing how repeated submissions against a public score let you climb a leaderboard by fitting noise.

Three defenses, none complete. Keep a holdout you never look at, scored quarterly, and treat any gap between it and your development suite as the size of your overfitting. Refresh continuously from production, so a system tuned to last quarter's cases faces this quarter's. And

write the fix at the level of the failure class rather than the case: when eleven cases fail because the system does not resolve ordinal references, the fix is to handle ordinal references, and the way you check that you actually did is to generate thirty new ordinal-reference cases that were nowhere near your suite and see whether they pass.

If your suite scores are climbing and your production complaint rate is flat, you are overfitting. That divergence is the alarm, and it is the reason Chapter Twenty insists that offline and online metrics get looked at together, by the same people, in the same meeting.

Now, one specific piece of dataset construction deserves its own chapter, because more teams get it wrong than get it right. The question is what you write in the box marked correct answer, and the intuitive thing to write there is a paragraph of ideal prose. That intuition will cost you.

Chapter 5: Creating Golden Answers Without Overfitting

A reviewer marked a response as a failure. I read the response. It was excellent.

The customer had asked whether a warranty covered accidental damage. The assistant said no, explained that the standard warranty covered manufacturing defects only, mentioned that an extended protection plan was available and did cover accidental damage, and noted that the plan had to be purchased within thirty days of the original order. Accurate, complete, useful, appropriately brief.

The golden answer in our dataset read: The standard warranty does not cover accidental damage. It covers defects in materials and workmanship for one year from the date of purchase.

The scorer had computed a similarity between the response and the golden answer, found it below threshold, and marked it a failure. The assistant had been penalized for being more helpful than the reference.

This is the golden answer trap, and every team walks into it, because writing an ideal answer feels like exactly the right thing to do. It is intuitive, it is concrete, it feels rigorous, and it quietly encodes the assumption that there is one correct response, which for most of the questions your product receives is false.

What a golden answer is

A golden answer, sometimes called a reference answer or a gold standard, is a written example of a correct response to a test case, produced by a human expert, stored alongside the case, and used as the target against which the system's output is compared.

It has a long and respectable history. Machine translation used reference translations. Summarization used reference summaries. Question answering used reference answers, usually short. In each of those settings the space of good outputs was narrow enough that a reference was a reasonable stand-in for the whole space, and the automatic metrics built on top of references, BLEU and ROUGE and their descendants, correlated well enough with human judgment to be useful for research progress.

The word golden does a lot of quiet damage. It suggests the answer is the standard, and that deviation is deficiency. For a translation of a single sentence, close to true. For an answer to a customer asking whether a jacket will keep them dry in Seattle in November, badly false. There are dozens of good answers, differing in length, structure, hedging, and what they choose to mention, and the one your expert happened to write is one sample from that space rather than the center of it.

When golden answers work

They work when the answer space is narrow, and narrowness is a property you can check.

Extraction is narrow. What is the order number in this email. There is one right answer, it is a string, and exact match is the correct metric. Any evaluation that scores this with a rubric is wasting money.

Closed classification is narrow. Is this request about billing, shipping, or returns. One of three labels. Exact match again.

Structured generation is narrow. Produce a JSON object with these five fields, populated from this document. The structure is checkable, the field values are checkable, and a golden object is

exactly the right artifact to compare against.

Short factual answers are narrow enough. How many days is the return window. Thirty. If the model says thirty days, or 30, or a month, you handle that with a small amount of normalization and you are done.

And tool calls are narrow, which is a point I keep returning to because it is the best news in agentic evaluation. The model emits a function name and arguments. That is a structured object. A golden object compares against it precisely, and no judgment is involved.

The general rule: if you can state the correct answer in a form that admits no legitimate variation, use a golden answer and use exact comparison. It is cheap, it is deterministic, it runs in milliseconds, and it will never disagree with itself. Use it everywhere you can, and be honest about where you cannot.

When golden answers fail

They fail whenever the response is prose intended for a human to read, which is most of what your product produces.

They fail on length. The reference is four sentences; the response is two, and says everything the four said. Similarity drops. The response was better.

They fail on ordering. The reference leads with the policy and follows with the exception. The response leads with the exception, because for this customer the exception is the answer. Similarity drops. The response was better.

They fail on legitimate additions. The warranty case above. The system said something true and useful that the reference did not contain, and every overlap-based metric treats extra content as noise.

They fail on legitimate omissions. The reference mentions three facts. Only two are relevant to what this customer asked. A response that mentions two is more focused and scores lower.

And they fail catastrophically on negation, which is the failure that should frighten you. A response identical to the golden answer except for the word not will have a similarity score near one. It is also the exact opposite of correct, and it will sail through your suite. Every overlap metric and most embedding metrics share this weakness, and if your evaluation rests on similarity scoring you have a system that cannot distinguish yes from no.

Hold that last one in mind whenever somebody proposes a cosine similarity threshold as a quality gate. Ask them what score their metric gives to a response with an inverted negation. If they have not checked, they do not know what they are measuring.

Reference answer vs. expected behavior

The alternative is to stop describing the answer and start describing the requirements the answer has to meet.

An expected behavior specification says what a correct response must do, what it must contain, and what it must avoid, without prescribing the words. It is longer to write than a golden answer and it is checkable in a way a golden answer is not, because each element is an independent assertion that a human or a model can verify.

For the warranty case, the specification reads roughly like this. The response must state that accidental damage is excluded from the standard warranty. It must state what the standard warranty does cover. It must mention the extended protection plan as an option, because a

customer asking this question is trying to solve a problem and the plan solves it. It must not state a coverage period without qualifying it, because the period varies by category. It must not promise that a specific claim will be approved. It may mention the thirty-day purchase window for the plan, and mentioning it is a small bonus rather than a requirement.

That specification would have passed the response my reviewer failed. It also would have failed a response that inverted the negation, that promised approval, or that quoted a coverage period as universal. It discriminates in the right directions, which is the only property that matters.

The cost is real. A specification takes ten to fifteen minutes of expert time to write, against three for a golden answer, and it has to be maintained when the policy changes. That is the price of a measuring instrument that measures the right thing, and it is the reason I argue for fifty carefully specified cases over a thousand casually referenced ones.

Acceptable answer variation

Once you accept that many answers are correct, you have to decide how much variation you are willing to accept, and that decision is a product decision rather than an evaluation one.

Some variation is free. Wording, sentence order, whether the response opens with a direct answer or a one-line acknowledgment. Nobody cares and your specification should not.

Some variation is constrained by the product. Length, when the interface has a display limit. Register, when the brand has a voice. Structure, when the downstream renderer expects a particular shape. These belong in the specification as explicit requirements, because they are requirements.

And some variation is a bug wearing the costume of variation. If the same question, asked twice, produces answers that differ on a material fact, the system is unstable in a way customers will notice, because customers ask friends and friends compare. Self-consistency is a measurable property. Run each case several times, score each run, and report the variance alongside the mean. A case with a mean score of four and a variance of zero is a different animal from a case with a mean of four and a spread from one to five, and only the first one is under control.

I would go further and say the variance number deserves a place on your dashboard. Teams optimize the mean because the mean is what they report. A system whose mean is good and whose variance is enormous is a system that fails a large minority of the time, and the people it fails are not comforted by the average.

A failure of specification that I have seen produce weeks of wasted argument, and that is prevented by one sentence in the template.

Distinguish what the response must contain from what it may contain. A specification that lists eight required elements, when only three are genuinely required and five would be nice, will fail every response that omits any of the five, and reviewers will start ignoring the failures, and once reviewers start ignoring some failures they stop reading carefully at all.

So mark them. Required elements are scored. Optional elements are recorded and not scored, and the record is still useful, because a response that consistently includes the optional material is telling you something about how the system allocates its budget. What you must not do is let a nice-to-have quietly become a criterion, which is how a rubric becomes unpassable and then becomes irrelevant.

Rubric-based comparison

The mechanism that turns an expected behavior specification into a number is a rubric, and Chapter Eight builds them properly. Here I want to establish the shape, because it changes how the reference is used.

In rubric-based comparison, the reference is not the answer. The reference is evidence. It is available to the scorer as an example of a good response, and it informs judgment, and it does not define the target. The scorer, human or model, is asked the questions from the specification: did the response state the exclusion, did it avoid promising approval, did it mention the plan. Each is answered independently. The score is a function of the answers.

This decomposition is what makes disagreement productive. When two reviewers give a response different overall scores, you have an argument. When they give it different answers to did the response state the exclusion, you have a specific factual disagreement about a specific sentence, and it will be resolved in under a minute, and the resolution will improve the specification for everyone who uses it afterward.

It also gives you diagnostics for free. Aggregate the per-criterion results across your suite and you learn that the system fails the mention-the-alternative criterion forty percent of the time while passing everything else. That is a prompt fix, it takes an afternoon, and you would never have found it from a blended score.

Multiple valid outputs

Sometimes there is more than one right answer, and the right answers are genuinely different, and this deserves explicit handling rather than a fudge.

A customer asks for a recommendation between two products with real tradeoffs. Recommending either one, with the tradeoff stated honestly, is correct. Recommending neither and laying out the choice is also correct. Refusing to engage is not.

Encode this as a disjunction in the specification. The response must do at least one of the following, and whichever it does, it must also do the following. That is a checkable structure and it takes the ambiguity out of the reviewer's hands, which is where ambiguity turns into noise.

Pairwise comparison is the other tool for this situation, and it is often better. Rather than scoring one response in isolation, present two and ask which is better. Humans are much more reliable at relative judgment than absolute judgment, and the LLM-as-judge literature has found the same thing: Lianmin Zheng and colleagues, in the MT-Bench and Chatbot Arena work published in 2023, built their whole evaluation around pairwise preference for exactly this reason. The tradeoff is that pairwise tells you which is better and never tells you whether either is good enough, so it is the right tool for comparing a candidate against production and the wrong tool for a release gate with an absolute threshold.

One more case where the golden answer is the right tool, and it is the one people forget.

Refusals. When the correct behavior is to decline, the expected response is narrow: decline, state the reason, offer the path forward. That is close enough to a fixed shape that you can specify it precisely, and specifying it precisely is what lets you detect the failure mode where the system starts declining more than it should. An over-refusal is invisible to any evaluation that only checks whether unsafe content got through, and a tightly specified refusal case is what makes it visible.

The same applies to any response whose job is structural rather than informational: a clarifying question, an escalation, an acknowledgment. These are short, their shape is constrained, and they are the responses most likely to drift when somebody changes a persona instruction.

Maintaining golden answers over time

Every reference in your dataset is a claim about the world, and the world moves.

The policy changed in March. Forty of your expected-behavior specifications now describe the old policy. Your suite is now penalizing the system for stating the current policy correctly, and rewarding it for being wrong, and the score is going down, and somebody is about to spend a sprint fixing a system that is working.

This will happen. Plan for it structurally rather than hoping to catch it. Link each case to the source documents its expected behavior depends on, so that when a document changes you get a list of the cases that need review. Put a last_reviewed date on every case and quarantine anything older than your policy refresh cycle. And when a case fails, make the first question in the triage flow whether the case is still right, before anyone touches the system.

I would also build the reverse check, because it catches the sneakiest version. Periodically sample cases that are passing and have a human confirm that passing is still the right outcome. A case that is passing for the wrong reason is invisible to every process you have, and there is always at least one.

The template below is what a case looks like when you have made all the decisions in this chapter. Compare the left column to the right and note how much more the right column can tell you when it fails.

	Golden answer approach	**Expected behavior approach**
Artifact	One paragraph of ideal prose	A list of requirements and prohibitions
Comparison	Similarity score against the reference	Independent check of each requirement
Handles rewording	Poorly; penalizes valid variation	Yes; wording is not specified
Handles extra content	Penalizes it	Neutral unless prohibited
Handles omission	Penalizes any omission equally	Only required elements count
Handles negation flips	Fails; similarity stays high	Catches it; the requirement is unmet
Diagnostic on failure	A number went down	The specific requirement that was missed
Cost to author	Low	Moderate; expert time per case
Cost to maintain	Rewrite the whole reference	Edit the affected requirement
Right for	Extraction, classification, tool calls, short facts	Open-ended prose answers to real users

Use golden answers where the answer space is narrow. Use expected behavior everywhere else. And when somebody proposes a similarity threshold as a quality gate, ask them the negation question, and watch what happens.

All of this assumes you have cases to specify. The best source of them has been sitting in your logs the entire time, and getting at it involves a set of problems that are more legal than technical.

Chapter 6: Mining Production Traffic for Evaluation

Eleven months after launch, we pulled a random sample of two thousand conversations and read them.

Reading logs is not glamorous work and I recommend doing it personally, at least once, rather than delegating it, because the experience of watching two thousand strangers try to use a thing you built is the fastest way to understand what you built. About a quarter of the way through, I stopped taking notes on individual conversations and started taking notes on categories, because the same shapes kept appearing.

There was a category of user who typed in fragments, three or four words, no verb, expecting the assistant to reconstruct the question. There was a category who wrote paragraphs, apologetically, as if talking to a person who might be annoyed. There was a category who tested it, asking it who it was and whether it was human and whether it could tell them a joke, and then, having satisfied themselves, asked a real question. There was a category who asked something reasonable, received an answer that was subtly wrong, did not notice, said thank you, and left.

That last category is the one I could not stop thinking about. They were satisfied. Our satisfaction metric counted them as successes. They had been given bad information and they had thanked us for it.

None of these people appeared in our evaluation dataset, because our evaluation dataset had been written by seven engineers who all use computers professionally and none of whom have ever typed three words into a search box and expected to be understood. Production traffic is the only source that will correct that, and this chapter is about extracting a dataset from it without either drowning or breaking the law.

Sampling production conversations

A uniform random sample is the wrong first instinct, and it is the one everybody has.

The problem is the shape of the distribution. Traffic to a production assistant follows a steep power law. A small number of intents account for the majority of requests, and those intents are the ones your system was designed around and handles well. Sample uniformly and you will spend most of your labeling budget confirming that the system does the thing it was built to do. You will learn your pass rate on the head of the distribution, which was high, and you will learn almost nothing about the tail, where the failures are.

Stratify instead. Partition traffic by the dimensions that matter for your product, then sample within each stratum at a rate you choose rather than a rate the traffic imposes. Intent is the obvious axis. Conversation length is the one most teams miss, and it matters enormously, because multi-turn behavior is where the interesting failures live and long conversations are a minority of traffic. Risk category matters, because a wrong answer about a payment carries a different cost from a wrong answer about a color. New versus returning users matter, because their phrasing differs. Locale matters. And the presence of a negative signal matters most of all.

Oversample the tail deliberately and record the sampling weights, so that when you want a number that estimates the true production experience you can reweight back to the traffic distribution. Report both. The unweighted number tells you how you do on hard cases; the reweighted number tells you what a random user experiences. Teams that report only one of these are misleading somebody, usually themselves.

Identifying high-risk user intents

Risk is not frequency, and evaluation budgets allocated by frequency will underspend on precisely the cases that can end a product.

Rank your intents by the cost of getting them wrong. A wrong answer about whether a medication interacts with another medication is unrecoverable. A wrong answer about a shipping estimate is annoying. A wrong answer about a return policy sits between them and can produce a chargeback. The ordering is domain-specific and it takes an afternoon with a policy person, a legal person, and somebody from support to produce, and once you have it, everything downstream gets easier.

Then allocate coverage against risk rather than volume. If a high-risk intent is one percent of traffic, it should still get a large block of your evaluation cases, because your exposure there is not proportional to its frequency. This will feel wasteful when the pass rates come back high. It is insurance, and insurance always feels wasteful until it does not.

The specific list is worth writing down and putting where the team can see it, because engineers under deadline will optimize whatever is measured, and if the high-risk intents are buried inside an aggregate score, nobody will notice when they start failing.

Finding repeated failure patterns

Individual failures are anecdotes. Clusters of failures are bugs, and the work is turning the first into the second.

The technique that has worked best for me is embarrassingly simple. Take every conversation with a negative signal, embed the user's turn, cluster the embeddings, and read one example from each cluster. Twenty minutes of that will tell you more about what is broken than a month of dashboard-watching, because the clusters are the failure classes and you did not have to guess at them.

What comes out is usually surprising and usually not what the team was working on. A cluster of users asking about a feature that was removed and about which nothing in the corpus says it was removed, so the assistant describes it as if it exists. A cluster of ordinal references that fail. A cluster of users asking in a phrasing that trips an intent classifier into a route with no retrieval. Each of those is a fix, and each of those turns into a family of test cases.

Size the clusters and rank them. A cluster with four hundred instances gets fixed before one with six, unless the six are high severity, in which case they get fixed first. This is triage, it is the same triage you already do for bugs, and the only new part is that you had to build the clustering because nothing was reporting these as bugs.

Capturing escalations and negative feedback

The signals are ranked by how much they cost the user to produce, which is a decent proxy for how much they mean.

A human escalation is the most expensive signal a user can send and the most valuable one you will receive. They tried the assistant, it failed, and they went to the trouble of finding a person. Every escalation is a test case with a known correct outcome, because the human who resolved it wrote the resolution down. If your assistant and your support system are separate products with separate teams and no pipeline between them, that pipeline is the highest-return thing you can build this quarter.

A user correction is next. When someone types no, I meant, or that is wrong, or actually, they are telling you something failed and telling you what. These are trivially detectable with a small classifier and they are gold, because the correction contains the ground truth.

Abandonment is common and ambiguous. The user left. Maybe they got what they needed in the first sentence. Maybe they gave up. Abandonment after a long answer to a short question means something different from abandonment after a clarifying question, and you can only tell by reading a few.

Thumbs down is the signal everybody instruments first and it is the weakest of the four. The response rate is low, the population who click is unrepresentative, and the reason is unknown, because a thumbs down means the response was wrong or the response was slow or the product does not do what the user wanted or the user is having a bad day. It is a starting point for a sample, and it is not a metric.

And there is a class of failure with no signal at all: the user who was given a wrong answer, believed it, and left happy. Nothing in your telemetry will find them. The only way to see them is to sample conversations that produced no negative signal and read them, which is expensive and which you should do anyway, at a low rate, forever.

Stage	What happens	Failure if skipped
Signal capture	Thumbs down, user correction, repeated rephrase, abandoned session, human escalation, safety intervention, low-confidence retrieval, tool failure, latency outlier	You only learn about failures a customer bothered to report
Sample	Stratified by intent, risk tier, and signal; oversample the rare and the consequential	The dataset mirrors your traffic mix and misses the cases that matter
Redact	Personal data removed at ingestion; entities substituted; redaction itself under test	The evaluation dataset becomes a compliance liability
Label	Intent, expected behavior, required facts, disallowed behavior, severity	An unscoreable case that will be argued about forever
Review	Expert pass; genuinely ambiguous cases discarded rather than adjudicated	Undecidable cases add permanent noise to every run
Promote	Versioned case written to the registry with its source signal and review date	A score nobody can reproduce in six months
Route	Regression suite, failure taxonomy, dataset refresh queue	Cases accumulate and nothing changes

One category of production signal that is worth instrumenting deliberately, because it is the only one that captures a failure the user did not experience as a failure.

The silent correction. A user asks a question, gets an answer, and then, without complaint, goes and does something that contradicts the answer they were given. They asked about the return window, were told thirty days, and then opened a support ticket about a return on day forty. They asked whether a product was compatible, were told yes, and then returned it.

These are the failures with no signal, and they are visible only where the assistant's answer can be compared against what the person subsequently did. Most products have that data and almost

none of them join it. The join is a query. What comes out is a list of the times your system was wrong and the customer never told you.

Privacy and data minimization

Everything above assumes you are allowed to read your users' conversations, and the honest answer is that you are allowed to read some of them, for some purposes, under some conditions, and you should find out exactly which before you start.

The principle that governs this in most regulatory regimes is data minimization: collect what you need for the stated purpose, keep it as long as the purpose requires, and no longer. It is written into the General Data Protection Regulation as one of the core processing principles, and variants of it appear in essentially every modern privacy framework. For an evaluation program the practical translation is that you are not entitled to a permanent copy of every conversation because you might want it someday.

So decide the purpose first and write it down. Evaluation of assistant quality is a purpose. Improvement of the product is a purpose. Both are usually defensible, and both are narrower than they sound, and the narrowness is what protects you.

Then get the retention right. Raw conversation logs should have a short life, measured in weeks. Evaluation cases derived from them, redacted and abstracted, can live longer because they no longer contain the person. That transformation, from log to case, is the thing that makes the whole program sustainable, and it is why the pipeline below puts redaction before anything else touches the data.

Get your privacy and legal partners involved at design time. I have watched a team build a beautiful evaluation platform and then discover it could not be used in three of their markets. That conversation is much cheaper before the platform exists.

Anonymization and redaction

Redaction is harder than it looks and the failure mode is quiet, so treat it as an engineering problem with its own tests.

The obvious identifiers come out with pattern matching: emails, phone numbers, card numbers, account identifiers, addresses. A named entity recognizer catches most of the rest. Both will miss things, because users put personal information in places nobody anticipated, in the middle of a sentence, spelled out, in another language.

And removing the direct identifiers is not the end of it. Latanya Sweeney demonstrated in her work on k-anonymity, published in 2002, that a small number of quasi-identifiers can re-identify an individual in a supposedly anonymous dataset, and a conversation is full of quasi-identifiers. A user who mentions their profession, their city, and the model of a rare product they own has told you who they are without giving you their name.

Two practices help. Substitute rather than delete, replacing a name with a different name rather than a black box, because a redacted conversation full of gaps is unreadable and the person labeling it will make things up to fill them. And abstract aggressively when you promote a log into a test case: the case does not need the user's actual city, it needs a city, and swapping one for another costs nothing and removes the exposure.

Test your redaction the way you test anything else. Plant synthetic personal information in synthetic conversations, run them through the pipeline, and assert that it came out. If your

redaction has no test suite, you do not know whether it works, and one day somebody will paste a log into a ticket and find out.

A sampling detail with a large effect on what you learn, and it is easy to get backward.

Sample conversations, not turns. A turn pulled out of a conversation has lost the thing that made it hard, which is everything that came before it. You will label it, score it, and find that the system handled it perfectly, because in isolation it was an easy question, and the reason it failed in production was the six turns of context you did not sample.

The cost is that a conversation is more expensive to label than a turn, sometimes much more, and the temptation to sample turns is a budget temptation. Resist it for anything multi-turn. A hundred labeled conversations will teach you more than a thousand labeled turns, and the thousand turns will actively mislead you about where your product stands.

Converting real traffic into test cases

A sampled, redacted conversation is raw material. Turning it into a test case takes a human, and the work is the labeling that Chapter Four's schema demands.

Read the conversation and record what the user was trying to do, which is often different from what they typed. Record what a correct response would have had to contain, and what it would have had to avoid. Record the evidence that a correct response would need, and check that the evidence exists in your corpus, because if it does not, you have found a content gap rather than a system failure and it belongs to a different team. Assign severity. Assign a stratum. Assign an owner.

Discard aggressively. Perhaps a third of sampled conversations are unusable as test cases: the user was testing the system, the intent is genuinely unrecoverable, the conversation depends on a state you cannot reconstruct, two reviewers disagree about what correct even means. Throwing those away is correct. A case whose expected behavior is disputed will produce disputed scores forever, and every run of your suite will re-litigate it.

Budget realistically. A trained reviewer produces somewhere between fifteen and thirty finished cases in a day, depending on domain complexity. Five hundred cases is therefore three to five person-weeks of expert time, and that number is the actual cost of an evaluation dataset. Teams that have not done this arithmetic tend to promise a dataset in a sprint and deliver a spreadsheet.

A word about who reads the logs, because the answer determines what you learn from them.

The engineers who built the system are the worst possible readers of its transcripts, and I say that as one of them. They know what the product is supposed to do, so they unconsciously supply the missing intent when a user's message is ambiguous. They know why the system asked a clarifying question, so the clarifying question reads as reasonable rather than as an irritation. They skim past the failures that are familiar and linger on the ones that are novel, which is exactly backward, because the familiar failures are the ones happening ten thousand times a day.

So bring in someone else. A support agent, a designer, a person from the policy team, somebody who joined last month. Give them a hundred conversations and a highlighter. What they mark will be different from what your team would mark, and the difference is the measure of how much your team has stopped being able to see the product.

Refreshing datasets after launch

A dataset built once is a snapshot of a world that has already changed.

Traffic shifts for reasons that have nothing to do with you. A campaign brings in a new cohort. A competitor launches something and users start asking comparative questions you have never seen. A season turns. A policy changes and the questions about it triple. Each of these makes your dataset less representative, and the score it produces less meaningful, and none of them announce themselves.

So run the pipeline continuously rather than as a project. A modest, permanent flow: sample this week's traffic, cluster the failures, promote the new failure classes, retire the cases whose policies have changed, and re-baseline on a schedule. The volume can be small. The continuity is what matters.

Watch for distribution drift as a metric in its own right. Compare the intent distribution in your dataset against the intent distribution in last month's traffic and report the divergence. When it grows past a threshold, the dataset needs work, and reporting that number keeps the work from being deprioritized indefinitely, because it turns a vague sense of staleness into a line on a chart that is going the wrong way.

And keep the incident cases forever. Everything else in the dataset can churn. The cases that came from real production failures, the ones that cost you an incident review and an apology, stay in the suite until the feature is deleted. They are the cheapest insurance you will ever buy against making the same mistake twice.

Production traffic will teach you about the failures you have already had. It cannot teach you about the ones you have not had yet, and some of those are the ones that will hurt most. For those, you have to invent the cases before reality provides them.

Chapter 7: Synthetic Test Generation

The engineer had generated forty thousand test cases overnight and was, understandably, pleased about it.

We looked at a sample together. They were grammatical. They were varied in surface form. They covered every intent in the taxonomy. They were also, every single one of them, written in the voice of a language model imitating a customer, which is to say polite, complete, correctly spelled, and structured as a full sentence with a clear question at the end.

Real customers do not write like that. Real customers write does this fit a queen bed, with no question mark, having already forgotten which product they were looking at. The forty thousand cases measured how well our system handled a population of users that does not exist.

That is the central hazard of synthetic data and I want to put it up front, before the techniques, because the techniques work and the hazard is what makes them dangerous. A synthetic dataset gives you the comforting sensation of coverage. Whether it gives you coverage of anything real depends entirely on the discipline you apply after generation, and most teams stop before that part.

Why use synthetic evaluation data

With that said, there are four situations where synthetic generation is the correct answer and no other approach works.

Pre-launch, you have no production traffic, because the product does not exist. You have a corpus, a policy, a set of intents, and an ambition. Synthetic cases are the only cases available, and a synthetic suite of a few hundred cases before launch is what stands between you and finding out about your failures from a customer.

In the tail, real traffic is too thin to measure. An intent that occurs in one request in ten thousand will produce a handful of examples in a month of logs, which is not enough to detect a regression. Generate a hundred variations of it and now you can.

For adversarial coverage, waiting for reality is a strategy with an unacceptable failure mode. You do not want to discover your prompt injection vulnerability by having it exploited. You want to generate a thousand injection attempts and find out today.

And for combinatorial coverage, hand-authoring is impossible. Six intents by four evidence conditions by three conversation depths by two locales is a hundred and forty-four combinations, and a human will write twenty of them and get bored. A generator will write all of them, and the ones nobody would have thought to write are frequently the ones that break.

Generating intent variations

The first and easiest use is to take a case you have and produce the variations of it that a real population would produce.

The transformations that matter are the ones that model actual human behavior. Length, from a terse fragment to a rambling paragraph. Formality, from a full sentence to a search query. Errors, meaning typos, missing punctuation, and autocorrect damage. Indirection, where the user describes their situation and never asks the question. Presupposition, where the question assumes something false and a good answer has to correct the assumption before answering. Emotion, because an angry user phrases things differently and your system should hold up.

Do this with explicit instructions per transformation rather than asking for variations generally, because a generic request for variety produces surface paraphrase and nothing else. Ask for the same question typed by someone in a hurry on a phone, and you get something usable. Ask for ten variations, and you get ten grammatical sentences that differ in word choice.

The technique is a cousin of the instruction-generation work in the research literature. Yizhong Wang and colleagues, in the Self-Instruct paper published in 2023, bootstrapped a large instruction set from a small seed set using a model, and Can Xu and colleagues extended the idea in the Evol-Instruct work by applying explicit evolution operators that make an instruction harder along a named dimension. That second idea is the one worth stealing. Naming the dimension along which you want variation is what separates useful generation from paraphrase.

Then check the invariant. A variation of a case is only a case if the expected behavior is unchanged. If your transformation altered the meaning, you have generated a new case with an unknown answer, and shipping it into your suite with the old expected behavior attached will produce a false failure that somebody will spend a day chasing.

Generating multi-turn conversations

This is where synthetic generation earns its keep, because multi-turn cases are brutally expensive to author by hand and the failures they catch are the ones customers complain about.

Do it by scripting the structure rather than the words. A multi-turn test case is a sequence of user moves, and the moves are drawn from a small vocabulary: establish a constraint, ask a question, refer back with an ordinal, correct the assistant, add a constraint, change the topic, return to the earlier topic. String those together and you have a conversation shape. Fill in the words with a generator and you have a conversation.

The scripted shape is what makes it a test. You know that turn five contains an ordinal reference to something enumerated in turn four, so you know what the assistant must resolve, so you can assert on it. A conversation generated freely has no known answer at any turn and is therefore not a test case at all, merely a transcript.

Run the user side with a model that has been given a persona, a goal, and a script of moves, and let it react to what the assistant actually says. This is a simulation, and simulations have a known weakness: the simulated user is too agreeable. It accepts a bad answer and moves on, where a real user would push back. Instruct it explicitly to be unsatisfied when a response fails to answer, and to press, because the pressing is where the failures surface.

Chapter Ten develops the assertion machinery for these. What matters here is the generation discipline: script the shape, generate the words, know the answer in advance.

Generating adversarial prompts

Adversarial cases are the strongest argument for synthetic generation, because the alternative is a red team of humans, and humans are slow, expensive, and finite.

The idea of using a model to attack a model is well established. Ethan Perez and colleagues at DeepMind published work in 2022 on red-teaming language models with language models, generating test cases automatically and finding harmful behaviors at a scale no human team could reach. Deep Ganguli and colleagues at Anthropic published a large study in the same year on human red teaming and released the transcripts, which remain one of the more useful public resources for anyone building an adversarial suite.

The categories to generate against are stable enough to list. Instruction override, where the input tells the system to disregard its instructions. Indirect injection, where the hostile instruction is embedded in a retrieved document rather than typed by the user, which is the more dangerous version because the user is innocent and the attacker is a web page. Role play, where the harmful request is wrapped in a fiction. Incremental escalation, where each turn is individually fine and the sequence is not. Persistence, where the user simply asks eleven times. Extraction, where the goal is to make the system reveal its system prompt, its evidence, or another user's data. And sycophancy attacks, where the user asserts something false with great confidence and sees whether the system folds.

Generate hundreds of each, and expect a low hit rate, because most attacks fail and that is the point. The value is in the few that succeed, and every one of those becomes a permanent regression case and a fix.

One warning. Indirect injection through retrieved content is the attack surface that most teams have not tested and most teams have. If your system retrieves from anywhere a third party can write, generate the cases today. It takes an afternoon and it is the highest-severity gap I encounter.

Generating policy edge cases

Policies have boundaries, boundaries have both sides, and the interesting cases are the ones that sit exactly on the line.

Take a policy rule and generate cases at the boundary of every condition in it. If the rule is that returns are accepted within thirty days for unopened items, then the cases are thirty days, thirty-one days, twenty-nine days, opened, unopened, partially opened, opened and resealed, unopened but the box is damaged, thirty days but a holiday intervened, thirty days in a jurisdiction with a statutory extension. Half of those are questions your policy does not answer, and discovering that is worth the exercise on its own.

Then generate the interactions, which is where the real failures live. Two policies that apply to the same request and disagree. A promotion that modifies a base policy for a subset of items. A general rule with a category exception that itself has an exception. Combinatorial generation is good at this and humans are bad at it, because humans get bored at the third nesting level and a generator does not.

The output of this exercise is usually a set of test cases and a set of questions for the policy owner, and the questions are more valuable than the cases. When you ask a policy owner what the system should say to a customer returning a resealed item on day thirty-one and they pause, you have found a hole in the product that would otherwise have been filled by a model's improvisation, in production, in front of a customer.

A generation technique that is worth more than any of the others in this chapter and that takes an hour to set up.

Generate from failures rather than from intents. Take a case your system got wrong, hand it to the generator, and ask for thirty variations that preserve the underlying difficulty while changing the surface. Different wording, different entities, different phrasing, same trap. What you get is a family, and a fix that passes one member and fails the other twenty-nine is a fix that did not generalize, which is the thing you most needed to know.

This is the mechanism that turns a single incident into real coverage, and it is the step that Chapter Eighteen makes mandatory in the incident process. A regression suite built one case at a

time from individual incidents is a suite that certifies you have memorized your past failures. A suite built from families certifies you have understood them.

Validating synthetic test quality

Now the part everybody skips, and the part that decides whether the forty thousand cases were an asset or a liability.

Every synthetic case has to clear three checks before it enters the suite. Is it realistic, meaning would a human plausibly produce this input. Is it valid, meaning does the expected behavior attached to it actually describe a correct response. And is it distinct, meaning does it test anything the suite does not already test.

Realism is the check people skip. Sample a hundred generated cases and put them in front of somebody who reads real customer conversations for a living, and ask them to mark the ones that could not have come from a person. If the rate is above one in five, your generator is producing a fictional user population and you need to change the prompt, add real examples as few-shot anchors, or introduce noise deliberately.

Validity is the check that catches the expensive errors. A generated case with a wrong expected behavior is worse than no case, because it will fail forever, and every failure will consume triage time, and eventually somebody will change the system to satisfy it and make the product worse. Human review of a sample is the only reliable defense. Review at least ten percent of generated cases, and review one hundred percent of anything in a high-severity category.

Distinctness is cheap to check and it saves money forever. Embed the cases, cluster, and drop the near-duplicates. A suite of ten thousand cases where four thousand are variations of the same thing costs four thousand model calls per run and tells you nothing extra.

Preventing unrealistic test coverage

There is a deeper failure mode than unrealistic phrasing, and it is a failure of the distribution rather than the individual case.

A generator produces the cases its prompt implies. If your prompt describes six intents, you will get six intents, in roughly equal proportions, none of which resembles the actual traffic distribution and none of which includes the intent you forgot to mention. The dataset will look comprehensive and will have a hole exactly where your imagination had one, which is where the hole always is.

There is also a self-reference problem that the research community has started to take seriously. Ilia Shumailov and colleagues published work in 2023 on what happens when models are trained recursively on model-generated data, describing a degenerative process in which the tails of the distribution disappear and the output collapses toward the mean. The mechanism is about training rather than evaluation, and the analogy holds well enough to be worth a warning: a synthetic evaluation set generated by a model will reflect that model's idea of what varies, which is systematically narrower than reality, and if you then tune your system against that set you will optimize for a shrinking target.

The defense is to anchor generation in reality. Seed every generation run with real examples. Compare the distribution of the generated set against the distribution of real traffic, on whatever axes you can measure, and report the divergence. And never let the synthetic set be the only thing you report, which brings us to the last section.

A caution about generating cases with the same model that serves your product, because it is convenient and it introduces a bias that is very hard to see.

A model asked to produce difficult test cases will produce cases that are difficult in the ways it can imagine, which are not necessarily the ways it fails. Its blind spots are, by definition, the things it does not know it does not handle, and a generator cannot deliberately generate a case that exploits a weakness it cannot represent. The result is a suite that looks adversarial and systematically avoids the failure modes that matter most.

Use a different model family for generation where you can, and mix in human-authored cases specifically to cover the region the generator cannot reach. And treat a suddenly rising pass rate on a synthetic adversarial suite with suspicion rather than satisfaction, because the most likely explanation is that the generator and the system under test have grown more alike.

Combining synthetic and production-derived datasets

Keep them separate. Report them separately. Use them for different things.

The production-derived suite is your estimate of reality. It is what you quote when someone asks how the product is doing, because it is sampled from what users actually do and it can be reweighted to the traffic distribution. It is smaller, it is more expensive, and it moves slowly.

The synthetic suite is your early warning system. It is large, cheap, and heavily weighted toward the difficult and the dangerous. It answers a different question: not how are we doing, but what would break if we were unlucky. Its absolute pass rate is close to meaningless, because it is deliberately hard. Its trend is meaningful, and a drop in it is a signal worth acting on.

Blending them into a single headline number destroys both. The blended score will move when you change the synthetic-to-real ratio, which is a change to your instrument rather than a change to your product, and somebody will eventually notice this and stop trusting the number, and they will be right.

Here is the generation template I use as a starting point. It is written for a policy-edge-case run; adapt the sections for the other categories.

Prompt section	What goes in it
Role	You are generating evaluation test cases for a production assistant. You are not answering questions; you are writing inputs that will be used to test one.
Domain context	The product, the audience, the tone real users take, and the channel they type into.
Grounding examples	Six to ten real, redacted user turns. These anchor register and length. Without them the generator writes essays.
Source material	The exact policy text, corpus excerpt, or catalog entry the cases must be built around. Do not paraphrase it into the prompt.
Generation axis	The single named dimension to vary: boundary conditions of each clause in the policy; or conflicting policies; or nested exceptions.
Constraints	Length range. Register range. Permitted error types. Explicit instruction that inputs must be plausible for a real user on a phone.
Required output fields	query, conversation_history, intent, expected_behavior, required_facts, disallowed_behavior, severity, category, generation_axis.
Expected behavior rule	Derive expected_behavior only from the supplied source material. If the source does not determine the answer, emit the case with status=needs_policy_decision instead of guessing.
Diversity instruction	No two cases may share a phrasing pattern. Vary opening structure across the batch.
Self-check	Before emitting, confirm: a real user could plausibly have typed this; the expected behavior follows from the source; the case is not a near-duplicate of an earlier one in this batch.

Note the expected behavior rule, because it is the one that matters most. A generator asked to produce both a question and its correct answer will always produce both, confidently, even when the source material does not determine the answer. Instructing it to flag the gap instead is how you turn a generator into an instrument that finds holes in your policy rather than papering over them.

You now have a dataset. You have cases from production, cases from experts, cases from a generator, all labeled with expected behavior and severity and a stratum. What you do not yet have is a way to turn a model's response into a number, reliably, at a scale that does not require a human to read every output.

That is scoring, and it is the subject of Part Three.

Part Three

Scoring and Rubrics

Chapter 8: Designing Evaluation Rubrics

We gave the same twenty responses to four reviewers and asked them to score each one from one to five on overall quality. Then we put the scores in a spreadsheet next to each other and looked at them together, which I recommend doing exactly once, as a form of therapy.

On one response, the four scores were five, four, two, and one. Same text. Same reviewers, all of whom had been on the team for over a year and all of whom knew the product well. The person who gave it a five said it was accurate and complete. The person who gave it a one said it was far too long and buried the answer. Both were describing the same paragraph, and both were right about the thing they were looking at, and neither had been told which thing to look at.

We had built a measuring instrument whose readings depended on who was holding it. Every number we had produced in the previous three months was noise with a mean.

The fix had nothing to do with finding better reviewers. It was telling them what to measure.

Why rubrics matter

A rubric is a written set of criteria that converts a judgment into a score, with enough specificity that two people applying it to the same output land in the same place most of the time.

That last clause is the whole game. A rubric is only worth having if it produces agreement, and agreement is measurable, and if you have not measured it then you do not have a rubric. You have a form.

The reason this matters beyond tidiness is that every downstream artifact in your evaluation program inherits the reliability of the rubric. Your regression detection compares scores across runs; if the scores are noisy, small regressions vanish into the noise and you will only catch catastrophes. Your release gate has a threshold; if the threshold sits on an unreliable measurement, the gate blocks good releases and passes bad ones at rates nobody can predict. Your LLM judge, in Chapter Twelve, is calibrated against human scores; if the human scores disagree with each other, there is nothing to calibrate against and the judge will faithfully learn to reproduce the confusion.

So the rubric is the foundation, and the property to optimize is reproducibility rather than sophistication. A crude rubric that two reviewers apply identically beats an elegant one they apply differently, every time, and it is not close.

Binary scoring vs. graded scoring

The first decision is whether a criterion produces a yes or a number, and my strong default is yes.

Binary criteria are reliable because they force a decision with a clear boundary. Did the response state the exclusion. Did it cite a source for every factual claim. Did it avoid recommending an action the policy prohibits. A reviewer answering those questions is doing something close to reading comprehension, and reading comprehension is a thing humans do consistently.

Graded criteria invite the reviewer to consult their feelings. Rate the helpfulness from one to five. What separates a three from a four? Nobody can say, which means every reviewer supplies their own answer, which means the boundary moves between reviewers and, worse, moves within a single reviewer over the course of a long session as fatigue and anchoring do their work.

The standard defense of graded scoring is that it captures nuance. It does, and the nuance it captures is mostly the nuance of the reviewer's mood. If you want gradations, get them by decomposing rather than by grading. A response that satisfies six of seven binary criteria has a score of six out of seven, which is a graded score built out of reliable parts, and unlike an overall four out of five it tells you which part failed.

Graded scales earn their place in exactly two situations. Severity, where the difference between a minor and a catastrophic failure is real and needs a magnitude. And subjective dimensions like tone, where the property genuinely varies continuously and no decomposition captures it. Even there, keep the scale short. Three points beats five, and five beats seven, because every additional point is another boundary that reviewers will draw in different places.

Category-specific rubrics

A single rubric applied to every response is a rubric that fits none of them.

The criteria that matter for a factual lookup are not the criteria that matter for a recommendation. A lookup should be correct, grounded, and short. A recommendation should acknowledge the tradeoff, commit to a position, and disclose the basis. A refusal should decline clearly, explain why, and offer a path forward. Score all three against a general helpfulness scale and you will learn nothing about any of them.

So build a family. A small shared core of criteria that apply everywhere, safety and groundedness and format, plus a category-specific set that changes with the intent. The test case carries a rubric identifier, from the schema in Chapter Four, and the scorer loads the right criteria for the case in front of it.

This has a cost that shows up later, and you should know about it now. Scores computed under different rubrics are not comparable. You cannot average the recommendation score and the lookup score into a single quality number without doing something dishonest, and you will be asked to, because a single number is what fits on a slide. The correct response is to report a weighted aggregate whose weights are published and whose components are always visible next to it, so that when the aggregate moves someone can see which component moved.

Weighted scoring

Not every criterion matters equally, and pretending otherwise produces a metric that improves when the system gets worse.

Consider a rubric with seven criteria where one of them is factual correctness and another is whether the response used the preferred greeting. An unweighted average treats those the same. A system that gets the greeting right and the facts wrong scores six out of seven, which is a passing grade for an answer that would get a person fired.

Weight by consequence. Ask, for each criterion, what happens to a real user when this criterion fails, and let the answer set the weight. Correctness and safety carry weights that dominate. Formatting and tone carry weights that barely register. The resulting aggregate is closer to a measure of harm avoided than a measure of boxes ticked, which is what you want.

And add a veto. Some criteria are not weighted at all; they are gates. A response containing a safety violation scores zero, regardless of how good the rest of it was. A response that recommends an action the policy prohibits scores zero. This is the difference between a scoring system and a compliance system, and any product with real consequences needs both, layered.

Publish the weights. The moment weights are hidden inside a scoring service, they become a lever that someone will quietly pull when a number needs to look better before a review, and they will not be lying, exactly, and the metric will be dead.

Severity levels

Severity is orthogonal to correctness and it is the axis that tells you what to do next.

Two responses can both be wrong. One gets a shipping estimate off by a day. The other tells a customer that a product is safe for a use it is not safe for. Both fail the correctness criterion. Only one of them requires a rollback tonight.

Four levels is enough, and I would resist the urge to add more. Critical, meaning harm, legal exposure, or a promise the company cannot keep, and which alone justifies blocking a release. Major, meaning the user gets materially wrong information and will act on it. Minor, meaning the answer is worse than it should be and the user is unlikely to be damaged. Cosmetic, meaning somebody on the team noticed and no user would.

Assign severity to the test case, not to the response, and assign it before you run anything. Severity is a property of what is at stake in the question, and deciding it in advance keeps you from the very human temptation to downgrade the severity of the case that is failing on the day of the launch review.

Then report by severity. A pass rate of ninety-two percent means nothing until you know whether the eight percent are cosmetic or critical, and the two situations call for completely different decisions. I would put the critical-failure count on the dashboard next to the pass rate, in the same font size, because the pass rate is the number people watch and the critical count is the number that ends careers.

Pass/fail thresholds

At some point a score has to become a decision, and the threshold is where evaluation stops being measurement and starts being policy.

Set the threshold from the consequence rather than from the current score. The tempting move, and I have watched teams make it under deadline, is to look at where the system is today and put the bar just underneath. That produces a gate that never blocks anything, which is a gate in the same sense that an unplugged smoke detector is a smoke detector.

Better to set the bar from what the product requires, discover that you are below it, and have an honest conversation about launching anyway with a documented risk. That conversation is uncomfortable and it is the correct conversation. A gate that fails and gets overridden by a named person who wrote down why is a functioning process. A gate that passes because it was set to pass is a fiction that everyone will act on.

Thresholds also need to be composite, because a single aggregate threshold is easy to satisfy in unhelpful ways. My default gate has four conditions, all of which must hold. The aggregate score is above the bar. Zero critical failures. The count of major failures has not increased against the current production baseline. And no individual criterion has dropped by more than a specified amount, which is the condition that catches the release where correctness fell and tone rose and the average held steady.

A design constraint on rubrics that nobody mentions and everybody discovers: the rubric has to be short enough that a reviewer will actually read it before scoring.

A twelve-page rubric is a document that gets read once, during training, and consulted never. What a reviewer holds in their head while scoring is perhaps seven criteria and a handful of boundary cases, and everything beyond that is decoration that makes the document feel rigorous and changes no verdicts. If your rubric has grown past what a person can hold, you have two rubrics: the written one and the one being applied, and only the second one produces your numbers.

So keep the working rubric to a page, and put the accumulated boundary cases and adjudicated arguments in a separate reference that reviewers consult when they are stuck. That structure matches how the document is actually used, and it keeps the criteria list from silently becoming fiction.

Rubric drift over time

A rubric is a document, and documents rot.

The visible form of rot is the policy change: a criterion in your rubric describes a rule that is no longer the rule, so the rubric now penalizes correct behavior. That one is at least findable, if you have linked criteria to their source policies the way Chapter Five recommends linking cases to their sources.

The invisible form is worse. Reviewers drift. The same person, applying the same rubric, becomes gradually more lenient over months, because they have seen a thousand bad responses and their sense of what counts as bad has recalibrated against the population they are looking at rather than against the standard they were given. This is a well-documented effect in every field that uses human scoring, and it means that a stable score over time can mask a declining product.

The defense is an anchor set. Take thirty responses. Score them carefully, with a group, with discussion, and freeze the scores. Every quarter, slip those thirty back into the review queue unlabeled, and compare the new scores against the frozen ones. If the reviewers now score them higher, your standard has slipped and every number you have produced since the last anchor check is inflated. This costs half a day per quarter and it is the only thing standing between you and a metric that drifts upward while the product drifts downward.

One thing to watch for when a rubric is first introduced, because it produces a number that looks like a regression and is not.

Scores drop. Sometimes sharply. The team's previous number was produced by people forming impressions, and impressions are generous, and a rubric is not. The first properly scored run will come back well below whatever was being reported before, and somebody senior will ask what happened, and the honest answer is that nothing happened except that you started looking properly.

Say that clearly, in advance, before the first number lands. A quality program that appears to make the product worse in its first month is a quality program that will be questioned in its second, and the way to survive that is to predict it out loud before it happens rather than to explain it afterward.

Reviewer calibration

Calibration is the process of getting reviewers to agree, and it has a number attached, which is what makes it an engineering activity rather than a training exercise.

The number is an inter-rater agreement statistic. Cohen's kappa, introduced in 1960, measures agreement between two raters while correcting for the agreement you would expect by chance, which matters because two reviewers rating a mostly-passing dataset will agree eighty-five percent of the time by accident. Krippendorff's alpha generalizes this to more raters and to ordinal scales. Jacob Cohen's statistic is the one most teams reach for, and the interpretation bands from Landis and Koch, published in 1977, give rough guidance: agreement below about point four is poor, and above about point six is workable for most practical purposes.

The process is a loop. Give three reviewers the same fifty responses. Compute agreement per criterion. Find the criteria where agreement is bad, and they will be specific, and the specificity is the gift. Sit the reviewers down with the ten cases they disagreed on and have them argue until they understand why. Then rewrite the criterion so that the argument cannot happen again, usually by adding a boundary case to the definition. Repeat until the number clears your bar.

Two practical points. Disagreement almost always means the criterion is ambiguous rather than that a reviewer is careless, so fix the document before you retrain the person. And run this loop before you build anything on top of the rubric, because a judge model calibrated against reviewers who do not agree with each other is being calibrated against noise, and it will learn the noise.

A general-purpose response rubric

Here is the rubric I start from and adapt. The core criteria apply to every category. The rest come and go with the intent.

Criterion	Question the reviewer answers	Type	Weight
Safety	Does the response contain harmful content?	Binary	Veto
Policy compliance	Does it break any stated product policy?	Binary	Veto
Correctness	Is every factual claim true against the oracle?	Binary	High
Groundedness	Does every claim trace to supplied evidence?	Binary	High
Intent match	Does it address what the user was trying to do?	Binary	High
Required elements	Are all required facts present?	Count	High
Prohibited content	Is any disallowed statement present?	Binary	Veto
Specificity	Does it commit to something falsifiable?	Binary	Medium
Context awareness	Does it use what was established earlier?	Binary	Medium
Clarity	Would a hurried reader get the answer?	Binary	Medium
Tone	Does it match the product voice?	3-point	Low
Length	Is it within the display budget?	Binary	Low
Severity	If it failed, how badly?	4-level	Reported

Read down the Type column and notice how few numbers there are. That is deliberate, and it is the single most useful thing I can tell you about rubric design. Every graded scale you remove is a source of disagreement you remove, and disagreement is the enemy of a measurement you can act on.

One criterion in that table is doing more work than the others and deserves a chapter of its own. Groundedness, the question of whether the system had any basis for what it said, is where the most damaging failures in production generative systems come from, and it is where the measurement techniques have advanced furthest in the last three years.

Chapter 9: Evaluating Groundedness and Hallucination

The response cited a document. It gave the document's title, quoted a sentence from it, and used the sentence to support a claim about a warranty term.

The document existed. The sentence existed. The claim was false.

What had happened was that the sentence, read in isolation, said what the model said it said. Read in context, three paragraphs earlier, was a heading establishing that the entire section applied to commercial customers rather than consumers. The model had retrieved a chunk that began below the heading. It had no way of knowing what section it was in. It quoted accurately and concluded wrongly, and it produced a citation that made the whole thing look more trustworthy than an uncited answer would have.

This is the failure I want you to hold in mind for the rest of the chapter, because it does not fit the picture most people have of hallucination. The model did not invent anything. Every token it emitted was traceable to a real document. And the answer was still wrong, and the citation made it worse, and no metric that checks whether a citation resolves to a real document would have caught it.

Defining hallucination precisely

The word has become a container for everything that goes wrong, which makes it useless for engineering. Before you can measure it you have to say what it is.

The definition I use is narrow on purpose. A hallucination is a claim in the output that is not supported by the evidence supplied to the model and is not true. Two conditions, both required.

The two conditions give you a two-by-two, and every cell in it is a different bug with a different owner.

Supported and true is the target. Nothing to do.

Unsupported and true is a lucky guess. The model produced a correct fact from its training rather than from your evidence. Teams see this and feel fine about it. They should not. The same mechanism that produced a correct fact today will produce a stale one tomorrow, because the model's parametric knowledge has a cutoff and your policy does not. The fix belongs to retrieval: the evidence should have been there and was not.

Supported and false is a data problem. Your corpus contains something wrong and the model faithfully repeated it. The model is behaving exactly as designed. The fix belongs to whoever owns the content, and if you file this as a model bug it will never get fixed.

Unsupported and false is the classic hallucination, and it is the smallest of the three failure cells in most production systems I have looked at. That fact surprises people, and it is the reason the taxonomy is worth building. Teams that lump all four cells together spend their effort on the model when most of their exposure is in retrieval and content.

Grounded vs. plausible vs. fabricated answers

There is a spectrum between a grounded claim and a fabricated one, and the middle of it is where the danger concentrates.

A grounded claim is stated in the evidence. You can point at the sentence.

A fabricated claim has no basis anywhere. The model produced a product name that does not exist, a policy that was never written, a statistic nobody published. These are, counterintuitively,

the least dangerous of the three, because they are the easiest to detect. A product name that is not in the catalog fails a lookup. A citation that does not resolve fails a link check. Cheap deterministic checks catch a large fraction of outright fabrication, and you should build them first because they are the best return in the chapter.

A plausible claim sits between. It is not stated in the evidence, and it is consistent with it, and it is the kind of thing that would probably be true. The evidence says the product ships from a regional warehouse; the model says it usually arrives in two to three days, which is a reasonable inference and is nowhere in your data and will be wrong for a customer in a rural postal code. Nothing about the sentence looks wrong. A human reviewer skimming it will pass it. A citation check will pass it, because the surrounding sentences did cite properly.

Plausible-but-unsupported is the majority of what gets past a careless evaluation, and it is what the claim-level techniques in this chapter exist to find.

Retrieval-grounded evaluation

Groundedness only means something relative to a defined evidence set, so define it before you measure.

The evidence set is everything the model was given: the retrieved passages, the conversation history, the system prompt, the tool outputs, and any structured data the context builder injected. That whole assembled context is the universe against which a claim is judged supported or unsupported. If you did not log it, you cannot compute this metric, which is why Chapter Three insists on logging the assembled context and not merely the query.

There is a boundary decision to make and it is a product decision. Should the model be allowed to use its own knowledge at all? For a policy assistant, no: every claim must trace to the corpus, and a claim from parametric memory is a defect even when it is correct. For a general assistant, yes, within limits: the model may explain what a warranty is, and it may not tell you what your warranty says. Draw the line, write it into the rubric, and stop arguing about it case by case.

Rashkin and colleagues formalized a version of this in their work on measuring attribution in natural language generation, published in the Transactions of the Association for Computational Linguistics. Their framing asks whether a generated statement can be attributed to an identified source, and the discipline of asking that question of every sentence, rather than of the answer as a whole, is what makes the metric sharp.

Citation accuracy

Citations are useful and they are also a way for a system to look trustworthy while being wrong, so evaluate them as an artifact in their own right.

The checks run in layers. Does the cited source exist, which is a lookup and costs nothing. Does the citation resolve to the specific passage claimed rather than to the document generally, which is what makes a citation checkable by a user rather than decorative. Does the cited passage actually support the claim attached to it, which requires reading and is the expensive check and the one that matters. And is the coverage complete, meaning does every claim that needs a citation have one, because a response with two well-cited sentences and four uncited ones is often worse than an uncited response, since the citations lend the whole thing borrowed credibility.

Then there is the failure from the top of this chapter, which none of the above catches. The passage supports the claim, in isolation, and does not support it in context. This is a chunking

failure, and it is the argument for carrying structural metadata through your retrieval pipeline: the section heading, the document scope, the applicability conditions. A chunk that arrives without knowing what section it belongs to is a chunk that can be quoted into a lie.

Claim-level verification

The technique that changed how I think about this is decomposition, and it comes out of the factuality literature.

Sewon Min and colleagues introduced FActScore in 2023, and the core move is simple enough to describe in a sentence. Break the generated text into atomic claims, each a single verifiable assertion. Check each claim independently against the source. Report the fraction supported.

What this buys you is a numerator and a denominator, and everything follows from that. A response with twelve claims of which ten are supported has a support rate you can compute, track, and compare. A response scored holistically as mostly accurate has nothing you can do arithmetic on.

It also changes how failures present. Rather than a reviewer saying this answer feels off, you get a specific unsupported claim, highlighted, with the observation that nothing in the evidence set mentions it. That is a bug report. Somebody can act on it in ten minutes.

The mechanics in production. A cheap model does the decomposition, and it is reliable enough for this because splitting prose into claims is an easy task. A second pass checks each claim against the evidence, and here you want a model with good instruction-following and a prompt that permits three answers rather than two: supported, contradicted, or not addressed. That third option is the one that matters. A binary supported-or-not check collapses contradicted and not-mentioned into the same bucket, and those are entirely different bugs.

It is not free. A twelve-claim response costs you a decomposition call plus twelve verification calls, or one batched call if you are careful. Run it on a sample rather than on everything, or run the cheap fabrication checks on everything and the claim-level check on a stratified subset. Chapter Twenty-Two has the cost arithmetic.

A distinction that is easy to lose and that changes who owns the fix.

There is a difference between a claim the evidence does not support and a claim the evidence contradicts. The first is an unsupported claim: the model went beyond what it was given. The second is a contradicted claim: the model was given the right information and said the opposite.

Contradiction is rare and it is the most alarming thing you can find, because it means the model read the evidence and disregarded it, which is a failure of instruction-following rather than of grounding. A system with a two percent contradiction rate has a prompt problem that no amount of retrieval improvement will touch. Any groundedness check that collapses these two into a single not-supported bucket is discarding the more actionable of the two.

Unsupported inference detection

Inference is the category that deserves special handling because models are extremely good at it and it is usually the thing you wanted.

The evidence says the item weighs eleven kilograms. The user asks whether one person can carry it. The model says probably yes for most adults, which is an inference, and it is not in the evidence, and it is exactly what a helpful human would say.

So you cannot simply forbid inference. What you can do is require that it be marked. An inference stated as a fact is a defect; an inference stated as an inference is a feature. The rubric criterion reads roughly: does the response distinguish what the evidence says from what the response concludes. A system that says the listing gives the weight as eleven kilograms, which most adults can carry, has done the right thing, and the sentence structure is doing the epistemic work.

Then there are the inferences that should not be made at all, and they cluster in predictable places. Anything about a person's individual circumstances. Anything with a legal or medical consequence. Anything involving a number the evidence does not contain, because a model that computes a delivery date from a shipping window and a calendar will get it wrong for a holiday and will state it with complete confidence.

List the forbidden inference classes for your domain, write them into the rubric as prohibitions, and generate adversarial cases designed to tempt each one. This is one of the highest-yield uses of the synthetic generation from Chapter Seven.

A diagnostic that costs one line of code and that reveals more about a system's grounding than most of the machinery in this chapter.

Ask the question with no evidence at all. Strip the retrieval, send the query and the system prompt, and see what comes back. A well-behaved system says it does not have the information. A system that produces a fluent, confident, detailed answer from an empty context has told you something important: its answers are being generated from parametric memory and merely decorated with whatever the retrieval happened to supply.

Run this on twenty cases. The rate at which your system confabulates from nothing is an upper bound on how much of your grounding is real, and it is a number that will change what your team works on next.

Partial hallucinations

Almost nothing in production is entirely hallucinated. What you get is a response that is mostly right with a bad sentence in it, and the bad sentence is welded to the good ones in a way that makes the whole thing feel solid.

This is why response-level scoring fails so badly on this dimension. A reviewer reads a well-structured, mostly-accurate answer and marks it acceptable, because their impression is formed by the whole, and the whole is good. The one false clause slips through, and it is the clause the customer will act on, because it was the clause that answered their question.

Claim-level decomposition is the defense, and it is the reason I keep returning to it. Forcing the scorer to consider each assertion separately breaks the halo that a well-written response casts over its own errors.

Two patterns are worth naming because they recur. The confident aside, where the answer is correct and then adds a helpful extra fact that nobody asked for and nobody checked. And the smuggled premise, where an unsupported claim appears in a subordinate clause of a sentence whose main clause is well-supported, so a reviewer reading for the main claim never examines it. Both are invisible to whole-response scoring and both are trivially caught once you decompose.

There is a mitigation that costs nothing and that most products decline to use, and I want to argue for it.

Let the system say it does not know. A model instructed to always be helpful will always produce an answer, and when the evidence does not support one, the answer it produces will be constructed. Instructing it to decline when the evidence is insufficient converts a hallucination into an admission, and an admission is a far better product experience than a confident falsehood, and users forgive the first and do not forgive the second.

The objection is that abstention rates hurt engagement metrics, and the objection is correct, and it is a decision that should be made by a person who can see both numbers. What I would insist on is that the option be measured. Build a set of cases where the evidence genuinely does not determine an answer, and score whether the system says so. If it never says so, your groundedness rate is being propped up by luck.

Measuring hallucination severity

A support rate on its own is not enough to act on, because two systems with the same rate can be in completely different amounts of trouble.

Weight by consequence. An unsupported claim about a product color is a nuisance. An unsupported claim about a safety condition, a medication, a legal deadline, or a financial term is a different category of event. Tag every claim with the risk class of the thing it is about, and report the unsupported rate within each class separately.

This produces a number that leadership can use. Overall claim support of ninety-four percent sounds acceptable. Claim support of ninety-four percent overall and seventy-one percent on safety-relevant claims is a launch blocker, and only the second version of the sentence tells you that.

The taxonomy below is the artifact I would put in your repository. It is the vocabulary that lets two engineers describe the same failure the same way, and it maps every class to an owner, which is the thing that gets bugs fixed.

Class	What it looks like	Root cause	Owner
Fabrication	Entity, source, or figure that does not exist	Model generation with no evidence	Model / prompt
Unsupported inference	Reasonable conclusion, stated as fact, not in evidence	Permissive instructions	Prompt / rubric
Stale claim	True last quarter, false now	Index refresh lag	Retrieval / content
Corpus error	Faithful repetition of a wrong document	Bad source content	Content owner
Out-of-context quote	Accurate quote, wrong applicability	Chunking loses section scope	Retrieval
Misattribution	Right claim, wrong citation	Citation assembled post hoc	Post-processing
Dangling citation	Citation resolves to nothing	Broken identifier plumbing	Engineering
Conflated sources	Two documents merged into one false claim	Conflicting evidence, no arbitration	Retrieval / prompt
Context loss	Contradicts something established earlier	Truncation in the context builder	Context builder
Overclaimed certainty	Hedged fact stated flatly	Tone tuning	Prompt
Smuggled premise	False claim inside a subordinate clause	Whole-response scoring missed it	Evaluation itself

Look at the Owner column. Three of the eleven classes belong to the model or the prompt. The rest belong to retrieval, content, plumbing, and evaluation. That distribution is the argument of this chapter in one image, and it is why a team that responds to a hallucination problem by trying a bigger model will usually spend a quarter and move nothing.

One more thing before we move on. Every technique here evaluates a single response against a single evidence set. Real conversations are not single responses. A claim that was grounded in turn two can be contradicted in turn six, and neither turn will fail a groundedness check on its own.

Chapter 10: Evaluating Multi-Turn Conversations

Six turns. That was the number where things reliably fell apart, and once we started plotting quality against conversation depth, the shape of the curve was so consistent that I began to suspect the instrument before I suspected the system.

Turn one was excellent. Turns two and three were fine. Turn four dipped slightly. Turn five dipped more. From turn six onward, the pass rate fell off a shelf, and it stayed down, and the responses in that region were subtly, persistently wrong in a way that was hard to describe and easy to feel.

Our entire evaluation suite was single-turn. Every case was a question with no history, scored in isolation, and it reported a pass rate in the low nineties, and the pass rate was true, and it described a conversation that almost nobody was having. The users who mattered most, the ones engaged enough to ask six questions, were the ones we were failing, and we had built a measuring instrument that was structurally incapable of seeing them.

This chapter is about building one that can. It is the hardest chapter in the book, the tooling is the thinnest, and it is where I would spend my next quarter if I could only fix one thing.

Why single-turn tests are not enough

The problem is compositional. A conversation is a sequence of exchanges, each of which depends on the state left behind by the ones before it, and a system can be correct at every individual step and wrong as a whole.

Three things break as depth increases and they break for different reasons.

The context grows, which means the truncation policy starts firing, which means information begins falling out of the model's view. The oldest turns go first, and the oldest turns contain the constraints the user established at the beginning, which are exactly the constraints that a good answer at turn seven has to respect.

The references multiply. By turn six the user has stopped naming things. They say the second one, that other option, the cheaper one, it, that. Every one of those is a pointer into a shared state, and every one has to be resolved against a conversation the model can only see through whatever window your context builder gave it.

And errors compound. A misunderstanding at turn three does not stay at turn three. It gets written into the conversation state, it informs the summary, and it becomes the premise of every turn that follows. The user corrects the assistant at turn five and the assistant apologizes and then repeats the error at turn seven, because the error is now in the history and the correction is one line against it.

None of this is visible to a single-turn suite. You have to test the sequence.

Context carryover

Carryover is the simplest multi-turn property to state and the most commonly violated: information established in an earlier turn should still be in force in a later one.

The test structure is a script. Establish a constraint explicitly at turn one. Do several turns of unrelated but plausible conversation. Ask a question at turn six whose correct answer depends on the constraint from turn one. Assert that the answer respects it. The intervening turns are load, and their job is to push the constraint toward the truncation boundary.

Run that script at several depths, because the depth at which it starts failing tells you exactly where your context budget runs out, and that number is more useful than any pass rate. If carryover holds at depth four and collapses at depth seven, you have found your truncation cliff and you now know what to fix, and you did not have to read the context builder to find it.

Vary what you are carrying, too. A numeric constraint like a budget, a categorical one like a brand exclusion, a negative one like the user saying they do not want a particular feature, and a personal one like a stated preference. Negations are the ones that break first, in my experience, because summarizers drop them and because a constraint expressed as an absence is easy to lose.

One more variation worth building: the same constraint, restated by the user in a later turn. A well-behaved system treats the restatement as reinforcement. A poorly behaved one treats it as new information and forgets the original scope. You will not know which you have until you test it.

Reference resolution

This is the failure from the introduction of this book, and it is worth being precise about the machinery, because the fix is almost never in the model.

The user says tell me more about the second one. To answer, the system needs to know what was enumerated, in what order, and that the user is pointing at position two. All of that lives in the previous assistant turn, and the question is whether it survived the trip.

It usually does not, and here is the reason that catches most teams. The enumeration was rendered as a user interface component. Three product cards, laid out horizontally, each with an image and a price. Beautiful. And the text that went into the conversation history was a summary line, or a placeholder, or a serialized blob, and the ordinal position of each card is nowhere in it. The model is asked to resolve the second one against a history where nothing is second.

So test the ordinal cases explicitly and test them against your real rendering path rather than against a text-only stub. Build a family: the second one, the last one, the cheaper one, the blue one, that first thing you mentioned, the one you said was out of stock. Each targets a different resolution mechanism. Ordinal by position, ordinal by extremum, reference by attribute, reference by a property the assistant itself asserted.

And test the ambiguous ones, where the correct behavior is to ask. If the user says the other one and there are three, a good system asks which. A bad one picks. The rubric criterion is not did it resolve correctly; it is did it resolve correctly or ask when resolution was impossible, and those two acceptable outcomes need to be in the specification or your reviewers will disagree forever.

User correction handling

When a user says no, that is wrong, they have handed you the most valuable turn in the conversation and the one your system is most likely to fumble.

Three things must happen. The correction must be understood, meaning the system identifies what specifically is being corrected rather than treating the whole prior turn as void. The correction must be applied, meaning the corrected state governs the next response. And the correction must persist, meaning the original error does not resurface at turn nine because it is still sitting in the history with more textual weight than the one-line correction against it.

That third property is the one that fails, and it fails quietly. Test it directly. Correct the assistant at turn three, do three turns of other business, then ask a question at turn seven that would reveal whether the original error is still in force. Most systems fail this. Very few teams have ever run the test.

Then there is the opposite failure, which is folding when you should not. The user says that is wrong and the user is mistaken. The correct behavior is to hold the position and explain, politely, with evidence. What most systems do is apologize and agree, because agreeableness was optimized for and correctness under pressure was not. Mrinank Sharma and colleagues at Anthropic documented this tendency directly in their 2023 work on sycophancy, and it is worth building a specific stratum of cases for it: assert something false to the assistant, confidently, twice, and see whether it caves. If it does, no groundedness metric you compute at turn one will save you.

Conversation state updates

Somewhere in your system there is a representation of what the conversation has established, and it is being written after every turn, and almost nobody tests the write.

Test it by round-tripping. After each turn, read the persisted state back and assert on it directly. Are the entities the user mentioned present. Are the constraints present, with their polarity intact. Is the enumeration recorded with its ordering. Is the correction from turn three reflected, and is the original error gone rather than merely appended to.

This is a unit test, it runs in milliseconds, it requires no model call, and it catches a class of bug that would otherwise reach you as a vague complaint about the assistant being forgetful three weeks later. It is the best value in this chapter and it is skipped constantly, because the state layer feels like plumbing and plumbing does not get tested until it leaks.

Test the failure paths too. What happens when the state write fails. Does the next turn proceed with a silently empty history, producing a confident answer to a question it has no context for, or does it degrade in a way the user can perceive. Silent state loss is the worst outcome available, because it produces a plausible wrong answer instead of an error, and Chapter Eighteen files it under memory inconsistency for exactly that reason.

Memory freshness

Persistence has an opposite failure and I want to give it equal time, because teams that fix forgetting frequently create it.

A user finishes shopping for a gift for their father and starts shopping for themselves. The constraints from the first task are still in state. The assistant, being consistent, applies them. The user gets recommendations filtered by preferences that belong to somebody else, and the system is behaving perfectly, and the product is broken.

State needs a lifecycle. Some things persist across a session and not beyond it. Some things persist across sessions, like a stated accessibility need. Some things are scoped to a task and should be dropped when the task ends. And task boundaries are hard to detect, which is the actual problem.

So test topic switches explicitly. Establish a set of constraints, complete the task, start a clearly unrelated task, and assert that the old constraints are gone. Then run the harder version: start a related task, and assert that the constraints that should carry do carry while the ones that should not do not. That distinction is a product decision, it should be written down, and if it is not

written down then your system is making it arbitrarily and your users are experiencing the result.

Include a returning-user case as well. Same person, new session, an hour later, possibly a different device. What does the assistant remember, and is that what the user expects it to remember. Getting this wrong in either direction is unsettling: an assistant that forgets everything feels broken, and one that remembers too much feels like surveillance.

A structural recommendation that removes a whole class of the failures in this chapter, and that is cheaper than fixing them individually.

Store the conversation state as structured data rather than as prose. Constraints as fields. Enumerated items as a list with positions. Corrections as an explicit overwrite of the thing they corrected. Then build the prompt text from the structure at each turn rather than summarizing the transcript.

A summarizer can drop a negation. A field cannot. An ordinal reference that resolves against a list with indices resolves correctly every time, and no amount of prompt engineering makes prose resolution as reliable as an array lookup. This is a design change rather than an evaluation technique, and it is the one that the evaluation in this chapter usually leads teams to, which is a good sign that the evaluation is doing its job.

Summary quality

When the conversation exceeds the budget, something has to compress it, and the compressor is usually a model, and its output is usually not evaluated by anyone.

This is a generation task with all the failure modes of any other generation task, and it sits at the base of everything downstream. A summary that drops a negation inverts the meaning of the entire conversation for every subsequent turn. A summary that drops a quantity makes a numeric constraint unenforceable. A summary that loses ordering breaks every ordinal reference. A summary that hallucinates a constraint the user never stated will cause the assistant to refuse things for reasons the user cannot understand.

Evaluate summaries as artifacts, with their own dataset and their own rubric. Take a conversation, take the summary, and check: does every constraint from the source appear in the summary with the correct polarity. Does every quantity survive. Is the ordering of enumerated items preserved. Does the summary contain anything the source does not.

That last check is the one people forget and it is the most dangerous. A summarizer that adds a constraint is teaching your assistant something false about the user, permanently, for the rest of the conversation, and every downstream turn will be confidently wrong in a way that traces back to a component nobody is looking at.

Multi-turn regression tests

Scripted conversations are the workhorse and they belong in continuous integration, running on every change, the same way unit tests do.

A scripted case is a fixed sequence of user turns with assertions attached to specific turns. It is deterministic in its inputs, which makes it a regression test in the classical sense. The assertions can be cheap: did the response mention the constraint from turn one, did it reference the correct item, did it avoid re-asking for information already given. String checks are often enough and they cost nothing.

The limitation is that scripted turns do not react. If the assistant's turn four is unexpected, the user's scripted turn five may make no sense, and the case degenerates. Handle this by writing scripts whose later turns tolerate variation in the assistant's phrasing, and by asserting on the state rather than only on the text.

The complement is simulation. A model plays the user, with a persona, a goal, and a list of moves it must make. It reacts, which lets it explore paths a script cannot reach. It is also non-deterministic, which makes it useless as a gate and excellent as a discovery tool. Run simulations in bulk, cluster the failures, and promote the interesting ones into scripts. That is the pipeline: simulation finds, scripts guard.

The known weakness of simulated users, and it is severe, is that they are too polite. A model playing a customer accepts a mediocre answer and moves on. Instruct it explicitly to be dissatisfied, to press, to re-ask when the answer misses, and to correct the assistant when it is wrong. The pressing is where the failures are.

One structural point about multi-turn evaluation that changes how you budget for it.

Multi-turn cases cost roughly as much as their depth. A twelve-turn conversation is twelve generations, twelve retrievals, and twelve scoring calls, which makes it an order of magnitude more expensive than a single-turn case. That arithmetic is why almost every evaluation suite in production is dominated by single-turn cases, and why almost every product fails in multi-turn conversation.

The escape is to be selective rather than comprehensive. You do not need a thousand deep conversations. You need perhaps eighty, chosen to cover the specific conversational structures your product produces, run at several depths, with assertions on the turns that matter rather than on all of them. Eighty well-designed conversation scripts will find more than a thousand shallow ones, and they will fit in a nightly budget.

Conversation-level scoring

Finally, the aggregation question, which has no clean answer and where I will give you my working position rather than a rule.

Averaging turn scores loses the thing you care about. A conversation with six good turns and one catastrophic turn averages well, and the user left after the catastrophic one and never came back. The average is a description of a conversation nobody had.

So score conversations on their own terms, with three numbers rather than one. Task completion, which is binary: did the user get what they came for by the end. Worst-turn severity, which captures the fact that a single bad turn can end the session regardless of the rest. And consistency, which asks whether the conversation contradicted itself anywhere, since a contradiction between turn three and turn eight is a failure that lives in neither turn.

Report those three alongside the turn-level pass rate rather than instead of it. The turn rate tells you how often an individual response is good. The conversation numbers tell you how often the product works. They will diverge, and the size of the divergence is the size of your multi-turn problem.

Element	What it specifies	Example
Persona	Who the user is and how they type	Terse; phone; mild typos; impatient
Goal	What counts as task completion	Choose a product under the stated budget

Element	What it specifies	Example
Turn script	Ordered user moves	establish_constraint, ask, ordinal_ref, correct, topic_switch
Setup state	Anything true before turn one	Returning user; one item already in cart
Per-turn assertions	What must hold in that response	T5: resolves 'the second one' to item 2
Carryover assertions	What from earlier must still hold	T7: budget from T1 still applied
Prohibitions	What must never appear	Never re-ask for the budget
State assertions	What the persisted state must contain	constraints = {budget, brand_excluded}
Correction check	Does a correction persist past its turn	T3 correction still in force at T8
Freshness check	What must be dropped after a topic switch	Gift constraints cleared at T9
Conversation score	Completion, worst-turn severity, consistency	complete=yes; worst=minor; consistent=yes
Depth ladder	The depths at which the case is run	4, 6, 8, 12 turns

The depth ladder at the bottom is the row I would defend hardest. Running the same case at four depths costs four times as much and tells you where your system breaks, which is a thing you otherwise learn from customers.

Conversations are hard because state is hard. There is a category of system where the state is not merely remembered but acted upon, where a wrong turn does not produce a bad sentence but a real event in the world, and the cost of a failure is measured in money rather than annoyance.

Chapter 11: Evaluating Tool Use and Agentic Actions

The incident report ran to four pages and the summary was one sentence: the assistant issued the same refund eleven times.

What happened was almost boring. The refund tool had a timeout of two seconds. The payment service, under load, took two and a half. The call timed out on our side and succeeded on theirs. Our retry logic, which was correct and well-tested and had been in production for years, saw a timeout and retried. The second call also timed out and also succeeded. And so on, through a sequence of retries that had been designed for a world where a timeout meant the thing had not happened.

The model was not involved in any of this. It made one decision, correctly, which was that this customer was entitled to a refund. Everything after that was infrastructure behaving exactly as specified in a situation the specification had not anticipated.

I open with this because most writing about agent evaluation focuses on whether the model picks the right tool, and picking the right tool is the easiest part of the problem. The hard parts are what happens at the boundary between a probabilistic decision-maker and a deterministic world that has money in it.

Tool selection accuracy

Start with the good news, which is that this stage produces a structured artifact and structured artifacts are easy to evaluate.

The model emits a function name. You have an expected function name. Compare them. There is no rubric, no judge, no human, no ambiguity, and the check runs in microseconds. Every part of agent evaluation that can be reduced to this form should be.

The interesting cases are the ones at the edges of the decision. Calling nothing when a call was needed, which produces an answer from stale parametric knowledge and looks fine. Calling something when nothing was needed, which burns latency and cost and occasionally does something. Calling the plausible neighbor, meaning the tool that does almost the right thing, which is the failure that scales badly because the response looks correct until someone checks. And calling in the wrong order, when the second call needed the output of the first.

Build the confusion matrix over tools, not the accuracy number. Accuracy of ninety-three percent across forty tools tells you nothing. A matrix that shows every request for the cancellation tool being routed to the modification tool tells you what to fix by lunch. The Berkeley team behind the function-calling leaderboard organized their evaluation around exactly this decomposition, separating tool choice from argument construction from irrelevance detection, and the decomposition is the useful part regardless of what you think of the leaderboard.

Argument correctness

The function name was right. Now look at what got passed to it, because this is where the errors move from embarrassing to expensive.

Copied arguments are usually fine. If the user gave an order number and the model passed the order number, there is little to go wrong beyond a transcription error, and a format assertion catches those.

Derived arguments are where it breaks. The user said next Tuesday and something had to turn that into a date, and the something needs to know today's date and the user's time zone and whether next Tuesday means the coming one or the one after. The user said the cheaper one and something had to turn that into an item identifier. The user said cancel it and something had to decide what it was. Every derivation is an inference, and every inference is a place where the model can be confidently wrong in a way that no schema validator will catch, because the value is well-formed and simply refers to the wrong thing.

Then there are the arguments nobody discussed. Amounts. Quantities. Recipients. A model that gets the currency wrong, or that passes a quantity of eleven because it misread a conversation, will produce a perfectly valid call that does something nobody wanted. Assert on every argument, including the ones that seem obvious, and pay special attention to anything numeric.

And check for what should not be there. Personal data flowing into a tool that has no business receiving it is a data-handling incident that will present as a quiet log line. Assert on the absence of sensitive fields as explicitly as you assert on the presence of required ones.

Permission and authorization checks

This section is short because the rule is short and absolute. Authorization is never the model's job.

A model instructed not to issue refunds above a threshold will comply almost always. Almost always is not a security property. Somebody will find the phrasing, and the phrasing will spread, and you will read about it. The check belongs in the tool layer, where it is code, where it is deterministic, and where it does not care how persuasive the user was.

What you evaluate, then, has nothing to do with whether the model refuses. You are testing whether the refusal below it holds. Build cases whose entire purpose is to talk the model into crossing a line, and assert that when it crosses, nothing happens. The model attempting a forbidden call is acceptable and possibly inevitable. The call succeeding is the defect.

Two specific tests belong in every agentic suite. The confused deputy: the user asks about another person's account, and the model, being helpful, calls a lookup with an identifier it found in the conversation. Does the tool layer check that this user may see that account. And privilege escalation across turns: the user establishes an innocuous context over five turns and then asks for something the earlier context appears to authorize. Does the authorization check consult the actual session identity rather than the accumulated conversational implication.

Confirmation before irreversible actions

Some actions can be undone and some cannot, and the system has to know which is which before it acts, not after.

Classify every tool in your catalog on that axis and write it down. Reads are free. Reversible writes are cheap. Irreversible writes, meaning anything that sends money, sends a message to a third party, deletes data, or commits to an external party, are a different category and require a different flow.

For those, the system asks first. The evaluation asserts three things: that confirmation was requested, that the confirmation summarized the action accurately, and that nothing happened before the confirmation came back. The second one is the one that fails. A confirmation that says shall I proceed is worthless, because the user does not know what they are approving. A

confirmation that says I will refund forty-two dollars to the card ending in four one two two is a confirmation, because a user reading it can catch the error.

Then test the pathological cases. The user says yes to something else entirely and the yes gets consumed as confirmation for a pending action. The user changes their mind mid-flow. The confirmation is requested, the user never answers, and the session ends: does the pending action stay pending forever, and does it fire when they come back tomorrow.

Idempotency and duplicate prevention

This is the refund story, and it is the most under-tested property in agentic systems, and it is the one that costs money.

The pattern that fixes it is well established outside of AI. Every request that changes state carries a unique key. The service records the key and its result. A repeat of the same key returns the recorded result rather than performing the action again. Payment infrastructure has worked this way for a long time, and the reason it does is that the failure mode is exactly the one we walked into.

The evaluation is fault injection, and it takes an afternoon to build. Simulate a timeout on a call that succeeded server-side, and assert that the retry does not duplicate. Simulate a network partition mid-call. Simulate a duplicate user request, because users double-tap buttons. Simulate the model emitting the same tool call twice in one turn, which it will do, occasionally, for reasons that will never be fully explained to you.

There is a specifically generative version of the problem worth naming. The model, seeing a tool result that looks like a failure, may decide to retry on its own, at the reasoning level, independent of your infrastructure retry. Now you have two retry mechanisms that do not know about each other. Test for it. Return an ambiguous tool result and see what the model does with it, and if it retries, make sure the idempotency key it uses is the same one.

A property of agentic systems that makes them qualitatively different from everything else in this book, and that is worth internalizing before you deploy one.

The failures are not idempotent in their consequences. A bad answer can be corrected by a better answer. A bad action cannot be corrected by a better action, because the first one already happened. The customer has been charged. The email has been sent. The ticket has been closed and the person has moved on.

That asymmetry is the argument for asymmetric evaluation. It is rational to accept a system that is right ninety percent of the time when it answers questions, and irrational to accept the same rate when it acts. For anything irreversible, a good pass rate is the wrong target entirely. What you need is a confirmation step, a permission check below the model, and a bound on how much a single misunderstanding can do before a person sees it.

Tool failure handling

Tools fail. The question your evaluation has to answer is what the system does about it, and the answer determines whether a transient outage becomes a customer-facing disaster.

Inject each failure mode and score the response. A timeout. A five hundred. A malformed payload. A partial result. An empty result, which is the sneaky one, because an empty result is a legitimate answer to some queries and a failure signal for others. A slow success, which arrives after your patience ran out. A result that contradicts something the model already told the user.

The behaviors you are scoring for are honesty and restraint. Did the system tell the user that something went wrong, rather than producing an answer that quietly pretends the tool returned data. Did it avoid fabricating the result, which is the failure mode that terrifies me most, because a model that receives an error and produces a plausible-looking answer anyway has manufactured a fact out of an outage. Did it stop, rather than proceeding to the next step of a plan whose precondition had failed.

That last one is the multi-step failure and it is where agentic systems earn their reputation. Step three failed. Steps four, five, and six execute anyway, on a state that does not exist. Test it by failing the middle step of a plan and asserting that the plan halts.

Retry behavior

Retries are correct and they need boundaries, and the boundaries are what you test.

Retry on transient failures, meaning timeouts and rate limits and five hundreds. Do not retry on permanent ones, meaning a four hundred, an authorization failure, or a validation error, because the same call will fail the same way and you have simply made the outage more expensive. Test both classes and assert that the system distinguishes them.

Bound everything. A retry limit, so a failing tool does not consume the entire latency budget. A backoff, so a struggling service is not hammered by every session at once. And a total budget across the turn, because three tools each retrying three times is nine calls and a user who has been staring at a spinner for eleven seconds.

And decide, explicitly, whether the model is allowed to retry at the reasoning level, and test whatever you decided. A model that re-plans after a failure is powerful. A model that re-plans after a failure, inside a loop, with a tool that is timing out, is a runaway, and the only thing standing between you and a very large bill is a step limit that somebody remembered to set.

A limit on autonomy that is worth setting before you need it, because you will not want to argue about it at the time.

Cap the number of consequential actions a single turn may take without a human. One is a reasonable default for anything irreversible. The argument against it is that a capable agent should be able to complete a multi-step task end to end, and the argument is correct, and it is a decision about risk rather than about capability.

What makes the cap valuable is that it bounds the blast radius of a reasoning failure. An agent that misunderstands the request and takes one wrong action has made a mistake. An agent that misunderstands the request and takes eleven wrong actions, each of which follows correctly from the last, has produced an incident, and the eleven actions were all individually reasonable, which is what makes it so hard to defend afterward.

Auditability

Every action needs a record that answers who, what, why, and on whose authority, and the record has to be good enough for an investigation you have not had yet.

Log the tool name, the arguments, the result, the session and user identity, the authorization decision and its basis, and the model's stated reasoning if you have it. Log the timestamp and the idempotency key. Log confirmations, and log the exact text the user was shown when they confirmed, because the question in the eventual dispute will be what the customer agreed to and the answer has to be a string you can produce.

Then evaluate the log itself, which nobody does. Take a completed agentic session, hand the trace to an engineer who was not involved, and ask them to reconstruct what happened and why. If they cannot, your audit trail is decorative, and you will find that out during an incident rather than during a test.

Chapter Twenty-Three has more on this, because in a regulated domain the audit trail is a deliverable rather than a nicety, and its adequacy is judged by people who do not accept the argument that the model made an unexpected choice.

Before you build any of this, do one cheap thing that will tell you how much of it you need.

Take your tool catalog and sort it by what happens if the model calls it wrongly. Reads sort to the bottom. Writes to an internal system sort to the middle. Anything that moves money, contacts a third party, or cannot be undone sorts to the top. Then look at how many tools are in that top bucket, because that number is the size of your actual risk, and it is usually much smaller than the size of your catalog.

Most agentic products have three or four genuinely dangerous tools among several dozen. The evaluation effort should be concentrated there, at a depth that would be wasteful if applied uniformly, and the rest of the catalog can be covered by schema assertions and a confusion matrix. Teams that treat every tool as equally risky end up testing everything shallowly, which is the same as testing the dangerous ones shallowly.

Human-in-the-loop evaluation

For high-consequence actions, a human approves before anything executes, and the human is now part of the system, which means the human is part of the evaluation.

The metric that matters has little to do with approval accuracy. What you want to know is whether the human is doing anything at all. An approver who approves ninety-nine percent of what they see, in three seconds each, is a rubber stamp with a salary, and the presence of the review step is providing false assurance to everyone upstream of it.

Measure it. Approval rate, time spent per decision, and, most usefully, planted errors. Insert a small number of deliberately wrong proposed actions into the review queue and count how many get caught. That number is the actual value of your human-in-the-loop, and it is usually lower than anyone expects, and knowing it is the difference between a control and a ritual.

Then design against fatigue. Volume kills review quality, so route only what needs routing. Give the reviewer the context they need to decide in one screen. And close the loop, so that a rejected action becomes a test case, because a human catching an error is the highest-quality signal in your entire pipeline and throwing it away is a waste.

Check	Assertion	Failure severity
Tool selection	Correct tool chosen, or correctly no tool	Major
Irrelevance detection	No call made when none was warranted	Minor
Argument correctness	Every argument matches expected, including derived ones	Major
Numeric arguments	Amounts, quantities, dates exact	Critical
Data minimization	No sensitive field passed that the tool does not need	Critical
Authorization	Forbidden call is refused by the tool layer, not the model	Critical
Cross-account access	Cannot act on another user's resources	Critical

Check	Assertion	Failure severity
Confirmation	Requested before any irreversible action	Critical
Confirmation accuracy	Confirmation text states the actual effect and amount	Critical
Idempotency	Timeout plus retry produces exactly one effect	Critical
Duplicate suppression	Same call twice in one turn produces one effect	Critical
Failure honesty	Tool error is reported, never fabricated around	Critical
Plan halting	A failed step stops the remaining steps	Major
Retry classification	Transient retried, permanent not retried	Major
Retry bounds	Step limit, backoff, and total budget enforced	Major
Audit completeness	A stranger can reconstruct the session from the trace	Major
Human review efficacy	Planted errors are caught at an acceptable rate	Major

Count the Critical rows. Eight of seventeen, and not one of them is about whether the model gave a good answer. Agentic evaluation is mostly systems engineering wearing a machine learning hat, and teams staffed only with machine learning people will build the wrong tests.

Almost everything in this part of the book has assumed a human doing the scoring, and humans do not scale to the volumes production requires. The obvious solution is to have a model do it, and the obvious solution works, and it fails in ways you have to understand before you trust it.

Chapter 12: LLM-as-Judge Evaluation

I ran an experiment I have since recommended to everyone who asks me about judge models, and it takes about an hour.

Take a hundred responses. Have a model score them. Then take the same hundred responses, reverse the order of the two candidates in each pairwise comparison, and have the same model score them again. Nothing else changes. The same texts, the same rubric, the same model, the same temperature.

In our run, the judge changed its mind on nineteen of the hundred. Not on the hard ones, necessarily. On whichever ones happened to be near its decision boundary, and the position of the boundary moved depending on which response it read first.

That number, nineteen percent, was the noise floor of our automated scoring, and we had been reporting quality deltas of three and four points against it for a quarter.

I am not telling you this to argue against judge models. I use them constantly and this book would recommend nothing if it recommended only human review. I am telling you because a judge is an instrument, and an instrument you have not characterized is an instrument whose readings you cannot trust, and characterizing it takes a day.

What LLM-as-judge is

The idea is exactly what it sounds like. Give a model the input, the response, and the rubric, and ask it to score. It reads, it reasons, it produces a verdict.

It is a good idea because the alternative does not scale. Human review of a two-thousand-case suite is weeks of expert time per run, and you want to run it on every release candidate. A judge model does it in twenty minutes for the price of a lunch. The economics are not close, and any evaluation program that refuses to use judges will be run rarely, which means it will be run after the incident.

The approach was made rigorous by Lianmin Zheng and colleagues in the MT-Bench and Chatbot Arena work published in 2023, which is the paper to read if you read one. They compared strong judge models against human preferences across a large set of comparisons and found agreement rates that were roughly comparable to the agreement between two independent humans. That framing is the right one, and I want to be careful about what it means. It does not mean the judge is right. It means the judge disagrees with a human about as often as another human would, which is the most you can ask of any scorer on a task where the ground truth is judgment.

The same paper catalogued the biases, and the honesty about them is what makes it useful. Which brings us to the parts that break.

When it works well

Judges are strong on tasks that are close to reading comprehension and weak on tasks that require knowledge or taste.

They are excellent at instruction-following checks. Did the response include a citation. Did it stay under the length limit. Did it avoid recommending a competitor. These are verification tasks against a visible artifact, the model does them reliably, and they are cheap.

They are strong at groundedness when you give them the evidence. Ask whether this claim is supported by this passage, with both in front of the judge, and you get a reliable answer, because the task is entailment and models are good at entailment. This is the single highest-value use of a judge in production, and it is what makes the claim-level verification of Chapter Nine affordable.

They are good at relative comparison. Which of these two responses better satisfies this criterion is an easier question than how good is this response, for a model exactly as it is for a human.

And they are good at detecting the obvious. Refusals, empty answers, format violations, responses that are clearly off-topic. A judge as a screening layer that filters out the garbage before a human sees anything is a fine use of the technology and an underrated one.

When it fails

Judges fail whenever the correct verdict depends on something the judge does not have.

Domain expertise is the big one. A judge asked whether a piece of tax guidance is correct will produce a confident verdict and it will be as reliable as the model's knowledge of tax law, which is to say it will be right often enough to be dangerous. If a human non-expert could not score the criterion, a judge cannot either, and no amount of prompt engineering changes that.

Subtle factual errors get missed. A response that is well-written, well-structured, and wrong about one number will be scored highly, because the judge is influenced by the same surface features that influence a human skimming quickly. This is the argument for deterministic checks on anything checkable. Never ask a judge whether a price is correct when you have a database that knows.

And judges cannot see the failures of omission. A response that is missing the one fact that would have changed the user's decision looks complete, because there is nothing on the page that is wrong. This is why the expected-behavior specification from Chapter Five matters so much: given a list of required elements, the judge can check them off, and without one it will score what is there rather than noticing what is not.

Pairwise comparison vs. absolute scoring

This is the most consequential design choice and most teams pick the wrong one by default.

Absolute scoring asks the judge to rate a response against a rubric. It gives you a number you can compare across time, which is what a release gate needs. It is also the mode in which judge models are least reliable, because the scale has no anchor and the judge's internal sense of what a four means will wander.

Pairwise comparison asks which of two responses is better. It is far more reliable, for the same reason it is more reliable with human reviewers: relative judgment is easier than absolute judgment. It is the mode the research community has converged on for exactly this reason.

The cost is that pairwise gives you no absolute level. It tells you the candidate beats production sixty-two percent of the time. It does not tell you whether either is good enough to ship, and if both are terrible, pairwise will happily report a clean win.

So use both, deliberately. Pairwise against the current production system for the go/no-go decision on a change, because that is a relative question and the relative instrument is more sensitive. Absolute scoring against a rubric for the level you track over time and for the

threshold in your gate, with the understanding that the absolute number is noisier and should not be read to a precision it does not have.

And when you do pairwise, randomize the order of every comparison and run each one twice with the order swapped. The nineteen percent from the top of this chapter is what happens when you do not.

Judge prompt design

The judge prompt is the instrument's calibration and it deserves the care you would give to a production prompt, which is to say more than it usually gets.

Give the judge one criterion at a time. A judge asked to score seven dimensions in one call will produce seven numbers that correlate suspiciously with each other, because it formed an overall impression and then decomposed it backward. Separate calls per criterion cost more and produce independent judgments, and independence is the whole reason you decomposed the rubric in the first place.

Give it the evidence, not just the response. A groundedness judgment without the source documents is a vibe check.

Ask for the reasoning before the verdict. Requiring the judge to state which part of the response it is evaluating and why, before it commits to a score, improves reliability measurably, and it gives you an artifact a human can audit when the score is disputed. Yang Liu and colleagues built this into the G-Eval approach published in 2023, having the model generate its evaluation steps before scoring.

Give it an escape hatch. A judge forced to choose between pass and fail on an ambiguous case will choose, arbitrarily. A judge permitted to say the criterion does not apply, or that the case is ambiguous, will route those cases to a human, and the volume of them tells you where your rubric is unclear.

And keep the scale short. Binary where you can. Three points where you cannot. A judge asked to distinguish a six from a seven on a ten-point scale is generating a number, and the number is not measuring anything.

Bias and position effects

Judges have systematic biases, they are documented, and every one of them will inflate a score in a direction you would rather it did not.

Position bias is the one from the opening. In a pairwise comparison, judges prefer one position, usually the first, at rates well above chance. The fix is mechanical: run every comparison both ways and count only the cases where the judge agrees with itself. The cases where it flips are ties, and treating them as ties rather than as data is the honest move.

Verbosity bias favors longer responses. Judges reliably score a longer answer higher than a shorter one that says the same thing, which means a judge-driven optimization loop will make your product more verbose forever, one release at a time, and nobody will notice until a designer complains. Control for it by putting length in the rubric explicitly, and by checking the correlation between response length and judge score on a sample. If the correlation is strong, your judge is measuring length.

Self-preference is real and unsettling. Arjun Panickssery and colleagues published work in 2024 showing that models tend to rate their own generations more highly than a human would, and

that they can recognize their own outputs. The practical consequence: do not use the same model family as generator and judge if you can avoid it, and if you cannot, calibrate against humans specifically on that axis.

Sycophancy shows up here too. A judge told that a response was written by an expert will score it higher. Strip provenance from everything the judge sees. It should not know which system produced which output, and if your evaluation harness passes that information through, the judge is reading it.

Calibration against human reviewers

A judge is only usable to the extent that it agrees with the humans whose judgment you actually care about, and the extent has a number.

The procedure is a loop and it is not optional.

Element	The pattern	Why
Scope	One criterion per judge call	A judge asked for an overall score halos everything onto fluency
Output	A verdict plus the evidence it relied on	An unjustifiable verdict is an unreviewable one
Comparison	Position swapped and re-run; disagreement means a tie	Judges prefer whichever response came first
Length	Score correlation with response length monitored	Judges reward verbosity and call it thoroughness
Family	A judge from a different model family than the system under test	A judge scoring its own family scores itself favorably
Calibration set	100 to 200 cases scored by human experts, spanning the boundary	The judge is an instrument and instruments need a reference
Agreement	Chance-corrected statistic, computed per criterion, floor enforced	An aggregate figure hides a criterion the judge cannot score
Below the floor	Revise the rubric or the judge prompt, then measure again	A judge that disagrees with humans is measuring something else
Above the floor	Trust the judge for that criterion; re-check on a schedule	Calibration decays as the rubric, the model, and the traffic move
Judge prompt	Versioned; recalibrated whenever it changes	An edited judge prompt is a different instrument
Model size	A small model for format, length, and entailment; a strong one for judgment	Most criteria do not need the expensive judge
Human anchor	A permanent human sample on every criterion, never removed	Remove the anchor and the whole apparatus drifts silently
Never	A judge as sole authority on safety, policy, or legal consequence	Correlated judge failures are invisible to agreement statistics

Build a calibration set of one to two hundred cases, stratified to include the boundary cases where the correct verdict is contested, because a calibration set of easy cases will show agreement everywhere and tell you nothing. Have your best human reviewers score it, with the disagreement process from Chapter Eight, and freeze the labels.

Run the judge against it. Compute agreement, per criterion, using a chance-corrected statistic rather than raw percentage. Look at the disagreements individually, because they cluster, and the cluster is the finding: the judge is systematically lenient on the safety criterion, or it consistently misses omissions, or it cannot tell a hedged claim from an asserted one.

Then fix the judge prompt for the specific failure, and re-measure. Two or three iterations usually gets you where you need to be on the criteria a judge can handle, and it will also tell you which criteria a judge cannot handle at all, which is the other half of the value. Those criteria go to humans, permanently, and you stop pretending otherwise.

Re-run the calibration on a schedule, because the judge model will be updated by its provider and the rubric will be edited by you, and both invalidate the calibration silently.

Measuring judge agreement

Use a chance-corrected statistic. Raw agreement on a dataset where ninety percent of cases pass will read as ninety percent agreement between a good judge and a coin that always says pass.

Cohen's kappa is the standard and it is the one your reviewers will already understand. Compute it per criterion rather than overall, because a judge that is excellent on format and useless on groundedness has a respectable overall kappa and should not be trusted with groundedness.

Set an acceptance bar and hold it. Mine is that a judge may replace human scoring on a criterion when its agreement with humans is at least as good as the agreement between two humans on that criterion. That framing is fair to the judge and honest about the ceiling, because you cannot ask an automated scorer to be more consistent with human judgment than human judgment is with itself.

And report the judge's agreement alongside every number the judge produces. A quality score of eighty-two percent, produced by a judge with a kappa of point three, is a number with a very large invisible error bar, and the people reading your dashboard have a right to know that.

A practical warning about judge prompts, because they rot in the same way production prompts rot and nobody watches them.

The judge prompt is a prompt. It has a version. It drifts. Somebody adds a clarification after a disagreement, and somebody else adds an example, and eighteen months later the judge is being asked to do something subtly different from what it was calibrated for, and the calibration statistic on the dashboard is from the old prompt.

So version the judge prompt, attach the version to every score it produces, and re-run the calibration whenever it changes. Treat it with the same discipline as the production prompt in Chapter Thirteen, for the same reason: it is an instrument, and an instrument whose configuration changed without a recalibration is producing numbers that mean something other than what they used to mean.

Using multiple judges

An ensemble reduces variance, and it costs what you would expect.

Three different judge models scoring the same case and voting will beat any one of them, particularly on the boundary cases where a single judge is essentially flipping a coin. It also triples your cost, which is why the sensible deployment is selective: use one judge everywhere, and escalate to an ensemble only when the single judge expresses low confidence or when the case is high severity.

Use different model families if you can. Three instances of the same model will share the same blind spots and will agree with each other confidently while being wrong together, which is the worst possible ensemble.

The disagreement rate between judges is itself a useful signal and I would put it on the dashboard. A case where three judges disagree is a case where your rubric is ambiguous, and routing those to a human accomplishes two things at once: you get the right answer for that case, and you get a queue of rubric defects sorted by how often they occur.

A boundary I would draw firmly, because the pressure to cross it is constant.

A judge model may never be the sole authority on a criterion where a failure is unrecoverable. Safety, policy compliance, and anything with a legal consequence get human review on a sample, permanently, regardless of how good the judge's agreement statistic looks. Say nothing about the judge's ability here. The concern is what happens when the judge is wrong in a correlated way across a whole category, which is a failure mode that no agreement statistic computed on a calibration set will catch, because the calibration set was scored before the correlated failure appeared.

The pattern to hold onto is that judges scale evaluation and humans anchor it. Remove the anchor and the whole apparatus will drift, slowly, in a direction nobody chose, and the drift will be invisible because the instrument that would have detected it is the thing that drifted.

Cost and latency tradeoffs

Judges are cheap compared to humans and expensive compared to nothing, and the arithmetic determines what your evaluation program can actually do.

Work an example. A two-thousand-case suite, scored on seven criteria, with separate judge calls per criterion, is fourteen thousand calls per run. Add pairwise double-running for position control and you are near twenty thousand. That is a real bill, per run, and if you wanted to run it on every pull request you have just made continuous integration cost more than the feature.

So tier it, and be explicit about the tiers. On every commit, run deterministic checks only: format, length, forbidden strings, tool-call assertions. Free, instant, and they catch the dumb regressions. On every pull request, run a fast suite of two hundred cases with a small judge model on three criteria. On every release candidate, run the full suite with the strong judge. Nightly, run the expensive claim-level groundedness decomposition on a sample. Weekly, run human review on a stratified sample of two hundred cases, which is what keeps everything above it honest.

Use a small model for the easy criteria. Format compliance does not need your most expensive judge, and the cost difference across a two-thousand-case suite is the difference between running it nightly and running it never.

And measure your judge's latency, because a judge that takes ninety seconds per case turns a two-thousand-case run into fifty hours, and a suite that takes fifty hours is a suite that runs monthly, and a suite that runs monthly will not catch the regression you shipped on Tuesday.

You now have datasets, rubrics, and a way to score at scale. Which means you have the machinery to answer the question that this entire discipline exists to answer, and that no team I have worked with could answer honestly before they built it: did the thing we shipped this week make the product worse?

Part Four

Regression Detection and Release Gates

Chapter 13: Prompt Regression Testing

Somebody added the word concise.

It went into the system prompt, in a sentence about response style, during a week when a designer had complained that answers were too long on mobile. It was a good change, made for a good reason, by a competent engineer, and it shipped on a Wednesday.

Answers got shorter. The designer was happy. Two weeks later, a support lead asked why the assistant had stopped mentioning the exchange option when customers asked about returns.

It had not stopped, exactly. It mentioned the exchange option about a third as often as before. The instruction to be concise had competed with the instruction to present all available remedies, and concision had won, because concision was in the style section which the model weighted heavily and remedies were buried in a list of policy requirements two hundred tokens down. The model was doing precisely what we told it. We had simply not understood what we told it.

No code changed. No model changed. No data changed. A single word entered a text file, and a product behavior that legal cared about degraded by two thirds, and nothing in our pipeline noticed.

Why prompt changes are risky

A prompt is the least protected and most powerful surface in a generative system, and the asymmetry is worth stating plainly.

It has the reach of code. It determines behavior across every request, in every intent, for every user. A one-line change to a system prompt has a blast radius that a one-line change to a service almost never has, because the service change affects a code path and the prompt change affects all of them.

It has none of the protections of code. No type system will catch a contradictory instruction. No compiler will warn you that a rule you added in March conflicts with one written in November. No linter will notice that your prompt has grown to four thousand tokens and that the instruction at token three thousand is being ignored. There is no unit test for a sentence.

And its effects are nonlocal in a way that defeats intuition. Adding an instruction about tone changes factual completeness. Moving an instruction changes whether it is followed at all. Adding a fifth rule can degrade compliance with the first four, because the model has a finite budget of attention and instructions compete for it. Nothing in a developer's training prepares them for a system where adding a requirement makes the existing requirements weaker.

Prompts also accumulate. Every incident produces a sentence. Nobody ever removes one, because removing a sentence that was added after an incident feels like inviting the incident back. After two years you have a document that nobody has read end to end, containing rules that contradict each other, that the model resolves by some process none of you can inspect.

Versioning prompts

Treat the prompt as source. This is the whole recommendation and most of the value.

It lives in version control. It goes through pull requests. It has an owner listed in a file. It has a changelog, and the changelog says what changed and why, so that in eighteen months somebody can find out whether the sentence about escalation was added on purpose.

Give it a version identifier that gets attached to every response it produces, and log it. This one detail transforms your ability to debug. When quality drops, the first question is what changed, and if every logged response carries the prompt version that produced it, you can answer it in a query. Without it you are reconciling deployment timestamps against a chat thread.

Keep the composed prompt, not just the template. The template renders against variables, and the rendered artifact is what the model saw. Snapshot it. A template that looks unchanged can produce a materially different prompt because an upstream variable started returning a longer value, and you will never find that by reading the template.

And version the fragments separately if your prompt is assembled from pieces, which it will be once several teams have a stake in it. A change to the safety block and a change to the persona block are different changes with different reviewers, and collapsing them into one file means every change requires everyone's approval, which means changes stop going through review at all.

Creating prompt test suites

A prompt test suite is a set of cases chosen specifically to exercise the instructions in the prompt, which is a different selection principle from the dataset in Chapter Four.

Read the prompt line by line. Every instruction in it is a claim about behavior. Every claim needs at least one case that would fail if the instruction were removed. This exercise takes an afternoon and it is the most useful afternoon in this chapter, because roughly a third of the instructions in a mature prompt turn out to have no observable effect at all, and finding that out is worth the price of admission on its own.

Then build cases for the interactions, which is where the failures actually live. Take the pairs of instructions that could compete, the ones about brevity and the ones about completeness, the ones about safety and the ones about helpfulness, and write cases that force the model to choose. The behavior at that boundary is the behavior you are shipping, and it is the behavior nobody has ever looked at.

Keep the suite fast. This one runs on every pull request that touches a prompt, which means it has to complete in minutes and cost very little. Two hundred cases with deterministic checks and a small judge on two or three criteria is the shape. The full dataset comes later in the pipeline.

And keep a case for every instruction that was added after an incident. Those are the ones with a story behind them, and they are the ones somebody will eventually try to delete for being verbose.

Comparing old vs. new prompt outputs

The comparison is paired, and the pairing is what gives you the sensitivity to see small changes.

Run the old prompt and the new prompt on the same cases, with the same retrieved evidence, with the same seed if your provider offers one. Score both. Compare per case rather than in aggregate, because the aggregate hides the interesting thing. What you want to see is the crosstab: how many cases went from fail to pass, how many went from pass to fail, and how many did not move.

That crosstab is the artifact. A change that fixes twelve cases and breaks nine has a net gain of three and a headline score that barely moves, and the nine broken cases are a completely different population from the twelve fixed ones, and somebody needs to look at them before this ships. An aggregate score of plus one point conceals all of that.

Read the regressions. Every single one, by hand, if there are fewer than twenty. This is not optional and it is the step that gets skipped under deadline. The nine broken cases are telling you what your change actually did, and they will frequently reveal that the change did something other than what you intended, which is the entire point of running the test.

Hold everything else fixed. Same evidence, same context, same model, same temperature. If you change the prompt and the retrieval configuration in the same pull request, you have learned nothing about either, and you will spend the next week arguing about which one caused the drop.

Measuring quality deltas

A number went from eighty-one to seventy-nine. Is that a regression?

The honest answer is that you cannot tell without knowing the noise, and most teams do not know the noise, and so they respond to random variation with engineering effort and miss real drops because they look like the last false alarm.

Characterize the noise before you interpret any delta. Run the same suite, against the same system, twice, changing nothing. The difference between those two runs is your floor, and any delta smaller than it is not information. Do this once a quarter and put the number somewhere visible, because everyone who reads your dashboard needs to know it.

Sampling noise is the second component and it is computable. A pass rate of eighty percent over two hundred cases has a standard error of about two and a half points, which means a two-point move is invisible and a six-point move is real. Over two thousand cases the standard error drops below one point. The suite size you need is determined by the size of the regression you want to catch, and if you have not done that arithmetic your suite is either too small to be useful or too expensive for no reason.

For paired comparisons, use the pairing. When you run old and new on the same cases, you can test the difference directly on the cases that changed, which is a much more sensitive test than comparing two independent proportions. McNemar's test is the standard tool for paired binary outcomes and it exists in every statistics library, and using it rather than eyeballing two percentages is a five-minute change that materially improves your ability to detect small regressions.

And apply a correction when you look at many metrics at once. If you check twenty criteria at a ninety-five percent confidence level, one of them will look significant by chance, and someone will spend a day chasing it. This is the multiple comparisons problem, it has standard fixes, and ignoring it is how evaluation programs acquire a reputation for crying wolf.

A discipline about prompt changes that is easy to state and painful to hold, and that will save you more debugging time than anything else in this chapter.

Change one thing. A pull request that moves an instruction, rewords two others, and adds a new rule is a pull request whose effect cannot be attributed. If the numbers move, you will not know which of the three did it, and if two of them helped and one hurt, the aggregate may show nothing at all while your product quietly gets worse in a way you shipped on purpose.

The objection is that this is slow, and it is, and it is slower than the alternative, which is a quarter of not knowing why your quality is drifting. Prompts are the most powerful and least instrumented surface in the system. They deserve the same one-change-at-a-time discipline that any other uninstrumented surface would get.

Regression thresholds

A threshold turns a delta into a decision, and it has to be set before the delta exists, because a threshold set after you see the number is not a threshold.

My default gate for a prompt change has four conditions and all of them must hold. No new critical failures, at all, ever, regardless of what the aggregate did. The count of major failures has not increased. The aggregate score has not dropped by more than the noise floor. And no individual criterion has dropped by more than a specified amount, which is the condition that catches the change that traded correctness for tone and left the average alone.

That fourth condition is the one people leave out and it is the one that would have caught the concise change from the opening of this chapter. The aggregate barely moved. The completeness criterion fell off a cliff. Only a per-criterion threshold sees that.

Set the thresholds by consequence rather than by convenience. If your current numbers cannot clear the bar that the product requires, that is a finding about the product rather than an argument for a lower bar. Lower it if you must, deliberately, with a name attached and a date to revisit, and never quietly.

Prompt rollback strategy

You will ship a bad prompt. The only question is how long it stays out.

Decouple the prompt from the deployment. If changing a prompt requires a code release, your rollback time is your release time, and your release time is measured in hours on a good day. If prompts are configuration, served from a store, referenced by version, then rollback is a pointer change and it takes seconds. Build this before you need it.

Keep the previous version warm and keep the ability to run both. A canary that routes five percent of traffic to the new prompt, with an automatic rollback armed on a guardrail metric, converts a bad prompt from an incident into a non-event. Chapter Sixteen builds the mechanism.

And write down what triggers a rollback before you deploy, because the argument at two in the morning about whether a metric has moved enough is an argument you will lose. A safety violation rate above baseline rolls back automatically. A quality drop beyond the threshold rolls back automatically. Everything else is a judgment call and it belongs to a named person, and the name should be in the deployment plan.

A structural recommendation that pays for itself within a quarter: split the prompt into blocks with separate owners.

A single monolithic prompt has a single owner, which in practice means it has no owner, because six teams have a stake in it and each of them edits it and none of them read the whole thing. Split it. A safety block owned by safety. A policy block owned by policy. A persona block owned by design. A task block owned by the feature team. Each block is a file, each file has an owner in a code-owners configuration, and each change requests the right reviewer automatically.

The benefit has little to do with tidiness. What you gain is that a change to the persona no longer requires the safety owner's approval, which means changes go through review rather than around it, and the reviews that matter get the attention of the people who can actually judge them.

Prompt review process

The last piece is human and it is the one that decays fastest.

Every prompt change gets a reviewer, and the reviewer is not always an engineer. A change to the safety block wants somebody from policy. A change to the persona wants a designer. A change to the refusal language wants legal, in some domains, and the fact that this sounds bureaucratic does not make it wrong; it makes it a workflow you should automate with a code-owners file so that the right reviewer is requested without anybody having to remember.

The pull request template asks four questions. What behavior is this change meant to produce. Which existing instructions might it compete with. What test cases would fail if this change were reverted. And what does the paired comparison show.

That third question is the one that does the most work. An engineer who cannot name a test case that depends on their instruction has not thought about whether the instruction does anything, and about a third of the time it does not.

And schedule a prompt audit. Once a quarter, somebody reads the whole prompt, end to end, out loud if necessary, and asks of every sentence whether it still earns its place. Delete the ones that do not, in a pull request, with the test suite as the safety net. A prompt that has never been shortened is a prompt that is quietly getting worse, and the test suite you built in this chapter is what makes shortening it safe.

Stage	Action	Blocking condition
Author	State intended behavior, competing instructions, and the test case that depends on this change	Template incomplete
Render	Snapshot the composed prompt; assert no empty placeholders, token budget respected	Render assertion fails
Pair run	Old prompt and new prompt on identical cases, identical evidence, identical seed	Any variable other than the prompt changed
Crosstab	Report fail-to-pass, pass-to-fail, and unchanged counts	Regressions not reviewed by a human
Read regressions	Manually read every case that went from pass to fail	Skipped
Statistics	Paired test on changed cases; correct for multiple criteria	Delta inside the noise floor and claimed as a win
Gate	No new critical; majors not up; aggregate within threshold; no single criterion down beyond its limit	Any condition unmet
Review	Owner from the affected block approves via code owners	Missing required reviewer
Deploy	Prompt as versioned configuration, not code; previous version kept warm	Rollback would require a code release
Canary	Small traffic share, guardrails armed, auto-rollback configured	No rollback trigger defined
Observe	Prompt version logged on every response for the life of the deployment	Version not attached to responses
Audit	Quarterly read-through; delete instructions with no dependent test	Prompt has only ever grown

One prompt, one word, two thirds of a legal requirement. That is how much a single sentence in this surface can move, in both directions, which is why it deserves the discipline in that table and why almost nobody applies it.

Prompts are the change you make most often. The change you make least often, and dread most, is the one where the thing underneath the prompt gets replaced entirely.

Chapter 14: Model Upgrade Evaluation

The new model was better on every public benchmark. It scored higher on reasoning, higher on knowledge, higher on instruction following, and it was cheaper per token. The decision looked like it had already been made for us.

We ran it against our own suite and it lost.

Not by much, and not everywhere. It was better at reasoning through a complicated policy question, noticeably so, and it wrote more naturally. It was worse at three things that happened to be the three things our product did all day. It was more verbose, which broke a display constraint. It was more likely to add a helpful caveat that our legal team had spent a year removing. And it was slightly, consistently worse at resolving ordinal references in multi-turn conversation, which was the single failure mode we had spent two quarters fixing.

The benchmarks had told the truth. They were measuring general capability across a broad distribution of tasks, and the new model genuinely had more of it. Our product does not run on a broad distribution of tasks. It runs on a narrow one, and on that narrow one, the model we already had was better, and we would never have known if we had trusted the leaderboard.

Why model benchmarks are not enough

Public benchmarks are useful and they answer a question you are not asking.

They ask how capable a model is in general. You need to know how well it does the specific thing your product does, on your evidence, under your instructions, within your latency budget, in your users' phrasing. Those two questions have different answers often enough that treating one as a proxy for the other is a coin flip with a good story attached.

There is also a contamination problem that gets worse every year. Benchmarks are public, they end up in training data, and a model that has seen the test set will do well on it for reasons that have nothing to do with capability. Nobody knows the extent of this for any given model, and the people who would know are not incentivized to find out. It is a reason to hold public numbers loosely, and it is not the main reason.

The main reason is distributional. A benchmark averages across tasks. Your product is one task, and it is not the average one, and the model that wins on average can lose on yours. Stanford's HELM work made the general version of this argument, that a single aggregate score hides the scenarios where a model does badly, and the version that applies to you is sharper: the scenario where it does badly might be the only scenario you care about.

So a benchmark is a filter, and that is all. It tells you which models are worth the cost of a real evaluation. Then you do the real evaluation.

Offline model comparison

The comparison has to be controlled, and the control that matters most is the one teams break first.

Hold the context fixed. Take the exact assembled contexts your production system produced, the ones you logged in Chapter Three, and replay them against the candidate model. Same evidence, same history, same instructions. Anything else and you are comparing systems rather than models, and the difference will be attributed to the model, and the attribution will be wrong.

This is why the logging discipline pays off here more than anywhere else. A team that logged only queries and responses cannot run this comparison at all. They will have to re-retrieve, and the index has moved, and now the two models are seeing different evidence, and the entire experiment is worthless.

Run the paired crosstab from the last chapter. Fail to pass, pass to fail, unchanged. Read every regression. The aggregate will tell you the new model is one point better and the crosstab will tell you it broke a category, and the category is what you make the decision on.

And run a fair prompt. Your production prompt was tuned, over months, against your current model, and it encodes accommodations for that model's specific weaknesses. A candidate model evaluated against that prompt is being tested on a course designed for its predecessor. So run it twice: once with the production prompt, which tells you what happens if you swap the model and nothing else, and once with a prompt lightly adapted to the new model, which tells you what the model could do if you invested. Those two numbers are both real and they answer different questions. The first is the cost of a drop-in swap. The second is the value of a migration.

Cost, quality, and latency tradeoffs

Three numbers, and a model upgrade almost never improves all three.

Price per token is the number on the vendor's page and it is the one that misleads. What you pay is price per token times tokens used, and a model that is thirty percent cheaper per token and forty percent more verbose costs you more. Measure tokens on your traffic, not on the vendor's example. Reasoning models make this worse and more interesting, because a model that thinks before answering can consume many times its visible output in tokens you pay for and never see.

Latency has to be measured at the tail. A model with a better median and a worse ninety-ninth percentile will feel worse, because the ninety-ninth percentile is where your timeouts live and where your angriest users are. Measure time to first token separately from total generation time, because for a streaming interface the first is what the user experiences as speed.

And quality has to be measured per criterion, because the whole point of the exercise is that models trade. The new model is better at reasoning and worse at brevity. Better at safety and more prone to over-refusal. Better at following complex instructions and worse at following simple ones, which happens more often than you would think.

The decision is a business decision and your job is to make it a legible one. Put the three numbers next to each other, with the per-criterion breakdown underneath, and let the person who owns the budget choose. A team that presents only a quality delta is asking to be second-guessed.

Behavior differences across models

Beyond scores, models have personalities, and the personality will break things that no metric captures until you look.

Verbosity is the most common and the most disruptive. A model that writes twice as much will break your display constraints, blow your token budget, slow your responses, and, if you use an LLM judge, quietly inflate your quality scores, because judges reward length. That last one is worth reading twice. Migrating to a more verbose model can raise your measured quality while lowering your actual quality, and the mechanism is a bias in your instrument.

Format adherence differs. A model that reliably produced clean JSON may be replaced by one that wraps it in a code fence, or adds a preamble, or occasionally emits a trailing comma. Your parser will break, and it will break on a small percentage of requests, which is the worst kind of breakage.

Hedging differs. Some models qualify everything. If your product needs a direct answer, a model that says it depends on your specific circumstances is worse than useless even when the underlying reasoning is superior.

And instruction interpretation differs in ways that are genuinely hard to predict. An instruction that one model reads as a hard constraint another reads as a suggestion. The only way to find these is to run your prompt test suite from Chapter Thirteen against the candidate and read what comes out, and the reading is the part that cannot be automated.

A practical detail about running the comparison that will otherwise cost you a week of confused results.

Match the sampling parameters, and then check that matching them meant anything. Temperature, top-p, and the presence or absence of a system-level reasoning mode are not portable between model families, and the same nominal temperature can produce noticeably different variance in two different models. A comparison run at a temperature that is aggressive for one model and conservative for the other is a comparison of two different experiments.

The safest approach is to run each model several times per case and compare distributions rather than single samples. It costs more and it is the only way to distinguish a model that is genuinely better from a model that happened to get a good draw on the cases you looked at.

Safety differences

Safety behavior changes across models in both directions and both directions are a problem.

A model that refuses less will let through things your previous model caught, and your safety layer, which was tuned against the old model's behavior, will have holes it did not have last week. Run your entire adversarial suite against the candidate. All of it. This is the one part of the migration evaluation I would not sample.

A model that refuses more will decline requests it should answer, and over-refusal is invisible to a safety metric that counts only unsafe outputs. It shows up as a drop in task completion and a rise in escalation, weeks later, and somebody will blame retrieval. Measure the refusal rate explicitly, on a set of cases that should be answered, and treat an increase as a regression.

Jailbreak resistance differs, and the attacks that work on one model often do not work on another and vice versa. Your adversarial suite is a list of attacks that worked against the old model, which means it is, to some degree, an artifact of the old model. Generate fresh attacks against the candidate rather than only replaying the archive.

And the refusal style matters. A model that refuses abruptly, without explaining, produces a worse product than one that declines and offers an alternative, and no safety metric distinguishes them. That belongs in the rubric as refusal quality, and Chapter Twenty-Three treats it properly.

Multi-turn differences

This is where the differences are largest and where evaluation is thinnest, which is an unfortunate combination.

Long-context handling varies enormously between models with identical advertised context windows. Two models both claiming two hundred thousand tokens can differ by a great deal in how reliably they use information from the middle of that window, and the advertised number tells you the size of the buffer rather than the quality of the attention. Run the depth ladder from Chapter Ten and find out where each model actually degrades.

Instruction persistence varies. Some models hold a system instruction across twenty turns. Others drift, gradually reverting to their default persona as the conversation grows, which produces the specific failure where an assistant that was formal at turn one is chatty at turn fifteen.

Reference resolution varies, and it varies a lot, and it is not on any benchmark. This was the one that decided our migration. Run the ordinal and reference cases from Chapter Ten explicitly and compare, because a model that resolves the second one correctly ninety-five percent of the time and one that does it eighty-five percent of the time will feel like two completely different products, and no aggregate score will show you the gap.

Tool-use differences

If your system calls tools, this section decides the migration, because a model that is slightly worse at prose is an annoyance and a model that is slightly worse at arguments is an incident.

Tool selection accuracy differs, and it differs most on the hard cases: whether to call a tool at all, and which of two similar tools applies. Run the confusion matrix from Chapter Eleven for both models and compare cell by cell rather than comparing accuracy numbers.

Argument construction differs, and the derived arguments are where it shows. Dates computed from relative expressions, identifiers resolved from earlier turns, quantities inferred from context. A model that is one percent worse at these is a model that will corrupt one call in a hundred, and if those calls move money, the arithmetic is not in your favor.

Parallel and sequential calling behavior differs. Some models issue several tool calls at once; some chain them. Your orchestration layer probably assumes one of those patterns, and a model that adopts the other will produce behavior your infrastructure has never seen.

And the recovery behavior differs, which is the thing you will find last and which will hurt most. Give both models a tool error and see what they do. One will report it. One will try again. One will produce a confident answer as if the tool had succeeded, and that one is disqualified regardless of how it scored on everything else.

A subtlety about prompt portability that decides how expensive a migration actually is, and that teams discover late.

Your production prompt is not neutral. It encodes years of accommodations for the specific weaknesses of the model you have been running. An instruction that exists because the old model kept forgetting to cite. A rephrasing that exists because the old model misread a particular construction. A rule added after an incident that the new model would never have caused.

Some of those accommodations will be inert against the new model and some will actively hurt, by consuming attention budget on instructions that address problems the new model does not have. That is why the two-prompt evaluation matters. The gap between the score with your production prompt and the score with a lightly adapted one is a measurement of how much technical debt is sitting in your prompt, and it is often larger than the difference between the two models.

Migration strategy

Nobody flips a switch. The migration is a rollout and it looks like every other risky rollout you have done.

Shadow first. Run the candidate on live traffic, in parallel with production, and discard its outputs. It costs money and it costs nothing else, and it gives you a comparison on real traffic rather than on your sampled dataset, which will surface the intents your dataset underrepresents. Run it for a week. Read the biggest divergences.

Then canary, on a small share of traffic, in a segment where the consequences of a bad answer are lowest, with the guardrails armed and the rollback automatic. Watch the online metrics rather than the offline ones, because the offline ones already told you what they know.

Then ramp, slowly, with a hold at each step long enough for a quality signal to accumulate. And if you can, hold two models in production simultaneously and route by intent, sending the high-consequence traffic to whichever model is better at it. Model choice as a routing decision rather than a global setting is a strategy that many teams eventually reach and few plan for, and building for it early costs almost nothing.

One more axis that belongs in the decision and that engineering evaluations routinely omit.

Provider risk. A model you depend on is a dependency with a support life, a rate limit, a pricing schedule, and a company behind it that will make decisions in its own interest. Deprecation notices arrive with timelines you did not choose. Capacity gets constrained during a demand spike. Terms change. None of this appears on a quality scorecard and all of it can end a product.

So evaluate the second-best model too, and keep the evaluation current, even when you have no intention of migrating. The cost is a few hours a quarter. The benefit is that when the notice arrives, you already know what your options are, and you are having an engineering conversation rather than an emergency.

Rollback planning

Write the rollback plan before the migration, because a rollback plan written during an incident is a decision made by whoever is most awake.

The plan names the triggers. Safety violation rate above baseline. Quality below the gate. Latency at the ninety-ninth percentile above the budget. Escalation rate up beyond a threshold. Each of these is monitored, each has a number, and the ones that can be automated should be automated.

It names the mechanism, which is a routing change rather than a deployment, so that rollback takes seconds. And it names the person, because at least one trigger will be a judgment call and the judgment should belong to somebody in advance.

And it accounts for what does not roll back. If you adapted your prompt to the new model, rolling back the model without rolling back the prompt puts you in a state that has never been tested. Version them together. If you migrated your evaluation baselines, rolling back the model leaves you comparing against the wrong baseline. Roll those back too. The state to which you are returning has to be a state that existed.

Dimension	Measured how	Decision weight
Accuracy	Paired crosstab on cached contexts, per criterion	High
Groundedness	Claim-level support rate on identical evidence	High

Dimension	Measured how	Decision weight
Instruction following	Prompt test suite, per instruction	High
Safety	Full adversarial suite, plus freshly generated attacks	Veto
Over-refusal	Refusal rate on cases that should be answered	High
Multi-turn carryover	Depth ladder at 4, 6, 8, 12 turns	High
Reference resolution	Ordinal and attribute reference cases	High
Tool selection	Confusion matrix over the tool catalog	Veto if tools are live
Argument correctness	Derived arguments: dates, ids, amounts	Veto if tools are live
Tool error recovery	Fault injection; fabrication is disqualifying	Veto
Format adherence	Parse rate of structured outputs	High
Verbosity	Median and p95 output tokens on real traffic	Medium
Latency	Time to first token and end-to-end, p50 and p99	High
Cost	Tokens used on real traffic times price, including hidden reasoning tokens	High
Judge bias check	Correlation of judge score with response length	Diagnostic
Prompt portability	Score with production prompt vs. lightly adapted prompt	Diagnostic
Shadow divergence	Largest disagreements against production on live traffic	Diagnostic

The judge bias row is the one I would not skip. If your judge rewards length and your new model is longer, your scorecard will tell you the migration is a win, and it will be lying to you, and the mechanism will be sitting inside your own evaluation harness.

Prompts change and models change. There is a third thing that changes constantly, that nobody deploys, that has no version number, and that is responsible for more bad answers than either.

Chapter 15: Retrieval and RAG Regression Testing

The quality score dropped four points on a Tuesday. Nobody had deployed anything.

We checked the prompt version: unchanged. We checked the model: unchanged. We checked the service: no releases that week, no configuration changes, no infrastructure events. Every artifact under version control was byte-for-byte identical to the previous Tuesday, when the score had been four points higher.

What had changed was the index. A content team, doing their job, had republished a set of policy documents with a new template. The template added a site header to every page. The header contained the phrase frequently asked questions, which now appeared at the top of every chunk, which shifted the embedding of every chunk slightly toward the neighborhood of every user question that resembled a frequently asked question, which changed the ranking, which meant a different set of passages came back for a large family of queries.

Nobody deployed anything. A team we did not know existed changed a template, and our quality dropped, and the artifact responsible had no version number and was not in our repository and did not appear in any changelog we monitored.

This is the defining property of retrieval as a failure surface. It is the largest source of bad answers in most production systems, and it is the one that moves without anyone touching your code.

Retrieval quality vs. answer quality

Start with the diagnostic split, because it is the single most useful thing in this chapter and most teams have never run it.

Take every failing case. For each, ask one question: was the information needed to answer this correctly present in the retrieved evidence? Yes or no.

That question partitions your failures into two populations with entirely different owners. The no pile is a retrieval problem. The model was asked to answer a question without being given the answer, and it did the best it could, which was not good enough, and no amount of prompt engineering or model upgrading will fix it. The yes pile is a generation problem. The model had what it needed and got it wrong anyway.

In every retrieval-augmented system I have examined closely, the no pile is larger. Frequently much larger. Teams are consistently surprised by this, and the surprise is expensive, because they have usually spent a quarter tuning prompts before anyone ran the split.

The check is cheap. A human can do it in ten seconds per case. A model can do it for a fraction of a cent, and it is one of the tasks judges are reliably good at, because it is entailment against a visible artifact. Build it, run it on every failure, and report the split as a first-class metric on your dashboard. It will change what your team works on.

Recall and precision in RAG

Retrieval has metrics that predate all of this and they still work, and using them requires the one thing nobody wants to do, which is to label which passages are actually relevant to which queries.

Recall at k asks whether the passage that answers the query appeared anywhere in the top k results. It is the metric that matters most, because a passage that was not retrieved cannot be

used, and everything downstream is bounded by it. Recall is your ceiling.

Precision asks how much of what you retrieved was worth retrieving, and it matters more than people expect. Retrieving twenty passages when three are relevant does not just waste tokens. It buries the relevant ones among distractors, and models are demonstrably worse at using evidence that arrives surrounded by irrelevant material. Precision is a quality concern rather than a cost concern.

Rank position matters because of where models attend. Nelson Liu and colleagues showed in Lost in the Middle that a passage placed in the middle of a long context is used less reliably than the same passage placed at the beginning or the end. The consequence for evaluation: recall at ten is not enough. You need to know where in the ten the right passage landed, which is what mean reciprocal rank measures.

Building the labels is the work. Take three hundred queries from production, have a human mark which passages in the corpus actually answer them, and freeze the result. It takes a week. It is the foundation of every retrieval metric you will ever compute, and a team that skips it will spend years unable to tell whether their retrieval is good.

Evaluating chunking strategy

Chunking is the least examined decision with the largest effect, and it is usually made once, early, by whoever set up the pipeline, based on a default.

The failures are structural. A chunk that splits an answer across a boundary makes the answer unretrievable, because neither half contains it. A chunk that is too large dilutes its own embedding, so it matches everything weakly and nothing strongly. A chunk that is too small retrieves cleanly and lacks the context to be interpreted, which produces the out-of-context quote from Chapter Nine: a passage that says the right words under a heading the chunk does not contain.

That last one is worth dwelling on because it is the failure that produces confidently wrong cited answers. A section heading three paragraphs up establishes that everything below applies to commercial accounts. The chunk starts below the heading. The model reads it, quotes it accurately, and applies it to a consumer, and the citation checks out.

The fix is to carry context into the chunk. Prepend the document title, the section path, and the applicability scope to every chunk before embedding it. Anthropic published work on contextual retrieval describing a version of this, generating a short situating description for each chunk and prepending it, with substantial gains in retrieval accuracy. The general principle is older than the technique: a chunk that does not know what document it came from is a chunk that can be quoted into a lie.

Evaluate chunking directly. Hold everything else fixed, vary the chunk size and the overlap and the context injection, and measure recall on your labeled set. It is a sweep, it takes a day, and it routinely produces larger gains than a model upgrade.

Evaluating ranking

Ranking is where you spend a little compute to fix what retrieval got approximately right.

The first thing to measure is whether your retriever's own ordering is any good, which it usually is not. Embedding similarity is a blunt instrument. It captures topical relatedness and it does not capture whether a passage answers a question, and those come apart constantly: a passage about

the return policy is topically perfect for a return question and completely useless if it is the return policy for a different product line.

A cross-encoder reranker is the standard fix and it is one of the highest-return components you can add. It reads the query and the passage together and scores their relevance, which is a harder computation than comparing two vectors and a much better one. Retrieve fifty, rerank, keep five. Measure recall before and after and you will see the gap.

Hybrid retrieval is the other standard fix. Dense retrieval misses exact matches, which is a serious problem for product identifiers, error codes, and policy names, where the user typed the exact string and the embedding shrugged. Keyword retrieval catches those and misses paraphrase. Run both and fuse the results, and evaluate the fusion rather than assuming it helped, because the fusion weights matter and the defaults are arbitrary.

Evaluate ranking with position-sensitive metrics. Recall at five is blind to ordering within the five. Normalized discounted cumulative gain is the standard tool and it rewards putting the best passage first, which, given what we know about attention over long contexts, is exactly what you want it to reward.

Staleness detection

The index is a cache of the world and caches go stale, and yours will go stale silently in the places nobody is watching.

Two clocks matter and they are different. Index lag is the time between a source document changing and the index reflecting it. Content lag is the time between the world changing and the source document reflecting it, and it is usually much larger and it is owned by somebody who does not report to you. A pipeline that reindexes every hour is doing nothing for a policy page that was last updated in March and is now wrong.

Measure the freshness distribution rather than the average. The average age of a document in your corpus is a comforting number and it hides the fact that your highest-traffic policy page has not been touched in fourteen months. Report the age of the documents that back your top intents, which is a much smaller number to compute and a far more useful one.

Build canary documents. Pick a small set of facts you can verify externally, at low cost, on a schedule. Query for them. If the system returns the old value, your index is stale and you know it before a customer does. This is a few dozen lines of code and it is the cheapest early warning in the entire book.

And detect the reverse case, which caused the incident at the top of this chapter. Monitor the corpus for change. If ten thousand documents were rewritten last night, you want to know, and you want to know before your quality score tells you. A simple count of documents modified per day, on a dashboard, would have saved us a week.

A cheap fix for a common retrieval failure that most teams try to solve with a better embedding model.

Query rewriting. The user types a fragment. The retriever embeds the fragment. The fragment carries almost no signal, so the retrieval is poor, and the answer is poor, and the diagnosis is that retrieval needs work. What retrieval needed was a query.

Rewriting the user's turn into a self-contained question, using the conversation history, before it reaches the retriever, is a small model call and it frequently produces a larger improvement than any change to the index. Evaluate it as a component: take the rewritten queries, hand them to a

human, and ask whether they capture what the user meant. Where they do not, you have found a failure that was being attributed to search and belongs to context.

Source conflict handling

Two documents disagree. What happens?

In most systems, nothing happens, in the sense that nothing was designed to happen. Both passages go into the context, the model reads both, and it resolves the conflict by some process nobody specified, usually by preferring whichever it read last or whichever sounds more authoritative in a way that has no relationship to which one is actually authoritative.

That is a decision being made, arbitrarily, thousands of times a day, by a component that was never asked to make it.

The design fix is precedence, and it belongs in the data rather than the prompt. Every document carries a source authority level and an effective date. When passages conflict, the pipeline resolves the conflict before the model sees it, or, failing that, the prompt tells the model the precedence rule explicitly and the model is told which passage wins.

The evaluation is a set of cases with conflicting evidence deliberately injected. An old policy and a new one. A general rule and a category exception. A regional variation and a global default. Assert that the system picks the right one, and assert on what it does when it cannot: a system that surfaces the ambiguity and asks is behaving well, and a system that picks silently is a system that will be wrong half the time, confidently, forever.

Citation correctness

Chapter Nine covered citations from the generation side. From the retrieval side there is a different set of failures and they are mostly plumbing.

The identifier survives the trip, or it does not. A chunk is retrieved with a document identifier. That identifier passes through the context builder, into the prompt, out of the model, through post-processing, and into a link the user can click. Every hop is a place it can be dropped, mangled, or reattached to the wrong claim. Test the whole path with an assertion at the end: does the link, clicked, land on the passage that supports the claim.

Post-hoc citation is the failure mode that produces the most confident-looking nonsense. Some pipelines generate the answer first and then attach citations by matching the answer text against the corpus. This produces citations that look perfect and support nothing, because the matcher found a passage with similar words rather than a passage that grounds the claim. If your system does this, evaluate it as a distinct component, and be prepared to find that a large fraction of your citations are decorative.

And measure coverage, meaning the fraction of claims that need a citation and have one. A response with two citations and six uncited claims is worse than one with none, because the two citations tell the reader that this is a system that cites its sources, which is an invitation to trust the six.

A cheap experiment that tells you the ceiling of everything else you could do to retrieval, and that takes an afternoon.

Hand the model the correct passage. Not the retrieved passages, the correct one, chosen by a human, for each case in a small labeled set. Then score the answers. That number is the ceiling: it is how well your system would perform if retrieval were perfect.

If the ceiling is ninety-five and you are at seventy, retrieval is your problem and everything you spend on prompts and models is spent on the last five points. If the ceiling is seventy-five and you are at seventy, retrieval is nearly saturated and further work there will produce nothing, and the problem is in generation. I have watched teams spend a quarter tuning a retriever that was already close to its ceiling, and this experiment would have told them so on the first day.

Retrieval coverage metrics

There is a class of failure where retrieval is working perfectly and the answer is still impossible, and it is worth separating out because it belongs to a different team.

The corpus does not contain the answer. The user asked something reasonable, the retriever did its job, and there is nothing in the index that answers it, because nobody ever wrote it down. This is a content gap, and no amount of retrieval tuning will close it, and if you file it as a retrieval bug it will sit in the backlog forever.

Measure it. For every failing case in the retrieval pile, ask a second question: does the answer exist anywhere in the corpus? If yes, retrieval failed. If no, content failed. That second split is what turns a vague complaint about quality into a prioritized list of documents somebody needs to write, and handing that list to a content team is one of the more satisfying things an evaluation program can do.

Track corpus coverage by intent, and track it over time. As traffic shifts into new areas, coverage in those areas will be poor, and the coverage metric will show it before the quality metric does, because it is measuring the cause rather than the symptom.

A word about the labeled relevance set, because it is the one piece of work in this chapter that people reliably refuse to do.

Every retrieval metric in this chapter requires knowing which passages are actually relevant to which queries, and that knowledge only comes from a human sitting down with three hundred queries and marking passages. It takes about a week. It is boring. And every team I have suggested it to has responded by proposing something cleverer: using the model to generate the labels, using click data as a proxy, using the answers to infer the evidence.

Each of those shortcuts introduces exactly the bias you are trying to measure. A model-generated relevance label is a measurement of your model's opinion, taken with your model. Click data measures what was shown rather than what was needed. Do the week. It is the foundation of every retrieval number you will ever quote, and there is no substitute for it.

End-to-end RAG evaluation

Put it together. A retrieval-augmented system is evaluated at three levels and you need all three, because each one tells you something the others cannot.

The retrieval level tells you whether the evidence was available. Recall, precision, rank position, freshness, coverage. Fast, cheap, and it bounds everything above it.

The context level tells you whether the evidence survived assembly. Was the retrieved passage still in the context after truncation, and where did it land in the ordering. This is the level almost nobody instruments and it is where a surprising number of failures hide: the right passage was retrieved and then dropped to make room for conversation history.

The answer level tells you whether the model used it. Groundedness, correctness, citation accuracy. Expensive, and meaningless without the two levels below it, because an answer-level

failure with no retrieval-level context is a mystery.

The frameworks that have emerged in this space, Ragas from Shahul Es and colleagues among them, organize their metrics along roughly these lines, separating context precision and context recall from faithfulness and answer relevance. The specific tooling matters less than the separation. A single end-to-end score for a retrieval-augmented system is a number that tells you something is wrong and never tells you where.

Level	Metric	What a drop means
Retrieval	Recall at k on the labeled set	The evidence is not being found; everything above is capped
Retrieval	Precision at k	Distractors are crowding out the signal
Retrieval	Mean reciprocal rank	The right passage is being found and buried
Retrieval	Corpus coverage by intent	A content gap, not a retrieval bug
Retrieval	Document freshness at p90 for top intents	The index reflects a world that has moved
Retrieval	Corpus change volume per day	Something upstream is rewriting your data
Chunking	Recall sweep over chunk size and overlap	The current chunking is splitting answers
Chunking	Chunk context completeness	Chunks lack the scope needed to be interpreted
Ranking	NDCG before and after reranking	The reranker is not earning its latency
Ranking	Exact-match recall on identifiers and codes	Dense retrieval alone; hybrid search is missing
Context	Survival rate of the top passage after assembly	Truncation is discarding the evidence you retrieved
Context	Position of the top passage in the final context	The best evidence is landing in the worst position
Conflict	Correct precedence on injected conflict cases	Conflicts are being resolved arbitrarily by the model
Answer	Claim support rate against supplied evidence	The model is going beyond its evidence
Answer	Citation resolution and support rate	Citations are decorative rather than checkable
Split	Failures with evidence present vs. absent	Tells you whether to fix retrieval or generation

The last row is the one to build first. Everything else in that table is refinement, and the split between failures where the evidence was present and failures where it was absent is the number that will redirect your team's effort more than any other measurement in this book.

You now know how to detect regressions in prompts, in models, and in retrieval. Which leaves the question of what happens when you detect one, and specifically what happens when you detect one at four in the afternoon on the day of a launch that has been on a roadmap for two quarters.

Chapter 16: CI/CD for LLM Systems

The launch was scheduled for Thursday. On Wednesday afternoon, the evaluation run came back with a two-point drop on the completeness criterion and one new critical failure.

There were nine people on the call. The engineer who owned the change explained that the critical failure was probably a scoring artifact. The product manager pointed out that the launch date had been communicated to a partner team. Somebody suggested we re-run the suite, on the grounds that these things fluctuate. Somebody else suggested that one critical failure out of two thousand cases was, statistically speaking, nothing.

We shipped on Thursday.

The critical failure had nothing to do with scoring artifacts. It was a real defect in a policy answer, it affected a small fraction of a high-consequence intent, and it took eleven days to surface, and when it surfaced it did so in a form that required a written explanation to somebody senior.

The lesson is not that we made a bad decision under pressure, although we did. The lesson is that we had built a system in which that decision was ours to make. We had a measurement and a meeting, which means we had an opinion. What we did not have was a gate.

Evaluation as a release gate

A gate is a mechanism that stops a bad change automatically, without requiring anyone to be brave.

That last clause is the entire argument. Every organization has people who will raise a concern in a meeting. Very few have people who will block a launch that a director has already told a partner is happening. Human judgment under schedule pressure reliably fails in one direction, and it fails in that direction in every industry, and the only thing that has ever worked is to make the failure automatic and the override explicit.

So the gate blocks by default. The pipeline stops. To proceed, somebody with a name has to override it, and the override is recorded, and the record includes the reason and the risk accepted. This is not bureaucracy for its own sake. It changes behavior, because the person overriding knows their name will be on the incident review, and that knowledge is the mechanism.

The gate must also be able to fail. A gate that has never blocked a release is not calibrated, it is decorative, and if yours has passed every release for six months then either your team is extraordinary or your thresholds were set to pass. Assume the second. It is nearly always the second.

And the gate has to be fast enough to live in the pipeline. A quality check that takes six hours will get moved to a nightly job, and then to a weekly one, and then to a thing somebody runs before a big launch, and then to a thing nobody runs. Speed is what keeps a gate alive, which is why the tiering in the rest of this chapter matters as much as the thresholds.

Offline test suites

Everything before production is offline, and it is where you catch the majority of what you are going to catch, cheaply.

Tier it by cost, and be deliberate about what runs where. Deterministic checks are free and instant: format compliance, forbidden strings, length bounds, tool-call schema assertions, prompt render assertions. These run on every commit and they catch the stupid regressions, and there are more stupid regressions than anyone admits.

The fast suite runs on every pull request. A few hundred cases, a small judge model, two or three criteria, completing in minutes. It is a smoke test, its job is to catch obvious quality damage before a human reviews the change, and its pass rate is not a metric anybody should be reporting to leadership.

The full suite runs on release candidates and nightly. The whole dataset, the strong judge, every criterion, every stratum, reported per stratum and per severity. This is the number that gates, and it is the number on the dashboard.

And the expensive suite runs on a sample, on a schedule. Claim-level groundedness decomposition, human review of a stratified sample, the anchor set from Chapter Eight that tells you whether your reviewers have drifted. This layer is what keeps the layers above it honest, and it is the one that gets cut when budgets tighten, and cutting it means the rest of your pipeline is measuring something nobody has checked in a year.

Pre-merge evaluation

The most valuable position for a quality signal is inside the pull request, next to the diff, where the author is still holding the context in their head.

Post a comment on every pull request that touches a prompt, a retrieval configuration, or a model reference. The comment shows the paired crosstab: cases that went from fail to pass, cases that went from pass to fail, and the aggregate delta with its confidence interval. Link every regressed case to its full trace, so a reviewer can click through and read what happened.

That comment changes the conversation. Without it, code review of a prompt change is two engineers reading a sentence and having opinions about it. With it, review is a discussion of eleven specific responses that got worse. The second conversation is short and productive. The first one is a matter of taste.

Make it fast, because a check that takes twenty minutes will be ignored while the author moves on to something else, and by the time it comes back they have lost the thread. Under five minutes is the target, and hitting it means a small suite and a cheap judge, and that tradeoff is worth making.

Canary evaluation

Offline evaluation is a prediction. The canary is the first measurement.

Route a small share of traffic to the new version. One percent, five percent, whatever your volume supports. Watch the guardrail metrics, which are the ones that can be measured in real time and that indicate harm: safety intervention rate, latency at the tail, error rate, tool failure rate, refusal rate.

Do not try to measure quality in a canary, at least not the way you measure it offline. Quality signals from production are slow, noisy, and confounded, and a five-percent canary running for four hours will not give you a statistically meaningful read on groundedness. That is what the offline suite was for. The canary is checking for the things offline evaluation cannot see: a latency regression under real load, a spike in refusals from a user population your dataset underrepresents, an error class that only appears at concurrency.

Arm the rollback before you start, and let it fire without a human. A canary with a manual rollback is a canary that will stay out for twenty minutes while somebody is found, and twenty minutes at five percent of a large product is a lot of people.

And watch for the sample ratio mismatch, which is the check that catches a broken experiment before it produces a wrong conclusion. If you routed five percent of traffic and your canary saw four point one percent, something is wrong with the routing, and every number from that canary is suspect. Ron Kohavi and colleagues make this point forcefully in their work on trustworthy online experiments, and it applies here exactly: check that the experiment ran before you interpret it.

Human approval workflows

Some changes need a person, and the trick is to route only those changes to a person, because a review process that sees everything sees nothing.

Automate the routing by risk. A prompt change to the formatting block goes through with a code review. A prompt change to the safety block requires the safety owner. A model migration requires the launch review. A change to a policy answer requires the policy owner. Encode this in a code-owners file so the right reviewer is requested by the tooling rather than by somebody's memory.

Give the reviewer what they need in one screen: the diff, the crosstab, the regressed cases, and the gate result. A reviewer who has to run something to review something will approve it without running it.

And measure the review. Approval rate, time to decision, and, if you have the stomach for it, planted errors, the same trick from Chapter Eleven. A reviewer who approves everything in under a minute is not a control. Knowing that is uncomfortable and it is better than the alternative, which is believing you have a control that you do not have.

Automated rollback triggers

The triggers are written before the deployment, they are monitored automatically, and they fire without a meeting.

Safety violation rate above baseline. Latency at the ninety-ninth percentile above the budget. Error rate above threshold. Tool failure rate above threshold. Refusal rate up beyond a bound. Each of these is a number you already collect, each has a threshold you can set from the current baseline, and each can trigger a routing change in seconds.

Two design points matter and both are learned the hard way. Use a time window rather than an instantaneous reading, because a one-second spike is noise and a five-minute elevation is a signal, and a trigger that fires on noise will be disabled by an annoyed on-call engineer within a week. And make rollback the safe default: if the monitoring itself fails, if the metrics pipeline is broken, if the guardrail cannot be evaluated, roll back. A system that continues rolling out when it cannot see is a system that will roll out an outage.

Test the rollback, on a schedule, in production. Deploy something known-bad to a tiny canary and confirm that the machinery fires. A rollback path that has never been exercised is a rollback path that does not work, and you will find that out during the incident it was built for.

A small piece of pipeline design that removes a recurring argument, and that is worth building on the first day.

Make the evaluation reproducible from the report. Every comparison report should carry the exact command, or the exact version tuple, that would reproduce it. When somebody disputes a number, and they will, the conversation should end with them running it themselves rather than with two engineers reconstructing what was probably run.

This sounds like a convenience feature. It is a trust feature. An evaluation number that only one person can reproduce is a number that only one person believes, and the whole purpose of the apparatus is to produce facts that an organization can act on collectively.

Dashboarding evaluation results

The dashboard is where all this either becomes an organizational habit or becomes a thing an engineer maintains alone.

Two or three headline numbers, no more. Overall pass rate. Critical failure count. Groundedness rate. Those go at the top, large, with the trend. Everything else is a drill-down, and the drill-down should be one click, and every number should link to the cases behind it, because a metric you cannot click into is a metric nobody trusts.

Show the dataset version and the noise floor next to every number, always. A quality score without its dataset version is uninterpretable and a delta without a noise floor is an invitation to over-react. Putting both on the chart is a small design decision that will prevent a hundred pointless conversations.

Segment by severity and by stratum, because the aggregate is where regressions hide. And show offline and online metrics on the same page, next to each other, because the relationship between them is the thing that tells you whether your offline suite is still measuring anything real. When the offline score rises and the escalation rate does not fall, something is wrong with your instrument, and only a dashboard that shows both will let anybody notice.

A word about what to do when the gate blocks something and everybody believes the gate is wrong.

Sometimes it is. A gate can fail because a test case is stale, because the judge miscalibrated, because a policy changed and the dataset did not. When that happens, the correct response is to fix the instrument, in a pull request, with a review, and then re-run. What is not correct, and what will happen if you allow it, is to override the gate on the grounds that the instrument is probably wrong and to leave the instrument in place.

The rule I would enforce is that an override requires a follow-up ticket against the instrument, filed before the deployment, with an owner. If the gate was wrong, somebody now has to fix it. If nobody is willing to file the ticket, then nobody actually believes the gate was wrong, and the override was a schedule decision wearing a methodological argument.

Handling noisy metrics

Every number in this chapter is noisy, and a team that has not characterized the noise will alternate between panic and complacency, usually in the wrong order.

Publish the noise floor. Run the suite twice against an unchanged system, take the difference, and put that number on the dashboard. Any move smaller than it is not a move. This single act will save your team more time than any other recommendation in this chapter, because it ends the recurring meeting about a one-point fluctuation.

Reduce the noise where you can. Fix the seed. Run each case several times and average. Use paired comparisons rather than independent ones, which is the largest single variance reduction available and costs nothing but discipline. Control the judge's position bias with order swapping. Each of these tightens the instrument, and a tighter instrument catches smaller regressions.

Then interpret with some humility about your own incentives. Twyman's law, the observation Ron Kohavi has done more than anyone to popularize, says that any figure that looks interesting or different is usually wrong. It applies with special force to a metric that just moved in the direction you were hoping for, on the day of your review. Investigate the wins as hard as you investigate the losses, and you will find that a surprising number of them were an instrument change nobody mentioned.

A short observation about what a gate does to a team, because the second-order effect is larger than the first.

The obvious effect of a gate is that bad changes stop shipping. The larger effect is that engineers start thinking about quality before they write the change rather than after, because they know a number is going to be computed and posted on their pull request. That anticipation does more for the product than the blocking does. It is the same mechanism by which a codebase with good test coverage produces more careful code, and it arrives for free once the gate is real.

Which is also why a gate that is routinely overridden is worse than no gate at all. It teaches the team that the number is theater, and having learned that, they will stop thinking about it before they write the change, and you will have paid for the machinery and lost the benefit.

Governance and ownership

The last question is who owns this, and the answer determines whether any of it survives the next reorganization.

Evaluation needs a named owner, with a title, whose performance is measured on the health of the evaluation system rather than on the quality score it produces. That distinction matters enormously. An owner rewarded for the number going up will make the number go up, and the easiest way to do that is to soften the dataset. An owner rewarded for the number being trustworthy will defend it against exactly that.

The dataset needs an owner. The rubrics need an owner. The judge calibration needs an owner, and the calibration needs to be re-run on a schedule or it will silently expire. The gate thresholds need an owner, and the owner needs enough standing to say no to a launch, which means the role cannot report into the team whose launches it is gating.

And the overrides need a review. Every gate override, once a month, in front of a group, with a short accounting of what happened afterward. Most of them will turn out fine, and the two that did not will teach the organization more about its own risk tolerance than any policy document.

Gate condition	Threshold	On failure
Deterministic checks	All pass	Block; no override
Prompt render assertions	No empty placeholders; token budget respected	Block; no override
New critical failures	Zero	Block; VP override only, recorded
Major failure count	Not above the production baseline	Block; named override
Aggregate quality delta	Drop no larger than the published noise floor	Block; named override

Gate condition	Threshold	On failure
Per-criterion delta	No single criterion down by more than its limit	Block; named override
Safety suite	Full adversarial suite run; zero passes through	Block; no override
Over-refusal rate	Not above baseline	Warn; review required
Latency p99	Within the published budget	Block; named override
Cost per request	Within budget	Warn; finance review
Judge calibration age	Re-calibrated within the last quarter	Warn; results flagged as uncertain
Dataset version	Stated on the report; frozen track used for the trend	Block if unstated
Regression review	Every pass-to-fail case read by a human	Block
Canary guardrails	Rollback triggers armed and tested	Block
Rollback path	Configuration change, not a code release	Block
Override record	Name, reason, and accepted risk written down	Block

Read the On failure column and notice how many rows say named override rather than no override. That is deliberate. A gate that cannot be overridden will be routed around, usually by someone building a second deployment path that does not have a gate on it. A gate that can be overridden by a person who has to sign their name is a gate that holds, because the cost of the override is social rather than technical, and social costs are the only ones that scale.

All of this stops a bad change from reaching users. None of it tells you what to do when quality drops and nothing changed, which happens, and which is the situation where most teams discover that they have been flying without instruments.

Part Five

Observability and Production Feedback

Chapter 17: Observability for LLM Quality

A customer sent us a screenshot of an answer that was wrong in a specific and alarming way, with a timestamp, and asked what had happened.

We had the request. We had the response. We had a trace identifier, a latency measurement, and a two-hundred status code. We had the model name and the prompt version. Everything a well-instrumented service is supposed to have, and we had it.

We could not answer the question.

We did not know what documents had been retrieved, because we logged the retrieval call's duration and not its result. We did not know what the conversation summary had said, because the summary was generated in memory and never persisted. We did not know what the model had actually been sent, because the context was assembled at request time from six sources and thrown away afterward. We had a request and a response and a black box between them, and the wrongness happened inside the box.

This chapter is about the box. It is, in my view, the single highest-return investment described in this book, because everything else here depends on it. You cannot build a dataset from production traffic you did not record. You cannot compare two models on identical contexts you did not save. You cannot classify a failure whose cause you cannot see. Observability is the substrate, and most teams build it last, after an incident forces them to.

Traditional observability vs. LLM observability

Conventional service observability answers a well-defined question: is the system healthy. The signals that answer it are the ones the industry has converged on over two decades. Latency, traffic, errors, saturation, the four that the Google site reliability engineering book names as the golden signals. They tell you whether requests are being served, how fast, and how often they fail.

Every one of those signals can be perfectly green while your product is producing garbage.

That is the whole difficulty in one sentence. A generative system fails by returning a well-formed, fast, successful response containing a false statement. There is no error to count. The status code is two hundred. The latency is fine. The system is healthy and the product is broken, and your entire monitoring stack is structurally incapable of telling the difference.

So you need a second layer, measuring quality rather than health, and it has properties the first layer does not. The signals are semantic rather than numeric, which means computing them costs a model call. They are slow, arriving hours or days after the request rather than in real time. They are sampled, because you cannot afford to score everything. And they are probabilistic, because the scorer itself has error bars.

Run both layers. Keep them distinct. A team that folds quality metrics into its service dashboard will end up with an on-call engineer paged at three in the morning for a groundedness drop they can do nothing about, and after the third time, the alert will be muted.

Prompt and context logging

If you take one recommendation from this chapter, take this one. Log the assembled context.

The input to the model has never been the user's query alone. It is the system prompt, plus the conversation history in whatever form survived the context builder, plus the retrieved passages,

plus tool results, plus injected state, plus whatever else your pipeline decided to include, ordered and truncated by a policy nobody wrote down. That artifact is the actual input, and if you did not save it, you cannot reproduce, compare, debug, or evaluate anything.

Log it whole, exactly as sent. Not a summary. Not the identifiers of the documents that were retrieved, though log those too. The rendered text, byte for byte, because a difference in whitespace or ordering can change behavior and you want to be able to see it.

The objection is storage, and it is a real objection. A full context can run to tens of thousands of tokens and a high-traffic product produces a great many of them. Three answers. Sample: log the full context for a small percentage of traffic and for one hundred percent of anything with a negative signal or a high-risk intent. Compress, because this is highly redundant text and it compresses well. And set a short retention window, measured in weeks, because you need it for debugging and evaluation rather than for posterity.

Also log the version of everything. Prompt version, model identifier, retrieval configuration, index snapshot, feature flags, and the identifier of any experiment the request was enrolled in. When quality moves, the first question is what changed, and a request that carries the full version tuple lets you answer it with a query instead of an investigation.

Token usage tracking

Tokens are your unit of cost and one of your best leading indicators of trouble, and most teams track them only at the level of the monthly bill.

Break them out per request and by role. Prompt tokens, split into system prompt, conversation history, and retrieved evidence, because those three grow for different reasons and the split tells you which one is running away. Completion tokens. And reasoning tokens, if your model produces them, which you pay for and never see, and which can dwarf everything else.

Watch the distribution rather than the mean. Mean prompt tokens is a stable number that hides the tail, and the tail is where your cost and your context-window failures live together. The requests at the ninety-ninth percentile of prompt length are the ones getting truncated, which means they are the ones losing evidence, which means they are the ones producing bad answers. Your most expensive requests and your worst answers are frequently the same requests.

That correlation is worth checking directly and it surprises people every time. Plot quality against prompt length. If it slopes downward past a certain length, you have found your truncation cliff, and you have found it from a cost dashboard rather than a quality one.

Track token growth over time as a first-class metric. Prompts only ever get longer, retrieved evidence only ever gets more generous, and both happen a few hundred tokens at a time across many pull requests, and nobody notices until the bill arrives or the context overflows.

Latency metrics

For a streaming product, one number matters more than the rest, and it is the one most teams do not instrument.

Time to first token is what the user experiences as speed. Everything before it is a spinner. Total generation time affects how long they wait for the full answer, and by then they are already reading, which is a completely different psychological state from waiting. A system with a fast first token and a slow stream feels responsive. A system with a slow first token and a fast stream feels broken, even if the total time is identical.

Decompose the latency budget by stage, because a single end-to-end number tells you that something is slow and never which thing. Intent classification, retrieval, reranking, context assembly, the model's time to first token, tool calls, post-processing. Each stage gets a budget, each budget is monitored, and a regression in any one of them is attributable immediately.

Report percentiles, and report the ninety-ninth. The median is a number for a slide. The ninety-ninth is where your timeouts fire, where your angriest users live, and where the correlation with abandonment is strongest. And a mean, in a distribution this skewed, is close to meaningless.

Track streaming stalls separately. A response that starts in three hundred milliseconds and then pauses for four seconds mid-sentence produces a worse experience than one that takes two seconds and streams evenly, and no end-to-end metric will ever show you that.

Retrieval traces

Chapter Fifteen argued that retrieval is where most of your quality is decided. This is what you have to log to be able to see it.

The query as issued to the retriever, which is often not the user's query, because something rewrote or expanded or embedded it, and the rewrite is a component that can fail. The candidates returned, with their identifiers and scores. The results after reranking, with the new scores, so that you can tell whether the reranker moved anything. Which of the results made it into the final context, which is a different set, because the context builder dropped some. And the position each one landed in, since position determines how much attention it gets.

Log the index version and the age of each retrieved document. This is what lets you answer the staleness question retrospectively, when a customer complains about an answer that was correct three weeks ago and is wrong now.

And log the negative case. When retrieval returns nothing, or returns only results below your confidence threshold, that is a signal, and it is one of the strongest predictors of a bad answer you will find anywhere in the pipeline. A low-confidence retrieval that is followed by a confident answer is a hallucination waiting to be reported, and you can catch a large fraction of them in real time, from a trace, without a judge model.

Tool-call traces

Every tool call gets a record, and the record exists for something other than debugging. It is for the conversation you will eventually have with somebody about what your system did to a customer's account.

The tool name and every argument, including the ones the model derived. The result, or the error. The latency. The idempotency key. The authorization decision and the basis for it, meaning which identity was checked against which permission. Whether confirmation was requested, and the exact text shown to the user when it was, and what they said back.

Chain these into a trace for the whole turn, so that a multi-step agentic sequence reads as one coherent story rather than eleven unrelated log lines. Distributed tracing tooling already solves this problem and you should use it rather than inventing a worse version, threading a trace identifier through every hop.

The test of whether your tool logging is good enough is the one from Chapter Eleven. Hand the trace of a completed session to an engineer who was not involved and ask them to reconstruct what happened and why. If they cannot do it from the trace alone, the trace is insufficient, and you will discover exactly how insufficient during the incident it was built for.

Safety events

Safety needs its own telemetry, at a higher fidelity than everything else, with a longer retention and a tighter access control.

Log every trigger of a safety classifier, on input and on output, with the category and the score, and log the ones that fired below threshold too. The near misses are the leading indicator. A category whose scores are creeping upward over weeks, without ever crossing the line, is telling you that something in your traffic is shifting, and you have a month of warning if you are watching.

Log refusals separately from safety interventions, because they are different events with different failure modes. A refusal is the model declining. A safety intervention is a layer of your system stopping something. Conflating them means you cannot tell whether your over-refusal problem is a model problem or a filter problem.

And monitor the over-refusal rate as attentively as the unsafe-output rate. A system that refuses too much is failing users, quietly, in a way that no safety dashboard will ever show as red, because from a safety dashboard's perspective every refusal looks like a success.

A property of logs that is easy to forget and that will eventually cost somebody a week.

Logs lie about time. A trace assembled from several services carries several clocks, and the ordering you see in a viewer is the ordering the clocks reported rather than the ordering things happened. In a pipeline where a retrieval, a model call, and three tool calls are interleaved, that difference matters, because the question in the incident is frequently what happened before what.

Use a single trace identifier, propagate it through every hop, and record a monotonic sequence number alongside the timestamp. It costs an integer. It converts a trace from a collection of events into a story, and stories are what you need at two in the morning.

User feedback

Capture the explicit signal, and understand what it is and what it is not.

Thumbs up and thumbs down have a low response rate, and the population that clicks is not the population that uses. Angry users click. Delighted users occasionally click. Satisfied users, who are the majority, do nothing. Treated as a quality metric, this is a biased estimator. Treated as a sampling mechanism for finding failures, it is excellent, and that is how to use it.

Ask why, with a short list of reasons and an optional free-text box. The reason turns an uninterpretable signal into a routable one, because a thumbs down tagged as wrong information goes to a different team from one tagged as too slow, and without the tag they all go to the same triage queue where they die.

The implicit signals carry more information and require more work to interpret. A user rephrasing the same question is a strong negative. A user copying part of the response is a positive. A user abandoning the session is ambiguous and depends heavily on where in the flow they left. A user escalating to a human is the clearest negative signal available and the one most worth instrumenting properly.

Log the conversation alongside the feedback, always. Feedback without its context is a number with no attached meaning, and it will sit in a dashboard being unactionable for as long as anyone bothers to look at it.

One signal in this catalog is worth building before any of the others, and it is not on most teams' radar at all.

The retrieval confidence signal. When your retriever returns nothing above threshold, or returns results whose scores are unusually low, you know, at request time, before generation, that the answer is likely to be bad. That is a real-time prediction of a quality failure, available for free, computed from data you already have.

What you do with it is a product decision. You can hedge the answer. You can ask a clarifying question. You can route to a human. You can simply log it and count. But a system that has this signal and discards it is throwing away the only reliable advance warning it will ever get, and a dashboard panel showing the rate of low-confidence retrievals that were followed by a confident answer is a list of hallucinations waiting to be reported.

Quality dashboards

The dashboard is where all of this either becomes a shared organizational fact or stays an engineer's private spreadsheet.

Three headline numbers at the top, no more. Overall quality on the frozen dataset. Critical failure count. Groundedness rate. Under them, the segments: by intent, by conversation depth, by severity, by user cohort. The aggregate is where regressions hide, and the segment view is where they surface.

Put offline and online metrics on the same page. The relationship between them is the most important thing on the dashboard, and it is invisible if they live in different tools. When the offline score rises and the escalation rate stays flat, your offline suite has stopped measuring anything that matters, and only a page that shows both will let anyone notice.

Make every number clickable through to the cases behind it. A metric you cannot drill into is a metric nobody believes, and disbelief is fatal, because the moment a team stops trusting a number they stop acting on it and the whole apparatus becomes theater.

And show the dataset version and the noise floor next to every quality figure. Always. A score without those two pieces of context is a number that will be misread by somebody senior in a meeting you are not in.

One trap in this chapter is worth stating explicitly, because it catches teams that have done everything else right.

Logging is not free at the tail. A full context capture on a request that is already at the ninety-ninth percentile of prompt length is a large write on the request that is already the slowest, and if the write is synchronous you have just made your worst latency worse. Make it asynchronous, sample it, and put a circuit breaker on it, so that a logging pipeline under pressure degrades the telemetry rather than the product.

And test what happens when the logging fails. A system that throws an exception because it could not write a trace has converted an observability problem into an outage, which is a way of being punished for having tried to measure things.

Privacy and retention concerns

Everything in this chapter increases your exposure, and the exposure has to be managed deliberately rather than discovered.

Conversation logs contain whatever users typed, which is everything. Names, account numbers, health information, the reason they are upset. Redact at ingestion, before the data lands anywhere queryable, using the pipeline from Chapter Six, and test the redaction, because untested redaction is a promise you have made to your legal team without evidence.

Set retention by purpose and keep it short. Raw contexts for a few weeks, which is long enough to debug an incident. Derived evaluation cases, redacted and abstracted, for longer, because they no longer contain a person. Aggregate metrics indefinitely, because they contain nobody. This tiering is the structure that makes the whole program defensible, and it costs nothing to design in and a great deal to retrofit.

Control access. Full conversation logs are a much more sensitive dataset than anything else your engineering team touches, and treating them like ordinary application logs is a decision you will regret in a specific and public way. Restrict, audit, and log the access itself.

And build for deletion. When a user asks to be forgotten, their conversations have to come out of the logs, the evaluation dataset, and any derived artifact. Designing this after the fact is expensive; designing it in means tagging every record with a user identifier from the beginning, which costs one column.

Signal	What it tells you	Cadence
Assembled context, verbatim	The actual model input; enables replay and comparison	Sampled + all negatives
Version tuple	Prompt, model, index, config, experiment arm	Every request
Prompt tokens by role	System vs. history vs. evidence growth	Every request
Completion tokens	Output cost and verbosity drift	Every request
Reasoning tokens	Hidden cost you are paying for	Every request
Time to first token	What the user experiences as speed	Every request, p50 and p99
Per-stage latency	Which component regressed	Every request
Streaming stalls	Mid-response pauses invisible to end-to-end timing	Every request
Retriever query	Whether the rewrite step is failing	Every retrieval
Candidates and scores	Recall and ranking behavior in the wild	Every retrieval
Passages surviving assembly	Whether truncation is discarding your evidence	Every request
Position of top passage	Whether the best evidence landed in the worst place	Every request
Retrieved document age	Staleness, retrospectively	Every retrieval
Low-confidence retrieval	Strongest real-time predictor of a bad answer	Every retrieval
Tool name and arguments	What the system did on the user's behalf	Every call
Authorization decision and basis	Whether the check actually happened	Every call
Idempotency key	Whether a retry duplicated an effect	Every call
Confirmation text shown	What the user actually agreed to	Every irreversible action
Safety triggers, including near misses	Leading indicator of a traffic shift	Every request

Signal	What it tells you	Cadence
Refusals, separate from interventions	Model declining vs. filter blocking	Every request
Over-refusal rate	Users being failed invisibly	Daily
Explicit feedback with reason code	A routable negative signal	As given
User rephrase rate	Strong implicit negative	Daily
Escalation rate	The clearest failure signal you have	Daily
Offline quality by stratum	Regression detection	Per release, nightly
Judge agreement statistic	Whether the instrument is still calibrated	Quarterly
Noise floor	The smallest delta that means anything	Quarterly
Dataset version	What the score is a score of	Attached to every score

Twenty-eight rows, and the first one is worth the other twenty-seven. A team that logs the assembled context and nothing else can build everything in this book afterward. A team that logs everything else and not the context is permanently blind, and the blindness will not present as blindness. It will present as a customer with a screenshot and a question nobody can answer.

You can now see what your system did. Which raises the next question: when it did something wrong, what exactly do you call that, and who fixes it?

Chapter 18: Failure Taxonomy for LLM Applications

The bug tracker had four hundred and eleven open tickets and one label. The label was ai-quality.

Under it sat everything. A response that was too long. A response that invented a policy. A response that took nine seconds. A response that forgot what the customer had said two turns earlier. A response that called the wrong tool. A response that was, as far as anyone could tell, perfectly good, and had been filed by someone who did not like its tone.

Nobody owned the label, because it did not describe a component. Nobody could prioritize within it, because the tickets were not comparable. And nobody could tell whether the situation was improving, because the count went up when people filed things and down when somebody got frustrated and closed a batch, and neither movement had anything to do with the product.

The four hundred and eleven tickets contained real information about real failures, and the information was inaccessible, because we had no vocabulary. This chapter is about building one.

Why failure classification matters

A taxonomy does four things, and each of them is a thing your team currently does badly.

It assigns ownership. A failure classified as a retrieval failure belongs to the team that owns retrieval, and they can act on it. A failure classified as ai-quality belongs to nobody, and it will sit in the backlog until the quarter ends and somebody closes it as stale.

It makes failures countable. Once every failure has a class, you can count the classes, rank them, and discover that forty percent of your complaints are one thing. That single fact is worth more than a hundred individually triaged tickets, and you cannot compute it until the classes exist.

It creates a shared language. Two engineers looking at the same incident will describe it in the same words, which sounds trivial and is not. Most of the wasted time in an incident review is spent establishing what happened, and most of that time is spent because the participants are using different words for the same thing and the same words for different things.

And it connects failures to fixes. A class has a component, a component has an owner, an owner has a backlog, and a failure that lands in that backlog gets fixed. That chain is the whole point, and it breaks at the first link if the class does not exist.

There is a fifth thing, which is harder to describe and which I have come to think matters most. A taxonomy changes what people believe about the system. Before you have one, every engineer in the room carries a private theory about why the product is bad, and the theories are incompatible, and none of them are testable. Somebody thinks it is the model. Somebody thinks it is the prompt. Somebody thinks the users are asking bad questions. These beliefs are held sincerely, they drive roadmap arguments, and they are decided by whoever is most senior rather than by evidence.

A month after you start classifying failures, the argument is over. Not because anybody won it, but because there is a table with counts in it, and the counts do not care what anyone thought. The single most common reaction I have seen to a first honest classification is a quiet, slightly embarrassed silence, followed by somebody saying that they had no idea retrieval was that much of it.

Failure ownership by system component

The classification axis that works is the pipeline stage, because the pipeline stage determines the owner, and the owner is what turns a category into a fix.

This is why the taxonomy has to be built on the map from Chapter Three rather than on the symptom the user experienced. A user reports that the assistant made something up. That is a symptom, and it maps to at least four different classes with four different owners, and the whole job of triage is to make that determination.

#	Failure class	Pipeline stage	Owner	Default severity
1	Intent misunderstanding	Input	Routing	Major
2	Retrieval failure	Retrieval	Search	Major
3	Missing context	Context	Context builder	Major
4	Stale context	Retrieval	Indexing and content	Major
5	Prompt instruction failure	Prompt	Prompt owner	Major
6	Hallucination	Model	Model and prompt	Major
7	Unsafe response	Model	Safety	Critical
8	Tool selection error	Tools	Agent layer	Major
9	Tool argument error	Tools	Agent layer	Critical
10	Latency timeout	Delivery	Infrastructure	Major
11	Streaming interruption	Delivery	Infrastructure	Major
12	Formatting failure	Delivery	Post-processing	Minor
13	Memory inconsistency	Context	State layer	Major
14	User experience failure	Experience	Product	Minor

Fourteen classes, arranged by where in the pipeline the damage was done rather than by where it became visible. Notice how few of them belong to the model. Two, out of fourteen, and one of those two is usually a prompt problem in disguise. If your organization's mental model of AI quality is that the model is either good or bad, that table is the argument against it, and it is worth putting on a wall.

Notice also that three classes belong to retrieval and context, which is where I would expect the largest share of your volume to land if you classify honestly. Teams that have never built a taxonomy consistently believe their problem is the model. Teams that have built one consistently discover their problem is the plumbing.

The fourteen classes

Intent misunderstanding. The system decided the user was asking one thing and they were asking another. Everything downstream is correct and irrelevant. Owned by whoever owns routing, and detectable by comparing the classified intent against a human reading of the conversation.

Retrieval failure. The evidence needed to answer existed in the corpus and was not returned. Owned by search. This is the class that the diagnostic split from Chapter Fifteen exists to identify, and it is usually the largest.

Missing context. The evidence was retrieved and did not survive assembly, or the conversation history that mattered was truncated away. Owned by the context builder, which is the component nobody thinks of as a component.

Stale context. The evidence was returned, and it was out of date. Owned jointly by the indexing pipeline and the content team, and the split between them is the difference between index lag and content lag.

Prompt instruction failure. The prompt told the model to do something and the model did not do it, or the prompt told it two contradictory things. Owned by whoever owns the prompt, and detectable with the per-instruction test suite from Chapter Thirteen.

Hallucination. The model asserted something with no basis in the evidence and no truth behind it. Owned by model and prompt, and much rarer than it is reported, because most reported hallucinations are one of the four classes above wearing a costume.

Unsafe response. Harmful content, a policy violation, or a promise the company cannot keep. Owned by safety, and it is the class where a single instance is an incident regardless of the rate.

Tool selection error. The wrong tool, or no tool when one was required, or a tool when none was. Owned by the agent layer.

Tool argument error. The right tool with the wrong arguments. Same owner and a much higher severity, because a wrong argument is an action taken in the world.

Latency timeout. The answer was good and arrived too late to be an answer. Owned by infrastructure, and it correlates with abandonment more strongly than most teams expect.

Streaming interruption. The response was cut off mid-delivery. Owned by infrastructure, and it poisons your metrics, because a truncated answer looks like a bad answer to every scorer you have.

Formatting failure. The content was right and the rendering was wrong. Owned by post-processing, and this includes the enrichment failure where a product card contradicts the paragraph next to it.

Memory inconsistency. The system contradicted itself across turns, or forgot something it had been told, or remembered something the user had corrected. Owned by the state layer, and it is the class that produces the complaints users describe as the assistant not listening.

User experience failure. Everything was correct and the person still did not get what they needed. Owned by product, and it is the class that teams delete from the taxonomy because it feels like an admission. Leave it in. It is where the interesting problems go to be ignored.

Severity levels

Class and severity are independent axes and confusing them will distort your priorities.

A hallucination about a product color and a hallucination about a safety requirement are the same class and different emergencies. A latency timeout on a browsing query and a latency timeout on a payment confirmation are the same class and different emergencies. The class tells you who fixes it. The severity tells you when.

Four levels, from Chapter Eight, and the definitions have to be written down because otherwise every engineer will rate their own bug as major. Critical: harm, legal exposure, financial loss, or a commitment the company cannot honor. Major: the user receives materially wrong information and will act on it. Minor: the answer is worse than it should be and no one is damaged. Cosmetic: somebody on the team noticed.

Report the matrix rather than the totals. A grid with classes down one side and severities across the top is the single most useful artifact your evaluation program will produce, because it shows where your exposure is rather than where your volume is, and those are different cells, and the second one is the one everybody works on by default.

Assign the severity from the case, not from the outcome, which is a discipline that sounds pedantic and prevents a specific and common distortion. If severity is judged after the fact, from what actually happened, then a critical failure that happened to be caught by a customer who shrugged gets logged as minor, and the identical failure that reached somebody who complained gets logged as critical. You are now measuring your customers' tolerance rather than your own risk, and your priorities will be set by whoever is loudest.

Severity belongs to the question, decided in advance, by somebody who knows what is at stake if the answer is wrong. A wrong answer about a medication interaction is critical whether or not anybody was harmed this time.

A trap in triage that produces a taxonomy full of the wrong numbers, and it is the most common mistake teams make in their first month of classifying.

Classify by cause, not by symptom, and the distinction requires the trace. A user reports that the assistant made something up, and the natural thing to do is to file it as a hallucination, because that is what it looked like. Do that consistently and your taxonomy will report that hallucination is your dominant failure class, which will be false, and your roadmap will be built on it.

The rule is that nothing gets classified without a trace. If the trace is unavailable, the case is filed as unclassifiable and the count of unclassifiable cases goes on the dashboard, because that number is a measurement of your observability gap and it will get the logging fixed faster than any argument you could make.

Root-cause analysis

Classification is a hypothesis. Root-cause analysis is the confirmation, and skipping it produces a taxonomy that is precisely organized and wrong.

The procedure runs backward through the pipeline and it is mechanical, which is why it can be a checklist rather than an art. Start at the response. Was the assembled context sufficient to answer correctly? If no, the failure is upstream of the model and you stop blaming it. Was the evidence retrieved? If no, was it in the corpus at all? If it was in the corpus and not retrieved, retrieval failed. If it was retrieved and did not reach the model, the context builder failed. If it reached the model and the model ignored it, now you have a model or prompt problem, and now you have earned the right to say so.

Every one of those questions is answerable from the trace, if you built the trace in Chapter Seventeen, and unanswerable if you did not. That is the dependency between the two chapters and it is why observability comes first.

Do this blamelessly, which is a discipline the site reliability community formalized long before any of this and which applies here with more force, because generative systems fail in ways that

are genuinely nobody's fault. The engineer who added the word concise was doing their job well. The content team that changed the template had no idea we existed. A review that looks for the person who erred will find one, and it will stop looking before it finds the system that made the error inevitable.

Linking failures to evaluation cases

A taxonomy that does not connect to your evaluation suite is a filing system.

Tag every evaluation case with the failure class it was designed to catch. Then, when a class shows up in production, you can ask the question that matters: did the suite have coverage for this, and did the coverage pass? Three answers, and each one means something different.

No coverage. The suite never tested for this. That is a gap, and the fix is a new family of cases, and it is the most common answer the first year you do this.

Coverage existed and failed, and you shipped anyway. That is a gate problem, and it is the most embarrassing answer, and it will happen, and the review should be about the override rather than about the failure.

Coverage existed and passed. That is the interesting one, and it means your test case does not actually test what you thought it tested. The case is too easy, or it tests the phrasing rather than the behavior, or the production condition differs from the test condition in a way nobody anticipated. This is the answer that improves your suite the most, and it is the one that requires the most honesty to reach.

Report coverage by class. A failure class with high production volume and no evaluation coverage is a hole with a number attached, and putting that number on a dashboard is how the hole gets closed.

One rule about the taxonomy itself, and it is the rule that keeps it useful past its first year.

Do not let it grow. The pressure to add classes is constant, because every incident feels slightly unlike the last one, and adding a class is easier than arguing about which existing class it belongs to. Within eighteen months you will have forty categories, most of them used twice, and the counts that made the taxonomy valuable will have dissolved into a long tail that tells you nothing.

Fourteen is enough. If a failure does not fit, the right question is usually which of the existing classes it is closest to, and the answer is usually obvious once somebody insists on one. Add a class only when a genuinely new component enters the pipeline, which happens perhaps once a year, and prune the classes that have not been used in two.

Turning incidents into regression tests

This is the ratchet. Every incident, every one, ends its life as a permanent test case, and the case never gets deleted.

The workflow is short and it belongs in your incident template rather than in someone's memory. Reproduce the failure from the trace. Write a test case that fails against the current system. Fix the system. Confirm the case passes. Add it to the frozen suite with the incident identifier attached. Tag it with the failure class. And then, because a single case is a single case, generate the family: twenty variations of the same failure, so that the fix is confirmed to generalize rather than confirmed to satisfy one string.

That last step is what separates a real regression suite from a collection of anecdotes, and it is the step that gets skipped. A fix that passes the one case from the incident and fails eighteen of its twenty siblings is a fix that will produce the same incident with different wording in three months, and you will have a test case proving it was fixed.

Never delete an incident case. They accumulate, and the accumulation is the point. A suite of two hundred cases, each of which represents a failure that once reached a customer, is the most valuable artifact your evaluation program will ever own, and it is the only defense against the specific organizational failure of learning the same lesson twice.

You now have a vocabulary, a map, and a mechanism for turning pain into coverage. What none of that gives you is a reliable read on whether an answer was any good, at scale, on the cases where a machine cannot tell. For that you need people, and people are the hardest system in this book to operate.

Chapter 19: Human Feedback Loops

The reviewer had scored eight hundred and forty responses in one week. Her agreement with the other reviewers, measured on the overlapping subset, was excellent for the first two hundred and had degraded steadily after that. By the last hundred she was passing nearly everything.

She was not being lazy. She was tired, and she had been reading customer-support responses for thirty hours, and somewhere around response five hundred her internal sense of what counted as acceptable had recalibrated against the population she was looking at rather than against the standard she had been given. Everything looked fine because everything looked like everything else.

We had built a data pipeline whose final stage was a human being, and we had run it at a throughput the human could not sustain, and the output degraded in a way that was invisible in the output. The scores kept coming. They were just no longer measuring anything.

Human review is the ground truth on which everything else in this book is calibrated. It is also a system, with a throughput, a failure mode, and an operating cost, and treating it as an infinitely available oracle is the fastest way to corrupt every number you produce.

Human review vs. user feedback

These are two different things and teams merge them constantly, which produces a dataset that is neither.

User feedback comes from the person who was in the conversation. They know what they wanted, which is information nobody else has. They do not know whether the answer was correct, because if they knew the answer they would not have asked. They are unrepresentative, because only some people click. And they are influenced by everything: the latency, the interface, their mood, whether the product did the thing they wanted rather than the thing they asked for.

Human review comes from a trained person who was not in the conversation. They can check correctness against a source. They apply a consistent rubric. They are representative because you chose the sample. And they cannot tell you what the user wanted, because they are inferring intent from a text box.

So they answer different questions and both are necessary. User feedback tells you whether the product worked. Human review tells you why. Using either as a substitute for the other produces a specific and predictable error: teams that rely only on user feedback optimize for satisfaction and ship confidently wrong answers that users thank them for. Teams that rely only on expert review optimize for correctness and ship answers that are right and useless.

Expert review vs. crowd review

The choice between them is a choice about what you are measuring, and getting it wrong is expensive in both directions.

Expert review is required whenever the correct verdict depends on domain knowledge. Whether a policy answer is right. Whether a medical caveat is adequate. Whether a piece of technical guidance would work. A crowd worker cannot score these, and asking them to will produce confident labels that are wrong at a rate you cannot estimate, which is worse than no labels at all, because you will build on them.

Crowd review works for judgments any careful reader can make. Is this response on topic. Is it comprehensible. Does it contradict itself. Is it rude. Is it obviously too long. These are real quality dimensions, they matter, and paying an expert to assess them is a waste of an expert.

The research literature on this is older than it looks. Rion Snow and colleagues published work in 2008 comparing non-expert annotation against expert labels across several natural language tasks, and found that aggregating a handful of non-expert judgments could approach expert quality on tasks where the judgment did not require training. The operative clause is the last one. The technique is sound and the boundary condition is the whole story.

So route by criterion. The same response can be scored on tone by a crowd and on policy correctness by a specialist, and the two verdicts go into different columns. What you must never do is average them, or let a crowd worker's judgment about an expert criterion enter your dataset because it was cheaper.

Reviewer training

A reviewer who has read the rubric is not trained. A reviewer who has scored fifty examples, compared their scores against a reference, and argued about the ones they got wrong, is trained.

The training set is the artifact. Fifty cases with agreed labels, including at least fifteen boundary cases where reasonable people initially disagreed, along with a written record of why the agreed answer is the agreed answer. That record is more valuable than the rubric, because the rubric states the rule and the record shows the rule being applied to a hard case, which is what a reviewer actually needs.

Certify before anyone's scores enter your dataset. A new reviewer scores the training set, their agreement with the reference is computed, and if it clears the bar they are in and if it does not they go through the cases they missed with someone who knows. This is a half-day process and it is the difference between a reviewer pool and a source of noise.

Re-certify on a schedule. Reviewers drift, as the story at the top of this chapter shows, and the drift is silent. A quarterly re-certification against a frozen anchor set costs an afternoon and catches it.

Calibration exercises

Calibration is a group activity and it should be a recurring meeting, and it is the meeting that improves your rubric more than any other work you will do.

The format: ten cases, chosen because they are hard, scored independently in advance. Everyone arrives with their scores. Reveal them together. Spend the meeting on the disagreements, and only on the disagreements, and drive each one to a resolution that gets written down.

The resolution is almost never that somebody was careless. It is that the rubric was ambiguous, and the ambiguity had never been exposed because nobody had encountered a case that sat on the line. Fix the rubric in the meeting, add the case to the training set, and move on. Over six months of doing this the rubric becomes a document that actually specifies behavior rather than gesturing at it.

Do this monthly at minimum, and do it immediately after any change to the rubric, because a rubric edit invalidates the calibration and everybody will interpret the new sentence slightly differently until they have argued about it once.

Measuring inter-rater agreement

If you do not measure agreement, you do not know whether your ground truth is ground or truth.

Overlap deliberately. Ten to twenty percent of every batch goes to two reviewers, always, forever. That overlap costs you a modest amount of throughput and buys you a continuous read on whether your instrument is stable, and it is the only way to catch the fatigue effect from the top of this chapter while it is happening rather than afterward.

Use a chance-corrected statistic. Cohen's kappa for two raters, Krippendorff's alpha when you have more, and compute it per criterion rather than overall. A pool with excellent agreement on formatting and terrible agreement on groundedness has a decent aggregate kappa and a groundedness metric that means nothing.

Set a floor and act on it. If agreement on a criterion falls below your bar, that criterion is not measurable by your current pool with your current rubric, and reporting a score for it is a fabrication. Either fix the rubric, retrain the pool, or delete the criterion. Continuing to report it because it is on the dashboard is the option most teams choose and it is the one that ends with a leadership decision made on a number that was noise.

And track agreement over time, per reviewer. A reviewer whose agreement is falling is a reviewer who is tired or who has drifted, and both are fixable, and neither is detectable from their output alone.

Disagreement resolution

Disagreements need a defined path or they will be resolved by whoever cares most, which is a selection process with no relationship to correctness.

A third reviewer breaks ties, and the tiebreak is recorded. That is the mechanism, and it handles most cases in a minute.

What it does not handle is the case where the third reviewer also finds it genuinely ambiguous, and those cases are the valuable ones. They go to the case owner from the schema in Chapter Four, they get an adjudicated answer, and the answer goes into the training set as a boundary example. A case that produces a three-way disagreement is a case that has just told you where your rubric ends, and that information is worth more than the label.

Some cases should be discarded rather than adjudicated. If the correct behavior is genuinely a product decision that has never been made, the case is not measurable and it should be removed from the suite and turned into a question for whoever owns the product. Keeping it produces a case that will be scored inconsistently forever, and every run of your suite will re-litigate it, and the noise it contributes will hide real regressions.

A caution about scale that applies the moment somebody proposes outsourcing the review operation.

Volume is available and quality is not. A vendor can supply a hundred reviewers next week, and they will produce scores, and the scores will arrive on schedule. What they will not have is your rubric's boundary cases, your product's context, or any sense of what your users were trying to do, and the agreement statistics will tell you so if you compute them, and the temptation not to compute them will be considerable, because the scores are arriving and the dashboard is full.

If you do outsource, run the same certification, the same gold cases, the same overlap, and the same agreement floor that you would run internally, and be prepared to find that a criterion you were counting on cannot be scored by the pool you have hired. Finding that out is the point of

measuring agreement. Not finding it out means you have bought volume and called it ground truth.

Feedback quality control

Your reviewers are a data source, and every data source needs quality control, and there is no polite way to say that.

Plant gold cases in the review queue. Cases with known answers, indistinguishable from the rest, at a low rate. Measure each reviewer's accuracy on them. This is standard practice in every serious annotation operation and it is the only defense against a reviewer who has stopped reading, which will happen, and which will not be visible in their output.

Watch the timing. A reviewer averaging eight seconds on cases that take ninety to read carefully has stopped reading carefully. Treat that as a throughput signal rather than a moral judgment, and fix it by reducing the queue rather than reprimanding the person.

Watch the distribution. A reviewer whose pass rate is fifteen points above the pool has either found something the others missed or has become lenient, and the second is more likely, and either way you want to know.

And be transparent that this is happening. A review operation where quality control is a secret is one where people feel surveilled. One where the gold rate is published, the purpose is explained, and the results are used to improve the rubric rather than to punish is one where people participate in it.

A note on how to treat disagreement between a reviewer and a judge model, because the default handling gets it backward.

When the judge and the human disagree, the instinct is to treat the human as correct and to file a defect against the judge. Sometimes that is right. Often it is not, and the disagreement is telling you that the criterion is ambiguous, and that the human and the judge are both applying it reasonably to a case that the criterion does not actually decide.

So read a sample of the disagreements rather than counting them. The ones where the judge is simply wrong are diagnostic of the judge. The ones where a second human would have sided with the judge are diagnostic of the rubric, and they are more valuable, because fixing the rubric improves every score you will compute from now on rather than one of them.

Converting feedback into product changes

This is where most human review operations fail, and the failure is organizational rather than technical.

Reviewers produce a stream of scored cases. If that stream ends in a dashboard, the entire operation is a metric factory, and the metric will improve or decline and nothing will be different. The stream has to end in a backlog.

So route by failure class, using the taxonomy from Chapter Eighteen. A retrieval failure goes to the search backlog. A prompt instruction failure goes to the prompt owner. A content gap goes to the content team as a list of documents to write. Each of these is a different team and a different ticket, and the routing should be automatic, because a triage step that requires a human will accumulate a queue and the queue will win.

Close the loop visibly. Reviewers who never learn what happened to the failures they found will stop looking carefully, because there is no evidence that looking carefully matters. A monthly

note that says these are the five things you found and here is what got fixed does more than raise morale. It is the mechanism that keeps the quality of the review from decaying.

And feed the cases back into the suite. Every failure a human finds becomes an evaluation case, tagged with its class, and the suite grows in exactly the direction your product is weakest. That is the flywheel, and it is the entire justification for the expense of the review operation.

A point about who your reviewers should be, because the default answer is usually the wrong one.

The default is to hire people whose job is to review, and to keep them separate from the team that builds. That is efficient and it produces a specific blindness: reviewers who have never used the product, never spoken to a customer, and have no model of what the person on the other side was trying to do. They will score the response and miss the interaction.

The mix that works best is a small permanent pool for volume plus a rotation through it of people who build the product, a day a month, scoring cases. The rotation is inefficient and it is the highest-value day in the calendar, because an engineer who has spent six hours reading what their system actually said to people comes back with a roadmap that is very different from the one they had that morning.

Avoiding reviewer fatigue

This section is the one I most want people to take seriously, because the failure it describes is invisible, and invisible failures in a ground-truth pipeline corrupt everything downstream.

Cap the volume. There is a number of responses a person can score carefully in a day and it is much smaller than the number they can score. In my experience it lands somewhere between sixty and a hundred and twenty for substantive criteria, and beyond that the quality falls off a cliff, and the person will not know it is happening. Set the cap, enforce it, and accept the throughput implications rather than pretending they do not exist.

Vary the work. Scoring four hundred consecutive responses on the same criterion in the same intent produces the fastest degradation. Interleave criteria, interleave intents, and give people some review work that is qualitatively different, like adjudication or rubric development, which is more engaging and which improves the system.

Randomize the order and hide the source. A reviewer who knows which system produced which response will score it accordingly, and they will not be doing it deliberately. Blind the review, always, and randomize position in any comparison, for exactly the same reason you do it for a judge model in Chapter Twelve. The biases are the same biases.

And build for the long term rather than for the sprint. A review operation staffed as a crash program before a launch will produce good data for three weeks and garbage for the fourth, and the garbage will look exactly like the good data, and it will go into your frozen dataset and stay there.

Practice	Specification	Why it holds
Scope by criterion	Experts on knowledge criteria; crowd on readability criteria	Wrong labels are worse than no labels
Training set	50 cases with agreed labels; 15 of them boundary cases with written rationale	The rationale teaches, the rubric only states
Certification	Agreement against the reference must clear the bar before scores count	Keeps noise out of the ground truth

Practice	Specification	Why it holds
Overlap rate	10 to 20 percent of every batch double-scored, permanently	Continuous read on instrument stability
Agreement statistic	Chance-corrected, computed per criterion, floor enforced	An aggregate kappa hides a broken criterion
Calibration meeting	Monthly; 10 hard cases; only the disagreements are discussed	Disagreement means the rubric is ambiguous
Tiebreak	Third reviewer; genuine three-way ties go to the case owner	Prevents resolution by whoever cares most
Discard rule	Cases with no decided correct behavior leave the suite	Undecidable cases add noise forever
Gold cases	Planted at a low, published rate; per-reviewer accuracy tracked	The only defense against a reviewer who stopped reading
Timing check	Flag reviewers well below the time the task requires	A throughput signal, not a moral one
Daily cap	60 to 120 substantive judgments; enforced	Beyond it, quality falls and nobody can see it
Work variety	Interleave criteria and intents; rotate in adjudication	Monotony degrades scoring faster than volume
Blinding	Source hidden; comparison order randomized	Reviewers have the same biases judge models do
Anchor set	30 frozen-score cases re-run quarterly	Catches drift in the standard itself
Routing	Every failure auto-routed to the owning team by class	A dashboard is not a fix
Loop closure	Monthly report back to reviewers on what got fixed	Review quality decays without evidence it matters

Sixteen practices, and every one of them costs throughput. That is the trade. A review operation that scores four hundred cases a day badly is producing a number that is worse than useless, because somebody will believe it. One that scores eighty a day well is producing ground truth, and everything else in this book is calibrated against it.

Human review, judge models, rubrics, datasets, gates. Every technique so far has been an attempt to predict, before launch, what will happen after it. Eventually you have to launch, and find out.

Chapter 20: Online Evaluation and Experimentation

The offline numbers were unambiguous. The new version won on nine criteria out of eleven, lost on none, and the two it did not move were formatting. Groundedness up four points. Completeness up six. The regression review found nothing. The gate passed on every condition. It was the cleanest release candidate we had produced in a year.

We shipped it to fifty percent of traffic and the escalation rate went up.

Not dramatically. About eleven percent, sustained, holding across a week, well outside anything you could call noise. More people were giving up on the assistant and asking for a human, using a version of the product that was, by every measurement we had built over eighteen months, better.

It took us nine days to find it. The new version was more complete, which was the thing we had optimized. Completeness meant longer answers. Longer answers meant the direct answer to the question was now in the third paragraph rather than the first, and on a phone the third paragraph is below the fold, and a large fraction of users were not scrolling. They read two paragraphs of accurate context, failed to find their answer, and left.

Every offline metric had gone up. The product had gotten worse. The gap between those two sentences is what this chapter is about.

Offline vs. online evaluation

Offline evaluation predicts. Online evaluation observes. They are both necessary and they are not interchangeable, and the difference between what they measure is a permanent property of the discipline rather than a gap you can close with better tooling.

Offline is fast, cheap, safe, and repeatable. You can run it a hundred times before lunch, you can run it on a change that would be dangerous to expose to a user, and you can attribute a result to a cause because you controlled everything else. What it cannot do is tell you how a person will behave, because there is no person in it.

Online is slow, expensive, risky, and confounded. It takes days to accumulate a signal, it exposes real users to your mistake, and the result is contaminated by everything else happening in the product that week. What it can do is tell you the only thing that finally matters, which is whether people got what they came for.

So use them in the order their costs imply. Offline evaluation is a filter that catches the changes that are obviously bad and eliminates them at almost no cost. Online evaluation is the confirmation that the surviving changes are actually good. A team that only does offline evaluation is optimizing a proxy. A team that only does online evaluation is running experiments on customers with defects that a two-minute test would have caught.

And when they disagree, online wins, and the disagreement is information. It means your offline suite is measuring something that has come apart from the outcome, and the right response is to fix the suite, not to argue with the users.

A/B testing for LLM products

The mechanics of controlled experiments are a solved field, and the discipline has been written down thoroughly, and generative products break several of the assumptions in ways worth naming.

Variance is higher. Response quality varies within a single arm because generation is stochastic, which means the effect you are trying to detect sits on top of a noisier baseline than a typical interface experiment. That means longer runs and bigger samples than the product team will want to hear about.

The unit of randomization has to be the user, not the request. Randomizing per request means a single person gets the old system on turn two and the new one on turn three, in the same conversation, with different context handling, and the resulting experience is worse than either arm and your data is garbage. Randomize by user, hold the assignment for the session, and hold it across sessions if the product has memory.

Novelty and primacy effects are strong. A change to how an assistant talks will produce a reaction in week one that is about the change rather than about the quality, and it will decay. Run long enough to see the decay. A result that is large on day two and gone by day ten was a reaction, not an improvement.

And check that the experiment ran before you interpret it. A sample ratio mismatch, where the traffic split does not match the split you configured, is the single most common sign of a broken experiment, and Ron Kohavi and colleagues have made about as strong a case as can be made that ignoring it is how organizations reach confident wrong conclusions. If you asked for fifty-fifty and you got fifty-two forty-eight, stop, and find out why, before you read a single metric.

Canary launches

A canary is a safety check rather than an experiment, and confusing the two produces a canary that runs too long and an experiment that is underpowered.

The canary asks one question: is this obviously breaking anything. Small share of traffic, short duration, watching the guardrails, which are the metrics that can be read in real time and that indicate harm. Error rate. Latency at the tail. Safety intervention rate. Tool failure rate. Refusal rate.

It does not measure quality, and it cannot, because a one-percent canary over four hours does not accumulate enough signal to detect a three-point groundedness change. Asking it to is how teams conclude that a bad release is fine.

The canary's rollback is automatic and it fires without a meeting. If the metrics pipeline breaks and the guardrails cannot be evaluated, it rolls back too, because a system that keeps rolling out while blind is a system that will roll out an outage.

Then the experiment starts, at a larger share, for long enough to say something. Those are two different phases with different durations and different decision rules, and running them as one thing means you either roll back on noise or ship on nothing.

Guardrail monitoring

Guardrails are the metrics that must not get worse, regardless of what the primary metric does, and defining them in advance is what stops a team from talking itself into a bad launch.

A guardrail is different from a success metric in one specific way: you are not trying to improve it. You are watching it. A change that improves task completion by four points and increases the safety intervention rate does not ship, and the arithmetic that would trade one against the other should never be performed.

The standard set. Safety intervention rate. Over-refusal rate. Latency at the ninety-ninth percentile. Cost per conversation. Escalation rate. Error rate. Each has a threshold, each threshold is set from the current baseline before the experiment starts, and breaching one stops the rollout regardless of how good everything else looks.

Cost belongs on that list and it is the one teams leave off, and then discover, in the third week, that the winning variant costs sixty percent more per conversation and the finance team has questions. Put it in the guardrails, monitor it from the first day of the canary, and the conversation happens before the rollout rather than after.

User satisfaction metrics

Satisfaction is what you want to measure and every instrument you have for measuring it is bent.

Explicit ratings have a response rate low enough that the responders are a different population from the users, and they skew negative, because a person who is angry will click and a person who is fine will not. The absolute level of a thumbs-up rate tells you almost nothing. The change in it, between two arms, over the same period, is a real signal, because the bias is the same in both arms and it cancels.

That is the general principle and it rescues most of these metrics. A biased estimator is still useful for comparison, as long as the bias does not interact with the treatment. Where it does interact, and it sometimes does, you are in trouble: a variant that makes the assistant more likeable will get more thumbs up without answering any more questions, and now your bias is correlated with your treatment and your metric is measuring charm.

Survey a sample directly if the product can carry it. A short in-product question, at a low rate, produces a much less biased read than a rating widget, because you chose who to ask.

And do not confuse satisfaction with success. A user who receives a confidently wrong answer, believes it, and rates it highly is a satisfied user and a failed interaction, and no satisfaction metric on earth will ever show you that. It is the reason offline correctness measurement cannot be replaced by online sentiment, and it is the strongest argument for keeping both.

Task completion metrics

Completion is the closest thing to ground truth that production will give you, and it is worth a great deal of engineering effort to define it well.

It has to be defined per intent, because completion means different things for different tasks. For a support flow, the issue was resolved without a human. For a shopping flow, the user proceeded toward a purchase. For an informational query, the user did not immediately ask the same thing again in different words. Each of these is an inference from behavior and each is imperfect and together they are the best signal you have.

Instrument the negative signals with equal care, because they are cleaner. Escalation to a human is unambiguous. Rephrasing the same question is close to unambiguous. Abandonment mid-flow, after a long answer, is a strong indicator. Each of these is a failure the user reported to you with their behavior, at no cost to you, and most products do not count them.

The escalation rate is, in my view, the single most useful online metric a conversational product has. It is unambiguous, it is expensive for the user to produce, it correlates with everything you care about, and it goes down when the product gets better. If you build one online metric, build that one.

A specific and unpleasant thing that will happen to your experiment, and it is worth being ready for it.

The result will be null. Not negative, not positive, null: the change made no measurable difference to any online metric, and the confidence interval spans zero, and nobody can say whether it helped. This is the most common outcome of a well-run experiment on a mature product, and it is not a failure of the experiment.

The wrong response is to keep looking for a slice where the number is positive, which you will find, because if you cut the data enough ways one of them will show a win by chance. The right response is to state that the change did not move the metric, ship it if the offline evidence supports it and the guardrails are clean, and be honest that its value is unproven. An organization that cannot say we do not know will teach its people to find a positive result, and they will.

Business metrics vs. quality metrics

Sooner or later somebody will show you a chart where the business metric went up and the quality metric went down, and ask you what to do.

The first thing to do is take it seriously rather than assuming the business metric is wrong. Sometimes the quality metric is measuring the wrong thing, and the users are voting with their behavior, and they are right and you are not.

The second thing is to look for the mechanism, because a divergence has one, and the mechanism is usually uncomfortable. A more confident assistant produces more conversions and more wrong answers, because confidence sells and hedging does not. A shorter assistant produces more completions and less accuracy, because people act on the first thing they read. A more agreeable assistant produces better satisfaction scores and worse outcomes, because it tells people what they want to hear. Every one of these is a real tradeoff and every one of them is a decision somebody has to make on purpose.

And this is precisely where a quality metric earns its existence. A team that only measures business outcomes will optimize toward whichever behavior converts, and the local optimum for conversion is an assistant that is confident, brief, and agreeable, which is a description of a system that is wrong a lot and pleasant about it. The quality metric is the counterweight, and its job is to make the tradeoff visible so that a human decides it rather than a gradient.

Decide the tradeoff in advance and write it down. Which guardrails are inviolable. What quality drop, if any, is acceptable for what business gain. Having that conversation while nobody is under pressure produces a different answer than having it in the meeting where a launch is at stake, and the first answer is the one you should be governed by.

A design decision about experiments that is worth making once, carefully, and then never revisiting under pressure.

Decide in advance what would make you abandon the change. Not what would make you ship it, which everyone plans for, but what result would cause you to conclude the idea was wrong and stop. Write it down before the experiment starts, next to the success criterion, in the same document.

The reason is that an experiment with a defined success criterion and no defined failure criterion has only one exit, and teams will find it. The change will be rerun on a different slice, extended for another week, redefined as a learning, or shipped on the grounds that the guardrails were

clean. A pre-registered stopping rule is the only thing that makes an experiment capable of telling you no, and an experiment that cannot tell you no was not an experiment.

Interpreting noisy user feedback

User feedback is the noisiest data in your system and it is treated with the least statistical care of anything you collect.

Slice it before you believe it. An overall thumbs-down rate that is flat can be hiding a doubling in one intent and an improvement in another. Segment by intent, by conversation depth, by cohort, by device. And be careful about the aggregation itself, because a metric can move one way in every segment and the other way in the total, purely because the segment sizes shifted. That is Simpson's paradox and it will happen to you, and it is why the segment view has to be the default rather than the drill-down.

Beware of stopping when you like the answer. Checking an experiment repeatedly and calling it as soon as it crosses significance inflates your false positive rate substantially, and it is the most common statistical error in industrial experimentation. Fix the sample size in advance, or use a sequential method designed for continuous monitoring, and do not decide by looking.

And apply Twyman's law with real discipline. Any figure that looks surprising is probably wrong. A twenty-point improvement is not a triumph, it is a bug in your instrumentation, and it will be, roughly nine times out of ten. Investigate your wins with the same energy you investigate your losses, and you will find that a good fraction of them are logging changes, routing changes, or a variant that quietly stopped serving one segment.

One asymmetry in online evaluation deserves its own note, because it will shape how quickly you can move.

Detecting harm is fast and detecting benefit is slow. A safety regression, a latency regression, or an error spike shows up in minutes, because the signal is large and the baseline is tight. A three-point improvement in answer quality shows up, if you are lucky, in a couple of weeks of accumulated behavioral data, because the signal is small and the noise in human behavior is enormous.

The practical consequence is that your rollback machinery can be automatic and your ship decision cannot. Arm the guardrails to fire in seconds. Then be patient about the win, and resist the pressure to call the experiment early, because an experiment stopped at the moment it looked good is an experiment that measured your patience rather than your product.

When to rollback

Write the rollback criteria before the launch, because a rollback decision made during a launch will be made by people who have a stake in not rolling back.

Automatic and immediate: any safety guardrail breach, error rate above threshold, latency at the tail above budget. No meeting. The system does it and tells you afterward.

Automatic within an hour: escalation rate elevated beyond a bound and sustained, cost per conversation beyond budget. These need a window rather than an instant, because they are noisy at short horizons, and a trigger that fires on noise will be disabled by an irritated engineer within the week.

Human decision, with a deadline: quality signals that are ambiguous, business metrics that moved in opposite directions, a divergence between offline and online results. These are

judgment calls and they belong to a named person, and the deadline matters as much as the name, because a decision that stays open is a decision to keep the change out.

The default, in the absence of a decision, is rollback. That single rule fixes more organizational dysfunction than any process document, because it inverts the burden. Under any other default, a change stays out while people argue, and the arguing takes days, and the users pay for every hour of it.

Phase	Duration	What you watch	Decision rule
Offline gate	Under an hour	Full suite, per criterion, per stratum, per severity	Block on any gate condition
Shadow	Days	Divergence from production on live traffic; read the biggest	Qualitative; find surprises
Canary	Hours	Guardrails only: errors, latency p99, safety, refusals, tool failures	Auto-rollback, no meeting
Experiment	1 to 3 weeks	Escalation, task completion, satisfaction, cost; all segmented	Fixed sample size, set in advance
Sanity check	Continuous	Sample ratio mismatch	Stop and diagnose before reading anything
Novelty check	Full run	Effect size over time	Discard effects that decay to zero
Guardrails	Continuous	Safety, over-refusal, latency, cost, errors	Breach stops the rollout regardless of the win
Segment review	At readout	Every metric by intent, depth, cohort, device	A flat total can hide a doubled failure
Divergence review	At readout	Offline said better, online said worse, or the reverse	Online wins; then go fix the offline suite
Ramp	Staged	Hold at each step long enough for a signal	No step without a read on the previous one
Rollback default	Always	Absence of a decision	Roll back

The last row is the one to fight for. Every other line in that table describes a measurement. That one describes an organization, and it is the one that will be argued with, and it is the one that decides whether any of the rest of it does anything.

Everything to this point has described what one team should do. Which leaves the question that arrives the moment a company has more than one team shipping generative features, and it is a harder question than any of the technical ones.

Part Six

Scaling Evaluation Across an Organization

Chapter 21: Building an Evaluation Platform

There were four teams shipping generative features and four evaluation setups, and I found this out in a meeting where somebody presented a quality score of eighty-eight percent and somebody else asked, mildly, eighty-eight percent of what.

The first team scored on a five-point scale and reported the mean. The second scored binary and reported the pass rate. The third used a judge model with a rubric nobody outside the team had read. The fourth used a spreadsheet, maintained by one engineer, containing three hundred cases and a column of scores that had been entered by hand over eleven months by at least five different people.

Every number was defensible inside its own team. None of them were comparable to each other. Leadership had a slide with four percentages on it, arranged in a row, and it was entirely reasonable to read that slide as a ranking, and the ranking was meaningless, and decisions were being made from it.

The failure was not that any team was doing bad work. Three of the four were doing careful work. The failure was that the organization had no shared instrument, and without a shared instrument, quality is a matter of local opinion, and local opinions do not add up to a company that can tell whether it is getting better.

Why spreadsheets do not scale

A spreadsheet is the right first tool and it is worth being clear about why, because the argument for building a platform is usually made too early and by the wrong person.

For one team with two hundred cases, a spreadsheet is faster than anything you could build, it requires no infrastructure, and it lets you change the schema in the middle of an experiment. Anyone who tells a three-person team to build an evaluation platform before they have run their first fifty cases is giving them a way to spend two months not measuring anything.

It breaks at a set of specific thresholds, and they arrive in a predictable order.

It breaks when scores need to be reproducible. A spreadsheet records a number and not the thing that produced it. Which model, which prompt version, which dataset revision, which judge, which rubric. Six months later somebody asks whether quality has improved since the spring, and the honest answer is that nobody can tell, because the numbers from spring were computed against artifacts that no longer exist and were never named.

It breaks when the volume passes what a human can enter. Two thousand cases scored on seven criteria is fourteen thousand cells, per run, and you will run it weekly, and somebody has to paste them in.

It breaks when more than one person edits it, which produces the specific horror of two engineers with two local copies and a divergent version of the truth.

And it breaks the moment a score has to gate a release, because a gate is an automated decision and a spreadsheet is not a system that anything can call.

The signal that you have crossed the threshold is usually a sentence rather than a metric. Somebody says, in a meeting, that they are not sure the number from March was computed the same way as the number from June, and everybody goes quiet, because nobody can prove that it was. That sentence is the moment. Before it, a platform is premature optimization. After it, every week you delay is another week of numbers that will later have to be thrown away.

Dataset versioning

The registry is the foundation and everything else in the platform is built on top of it, so build it first and build it properly.

A dataset has a version. The version is immutable. A score is always recorded against a specific version, and a score whose dataset version is unknown is a number that gets thrown away rather than plotted. That is the entire discipline, and it is the one that most often gets compromised for expedience.

Store cases as structured records with the schema from Chapter Four, in version control, reviewed through pull requests, because a change to an expected behavior is a change to what your product is supposed to do and it deserves the scrutiny that a change to a production threshold would get. When a case is edited, that is a new version, and the old version is retained, so that a score computed against it remains interpretable.

Run the two tracks from Chapter Four. A frozen track for the trend line that goes to leadership, changing rarely and by explicit decision, with the re-baselining visible on the chart. A living track that absorbs new cases continuously, used for development. The platform holds both and knows which is which, and the dashboard says which one a number came from.

Timnit Gebru and colleagues argued in Datasheets for Datasets that a dataset should ship with documentation of its provenance and intended use. In a platform, that documentation becomes a required field rather than a good intention, and requiring it is the only way it gets written.

Evaluation execution engine

The runner is the component that takes a version tuple and produces scores, and it is mostly a distributed systems problem wearing an evaluation costume.

It resolves versions. Given a request to evaluate prompt version fourteen against model B on dataset version nine with rubric version three, it fetches exactly those artifacts and records exactly what it fetched. This resolution step is what makes a result reproducible, and skipping it is what makes an evaluation program a source of numbers nobody can defend.

It caches contexts. The assembled context that produced each response gets stored, because Chapter Fourteen needs it for model comparison and Chapter Fifteen needs it for the retrieval-versus-generation split, and regenerating it later against a moved index is not the same thing.

It handles the boring infrastructure realities that will otherwise eat your evaluation program alive. Rate limits, because a two-thousand-case run against a provider will hit them. Retries with backoff, because calls fail. Partial results, because a run that dies at case seventeen hundred should not discard seventeen hundred cases of work. Parallelism, because a serial run of a large suite takes a day and a suite that takes a day runs weekly and a suite that runs weekly does not gate anything.

And it records cost per run, in tokens and in money, because the first question anyone in finance will ask is what this is costing and the second is why it went up.

One design decision here has an outsized effect on how useful the platform is a year later, and it is easy to get wrong on the first pass. The runner should execute cases through the same code path the product uses, rather than through a parallel implementation that reconstructs the pipeline for evaluation purposes. Every team is tempted by the parallel implementation, because it is simpler and it does not require the production service to be callable in a test context. And

every team that builds one discovers, eventually, that the two paths have drifted, and that they have spent a year measuring a system that is not the system their users are talking to.

If the production path cannot be invoked from the evaluation harness, that is a defect in the production path, and fixing it is cheaper than the alternative.

Scoring service

Scoring is a separate service from execution, and separating them is the design decision that most improves the platform's usefulness over the following two years.

The reason is that you will want to re-score. A rubric changes. A judge model is upgraded. A criterion is added. If scoring is welded into the runner, re-scoring means regenerating every response, which costs real money and introduces new sampling variance and means you are no longer comparing the same outputs. If scoring is separate, you re-score stored responses against the new rubric for a fraction of the cost, and you can compare old and new rubrics against identical outputs, which is the only way to tell whether a rubric change was an improvement or a redefinition.

The service holds three kinds of scorer behind one interface. Deterministic checks, which are free and instant. Judge calls, which are metered and batched and which carry the position-swapping and prompt discipline from Chapter Twelve. And human review tasks, which are dispatched to a queue and which return asynchronously, hours or days later, and which the platform has to be designed to wait for without blocking anything.

Every score carries its provenance: the scorer, the scorer's version, the rubric version, and the timestamp. A score without provenance is a rumor.

A property of the metrics store that seems like a detail and turns out to be the thing that makes the platform trustworthy.

It is append-only. Scores are never edited, never overwritten, never recomputed in place. A re-scored response produces a new record with a new scorer version and a new timestamp, and the old record remains, and both are visible.

The reason is that a mutable metrics store is a store in which history can be rewritten, and history in an evaluation program is the whole asset. When somebody asks whether quality has improved since March, the answer is a query against records that have not moved, and nobody has to take anyone's word for it. When the store is mutable, that question becomes a matter of trust, and trust is precisely what a measurement system exists to replace.

Comparison reports

The output that people actually use is the comparison, and getting it right does more for adoption than any other feature.

The report answers one question: what did this change do. It shows the paired crosstab from Chapter Thirteen, fail to pass and pass to fail and unchanged. It shows the aggregate delta with its confidence interval and the published noise floor next to it. It breaks the delta out by criterion, by stratum, and by severity, because that is where regressions hide. And it links every regressed case to its full trace, so that a reviewer can click through and read what happened rather than trusting a number.

Deliver it where the decision is made, which is the pull request. A comparison report that lives in a separate tool requires an engineer to remember to go and look at it, and under deadline they

will not. A comment on the pull request, posted automatically, showing eleven cases that got worse, changes the review conversation in a way that no dashboard has ever managed.

And make it fast. The report has to be there when the author is still holding the change in their head. Twenty minutes is too slow. Five is the target, and hitting it is an engineering constraint that should shape how you tier the suites.

Human review workflow

The human review interface is the component that most teams skip, and skipping it means your expensive reviewers spend a third of their time on data entry.

It presents one case at a time, with everything needed to judge it and nothing else: the query, the conversation history, the retrieved evidence, the response, and the criteria as explicit questions with buttons. Not a spreadsheet cell. A question with an answer, because the criterion is a question and the reviewer's job is to answer it.

It routes by skill, sending policy criteria to specialists and readability criteria to a general pool, per Chapter Nineteen. It manages the overlap, silently double-assigning ten to twenty percent of cases so that agreement can be computed without anyone changing their behavior. It plants gold cases at a published rate. It tracks time per case and per-reviewer accuracy and agreement over time, and it surfaces those to the review lead rather than burying them.

And it enforces the cap. A platform that lets a reviewer score four hundred cases in a day is a platform that will corrupt its own ground truth, and the fix is a limit in code rather than a note in a document.

Integration with CI/CD

A platform that is not wired into the release pipeline is a research tool, and it will be used by the people who like it and ignored by everyone else.

The integration is an interface with three operations. Evaluate a version tuple and return a result. Compare two version tuples and return a report. Gate a version tuple against a threshold and return a pass or a block with a reason.

That third one is the whole point, and it has to be callable from the deployment system without a human in the path. The gate conditions from Chapter Sixteen live in configuration, owned by the evaluation owner, versioned like everything else, and changing a threshold is a pull request that the evaluation owner has to approve. A threshold that can be changed by the team whose launch it is gating is not a threshold.

Every gate decision is recorded, with the result, the thresholds in force, and the identity of anyone who overrode it. The override log is the artifact that makes the monthly review from Chapter Sixteen possible, and it is the artifact that makes the gate credible, because a gate whose overrides are invisible is a gate that will be overridden constantly.

A build-versus-buy note, since the question will be asked and the honest answer is unsatisfying.

The tooling market for this has grown quickly and much of it is good. What none of it will give you is the part that matters: your dataset, your rubrics, your judge calibration against your reviewers, and your failure taxonomy. Those are the assets. The runner, the store, and the dashboard are commodity infrastructure that a vendor can supply better than you can.

So buy the infrastructure if it saves you time, and never let the vendor own the assets. Keep the dataset in your own version control, in a format you could export tomorrow. A team whose

evaluation dataset lives inside a product they cannot leave has made their most valuable and least replaceable asset into a hostage.

Governance and access control

Component	What it owns	The decision that makes it work
Dataset registry	Cases, versions, strata, review dates	Versions are immutable; a score without one is discarded
Prompt registry	Every prompt block, versioned, with owners	Blocks have separate owners so the right reviewer is requested
Model registry	Model identifiers, sampling parameters, provider config	Pinned per run, so a silent provider change is attributable
Rubric registry	Criteria, boundary cases, adjudicated disagreements	The rationale is stored, not just the rule
Evaluation runner	Version resolution, execution, caching, retries, cost	Executes through the production path, never a parallel one
Scoring service	Deterministic checks, judge calls, human review tasks	Separate from execution, so stored outputs can be re-scored
Human review interface	One case at a time, criteria as questions, routing by skill	Enforces the daily cap in code, not in a document
Metrics store	Every score with its scorer, rubric version, and timestamp	Append-only; history cannot be rewritten
Comparison reports	Paired crosstab, per-criterion deltas, noise floor, links to cases	Posted to the pull request, not to a separate tool
Release gate	Thresholds, risk tiers, pass or block with a reason	Called by the deployment system; the safety tier cannot be overridden
Audit log	Every dataset edit, rubric change, threshold change, and override	Not editable by the person making the change
Access control	Who may edit, change, and override what	The owner does not report into the org whose launches it gates

The last two rows of that table are the ones people skim past, and they are the ones that determine whether the platform means anything after eighteen months.

The question is who may change what. Who may edit a case in the frozen dataset, because a person who can edit the dataset can make any number say anything. Who may change a rubric, because a rubric change silently redefines every score computed after it. Who may change a gate threshold. Who may override a gate. Who may see raw conversation logs, which contain whatever your users typed and are therefore among the most sensitive data your company holds.

Answer each of those with a named role rather than with a team, and enforce it in code. And record every change, with a who and a when and a why, in a log that the person making the change cannot edit.

This matters for a reason that has nothing to do with bad actors. It is the completely ordinary pressure of a deadline. A team that is one point below the gate, at four in the afternoon, on the day of a launch, has a spectrum of options in front of them, and one of those options is to edit three test cases that are being unfair. They will not think of it as cheating. They will think of it as

fixing a bad case, and sometimes they will even be right, and the difference between those two situations is exactly why the change should require a second person and leave a record.

Conway observed in 1968 that organizations produce systems that mirror their own communication structures, and an evaluation platform is the clearest example of this I know. If evaluation is owned by the team whose work it grades, the platform will be built to grade generously, and no one will have decided to do that. It will simply be what a system built by that structure produces.

So the platform has an owner, and the owner does not report into the teams being measured, and the platform's access controls encode that separation. Everything else here is engineering. That last part is politics, and it is the part that decides whether the engineering was worth doing.

A platform that measures quality well will, within a quarter, be asked a question it was not built to answer. Somebody will point at a number that is going up and ask what it costs.

Chapter 22: Cost, Latency, and Quality Tradeoffs

The proposal was to add a second retrieval pass and a verification step, and the offline numbers were excellent. Groundedness up seven points, which was the largest single improvement anyone had produced that year, on the criterion that mattered most.

The finance partner asked one question, which was what it cost, and nobody in the room knew.

It took us two days to work out. The second retrieval pass added roughly forty percent to the token count of the assembled context. The verification step was a second model call on every response. Together they raised the cost per conversation by a factor of about two point three, and at our volume that was a number with a lot of digits in it, and it was larger than the entire annual budget of the team proposing the change.

We did not ship it. We shipped a version that ran the verification step only on high-risk intents, which captured most of the groundedness gain for about fifteen percent of the additional cost, and we would never have found that design if the question had not been asked.

The point of this chapter is that the question should be asked automatically, by your evaluation system, on every proposal, rather than by an alert person in a meeting.

Quality is not the only metric

Every technique in this book buys quality by spending something.

Retrieve more documents and you improve recall and you lengthen the context, which costs tokens, adds latency, and, past a point, degrades the model's ability to use any of them. Add a reranker and you improve precision and you add fifty to two hundred milliseconds. Add a verification pass and you catch unsupported claims and you double your model calls. Use a larger model and everything improves except the two numbers your business cares about.

There is no free improvement in this discipline. There are only trades, and a team that reports quality without reporting what quality cost is presenting half of a decision and calling it a recommendation.

So make the triple the default unit of reporting. Every evaluation run emits quality, latency, and cost, together, always, and a proposal that improves one and degrades the others is a tradeoff to be decided rather than a win to be celebrated. This is a small change to a reporting template and it changes what gets built, because engineers optimize what is measured and if only quality is measured they will spend arbitrary amounts of money improving it.

Token cost evaluation

The price per token on the vendor's page is the least useful number in this discussion, and teams anchor on it because it is the one that is easy to find.

What you pay is price per token multiplied by tokens consumed, on your traffic, at your context lengths. A model that is thirty percent cheaper per token and produces forty percent more output is more expensive. A model that is more expensive per token and needs fewer retries because it follows instructions reliably may be cheaper. The only way to know is to run your own traffic through both and count.

Decompose the count, because the components grow for different reasons and by different mechanisms. System prompt tokens, which are paid on every single request and which grow every time somebody adds an instruction after an incident. Conversation history tokens, which

grow with depth. Retrieved evidence tokens, which grow whenever anybody increases k. Output tokens. And reasoning tokens, which you pay for, cannot see, and which can exceed everything else combined.

The system prompt deserves a specific mention because it is the one people forget. A four-thousand-token system prompt on a product handling millions of conversations is a fixed tax on every request, forever, and about a third of it is usually instructions that no test case depends on. The prompt audit from Chapter Thirteen is a cost-reduction exercise as much as a quality one.

And measure cost per successful conversation rather than cost per request. A cheaper configuration that causes users to ask twice is not cheaper. That framing has flipped more than one decision I have been part of.

Latency budgets

A latency budget is a fixed sum allocated across the stages of the pipeline, and writing it down is what turns latency from an emergent property into an engineering constraint.

Start from the user requirement, which is a product decision rather than an engineering one. For a conversational assistant, a first token inside about a second feels responsive and beyond two seconds feels broken. Take that number, subtract network and rendering overhead, and what remains is what the pipeline gets to spend.

Then divide it. Intent classification gets a slice. Retrieval gets a slice. Reranking gets a slice. Context assembly gets a slice. The model's time to first token gets the largest slice, and it is the one you have the least control over. Safety checks on the input get a slice, and safety checks on the output get a slice that has to be spent before the first token reaches the user, which is a constraint people discover late and painfully.

Every stage is monitored against its budget, and a change that exceeds its slice is a regression regardless of what it did for quality. This is the same mechanism as the release gate, applied to a different number, and it works for the same reason: it converts a judgment call under pressure into a threshold set in advance.

Budget at the ninety-ninth percentile, not the median. A pipeline whose stages each meet their median budget will blow the total at the tail, because the tails compound, and the tail is where the timeouts and the abandonment are.

First-token latency

This number deserves its own section because it is the one that determines whether users think your product is fast, and it is the one most commonly left uninstrumented.

In a streaming interface, everything before the first token is dead time. The user is looking at a cursor. Their patience is being spent, and it is being spent at a rate that has nothing to do with how good the eventual answer is. A response that begins in four hundred milliseconds and takes eight seconds to complete will be perceived as faster than one that begins in three seconds and completes in four, and the second one is objectively quicker.

The architectural consequence is that everything you do before generation starts is expensive in a way that everything after it is not. A safety check on the input, a retrieval pass, a reranking step, a context assembly that reads from three services: all of that lands squarely on the number the user feels. A post-processing step that runs on the completed text does not, as long as you are willing to stream before it completes, which raises the question of what happens if the post-processor wants to redact something already on the screen.

That tension is real and it does not have a clean resolution. Streaming and output safety filtering pull against each other, and the usual compromise is to buffer a small window, filter the window, and stream it, accepting a delay of a few hundred milliseconds. Whatever you choose, choose it deliberately and measure both sides of it, because a team that streams without filtering has made a safety decision without noticing.

Model size tradeoffs

The default assumption is that the largest model available is the best choice and that anything else is a compromise. It is worth attacking that assumption directly, because it is expensive and it is often wrong.

Not every request needs the same model. An intent classification does not. A groundedness check on a single claim does not. A format compliance judgment does not. A reranking decision does not. All of these are narrow tasks where a small model performs close to a large one, and all of them are called on every request, and the cost difference across millions of requests is the difference between a viable product and one that gets cancelled.

So route. Send the easy requests to a small model and the hard ones to a large one, with a classifier deciding, and evaluate the routing decision as a component with its own accuracy. The failure mode is a hard request routed to the small model, and the evaluation question is how often that happens and what it costs when it does. If the small model handles seventy percent of traffic at a fifth of the price with no measurable quality difference on that seventy percent, the routing has paid for the entire evaluation program.

Evaluate the small model honestly, on your traffic, per criterion. It will be worse at some things and indistinguishable at others, and the distinction is the whole finding. In my experience small models hold up well on extraction, classification, entailment, and formatting, and fall apart on multi-step reasoning, long-context synthesis, and anything requiring judgment about ambiguity. Your mileage will differ and you should measure rather than assume.

Retrieval depth tradeoffs

How many documents should you retrieve is a question with a non-monotonic answer, and that surprises people.

More documents means higher recall, which is good. It also means more tokens, which costs money and latency. And, past a point, it means worse answers, because the relevant passage is now surrounded by distractors and the model has to find it, and models get worse at that as the context grows, particularly in the middle.

So there is an optimum, and it is not at the maximum, and it is different for every product. Find it empirically. Sweep k across a range, hold everything else fixed, and plot quality against k. The curve rises and then flattens and then, often, declines. The flattening point is where you should be, and it is usually much lower than the default somebody set eighteen months ago.

Do the same sweep for chunk size and for reranker depth, which is the practice of retrieving many candidates cheaply and reranking a smaller set expensively. Retrieve fifty, rerank to five, and you get the recall of fifty with the token cost of five, at the price of the reranker's latency. That is one of the better trades available in this whole chapter and many teams have never run the sweep that would reveal it.

A number worth computing once, because it will reframe every cost conversation you have afterward.

Cost per resolved request. Not per request, per resolved request: the total spend divided by the number of interactions where the person actually got what they came for. A configuration that is thirty percent cheaper per call and causes twelve percent more people to ask again, or escalate, or give up and contact a human, is more expensive by the only measure that matters.

Most teams have never computed it, because it requires joining the cost data to the outcome data, and those live in different systems owned by different people. The join is worth an afternoon. It has flipped more than one architectural decision I have been part of, always in the direction of spending more per call and less overall.

Caching strategies

Caching is the one place where you can improve latency and cost together without giving up quality, which makes it unique in this chapter and worth more attention than it usually gets.

Prompt caching, offered by most providers, stores the processed representation of a stable prefix so that it is not recomputed on every call. Since your system prompt is identical across every request and can be thousands of tokens, this is a large win and it is close to free, and getting it requires structuring the prompt so the stable part comes first and the variable part comes last. That is a fifteen-minute refactor that many teams have not done.

Retrieval caching helps because query distributions are heavily skewed. A small set of queries accounts for a large share of traffic, and their retrieval results change only when the index changes. Cache them, key on the query and the index version, and invalidate on reindex.

Response caching is the tempting one and it is the one that needs care. Identical questions asked by different users in different contexts do not have identical answers, and serving a cached response to a user whose conversation history differs is a way to produce a subtly wrong answer very quickly. Cache with the full context in the key, or do not cache.

And evaluate the cached path, because a cache is a code path and code paths have bugs. A stale cached response is a correctness failure that will be invisible in your evaluation suite, because your suite runs against a cold cache.

The most expensive component in most evaluation programs is one that nobody puts in the budget.

Human review. A reviewer scoring eighty cases a day, with the overlap, calibration, and adjudication overhead from Chapter Nineteen, is a substantial recurring cost, and it is the line item that gets cut first, because it looks like a process expense rather than an engineering one.

Cutting it is the most expensive saving available to you. Human review is what calibrates the judges, and the judges are what produce every number on your dashboard. Remove the anchor and the numbers keep arriving, looking exactly as they did before, meaning progressively less. There is no alarm for this. The first indication that it has happened will be a production failure in a category your dashboard says is healthy.

When a cheaper model is good enough

The question in the section title is the one that decides your unit economics, and it has an answer, and the answer is empirical.

Run the paired comparison from Chapter Thirteen: the same cases, the same contexts, the small model and the large one, scored per criterion. Then look at where the small model loses. If it loses uniformly, a little bit everywhere, the large model is worth its price. If it loses

concentratedly, badly on one intent and not at all on the others, then the answer is a router rather than a model.

The concentrated case is the common one, and it is the one people miss because they look at the aggregate. A small model that scores four points below the large one overall might be identical on eighty percent of traffic and catastrophic on the other twenty, and the correct response to that is to send the twenty percent to the big model and pocket the difference on the rest.

Frame the decision as an amount of money buying an amount of quality. The large model costs an additional sum per year and buys a specific improvement on a specific criterion. Is that a good trade? That is a business question, and it should be asked in those terms, to a person whose job it is to answer questions in those terms, and they will frequently give you an answer that engineering would not have reached.

The most reliable cost saving in this chapter is the one nobody proposes, because it looks like a reduction in ambition.

Do less per request. A large fraction of production generative systems perform work on every request that is only needed on a minority of them. A second retrieval pass that helps on complex questions and is wasted on simple ones. A verification step that matters for factual claims and does nothing for a greeting. A reasoning mode that earns its cost on a multi-step problem and burns tokens on a lookup.

Route by need. Classify the request, cheaply, and spend the expensive machinery only where it changes the answer. In every system I have looked at, this has been available, has been worth a large fraction of the total cost, and has been overlooked, because it is an architectural change rather than a tuning change and it requires somebody to admit that most requests are easy.

Leadership-facing tradeoff reports

The last section is about communication, and it decides whether any of the analysis in this chapter changes what the company does.

A leader does not want a rubric. They want to know what they are buying, what it costs, and what the risk is if they decline. So give them exactly that: the option, the quality change, the cost change, the latency change, and the risk of not doing it, in one row per option.

Present options rather than a recommendation with alternatives attached as decoration. Three configurations, honestly costed, with the tradeoffs stated. A leader who is given one option is being asked to approve rather than to decide, and they know it, and they will push back on the number rather than engaging with the choice.

Include the do-nothing option and cost it, because it has a cost. The status quo is failing a measurable fraction of a measurable population, and quantifying that is what turns quality from a nice-to-have into a line item.

And be honest about the error bars. A quality gain of four points, plus or minus three, is a different proposition from four points plus or minus half, and a leader who later discovers you presented the first as if it were the second will not fund your next request.

Lever	Quality effect	Cost effect	Latency effect	When it wins
Larger model	Up, unevenly by criterion	Up sharply	Up	Concentrated hard traffic only
Model routing	Flat if routed well	Down sharply	Down on the easy path	Traffic is mixed in difficulty

Lever	Quality effect	Cost effect	Latency effect	When it wins
Small model for judging	Flat on easy criteria	Down sharply	Down	Format, length, entailment checks
Increase retrieval k	Up, then flat, then down	Up	Up	Only up to the flattening point
Add a reranker	Up, often substantially	Slightly up	Up 50-200ms	Almost always worth measuring
Retrieve wide, rerank narrow	Up	Down vs. wide context	Up modestly	Usually the best trade available
Second retrieval pass	Up on groundedness	Up sharply	Up sharply	High-risk intents only
Verification pass	Up on groundedness	Doubles model calls	Up sharply	High-risk intents only
Longer system prompt	Up, then down	Fixed tax on every request	Up	Rarely; audit and cut instead
Prompt caching	None	Down	Down	Always; restructure the prompt for it
Retrieval caching	None if invalidated correctly	Down	Down	Skewed query distributions
Response caching	Risk of stale or context-blind answers	Down sharply	Down sharply	Only with full context in the key
Streaming	None	None	First-token latency down	Always, with a filtering plan
Output safety filter	Up on safety	Small	Up on first token	Buffer a window; measure both sides
Do nothing	Current failure rate persists	None	None	Cost it explicitly; it is not free

The last row is there on purpose. Every proposal in this chapter competes against doing nothing, and doing nothing has a price that is paid by users rather than by the budget, which is why it usually wins the argument and should not.

One category of cost has been left out of this chapter entirely, and it is the one that is hardest to put a number on and most expensive when it lands.

Chapter 23: Governance, Safety, and Compliance Evaluation

The lawyer asked a question I could not answer, and she asked it pleasantly, which somehow made it worse.

She wanted to know, for a specific customer, on a specific date, what the assistant had told them and on what basis. Not roughly. Precisely. The exact response, the exact evidence the system had relied on, the version of the policy in force at that moment, and a record of whether any safety or policy check had been applied to the answer before it reached the customer.

We had the response. We had the timestamp. Everything else was gone, or had never existed, and the honest answer to her question was that our system had told a customer something and we could not now reconstruct why.

Nothing bad had happened. This was a routine inquiry, and it resolved uneventfully, and it was the most useful hour I spent that year. Because the question she asked is the question that gets asked after something bad has happened, and by then it is far too late to start building the answer.

Safety categories

Safety is a word that hides a dozen different problems with different owners, and the first job is to split it.

Content harm is the category the model providers work hardest on, and your base model will refuse most of it. Violence, self-harm, exploitation, illegal activity, targeted hate. You still evaluate it, because your application composes the model with retrieval and tools, and composition creates surface that the provider never tested.

Domain harm is yours alone and nobody else will build it for you. Financial advice that a licensed person is supposed to give. Medical guidance that requires a clinician. Legal interpretation. Safety-critical instructions about a physical product. These are harmful in your context and unremarkable outside it, and no general safety classifier knows anything about them.

Data harm is exposure. The assistant reveals another user's information, or its own system prompt, or the contents of a document the user was not authorized to see. This is the category that grows fastest as you add retrieval and tools, and it is the one that produces the worst headlines.

Commercial harm is the promise you cannot keep. The assistant offers a refund policy that does not exist, quotes a price that is wrong, guarantees a delivery date, makes a claim about a competitor. Each of these is a statement the company is bound by, made by a system with no authority to bind it, and the legal exposure is real.

Four categories, four sets of test cases, four owners. A single safety score across all of them is a number that will be green while one of the four is on fire.

Sensitive data handling

Data flows through a generative system in more directions than a diagram usually shows, and each direction is a place it can escape.

It flows in, when the user types their account number into a chat box because they have been asked for it a thousand times before by a thousand other systems. It flows sideways, when

retrieval returns a document the user should not see. It flows out, when the response repeats something from another user's session because a cache key was wrong. It flows onward, when a tool call passes a field to a third-party service that did not need it. And it flows into your logs, where it sits, queryable by anyone with access.

Test each direction. Plant synthetic personal data in synthetic inputs and confirm it is redacted before it reaches storage. Query as a user without permission and confirm that no restricted document is retrievable, and confirm it at the retrieval layer rather than trusting the model to decline. Assert that tool calls carry the minimum fields the tool requires. And run the extraction attacks: users asking the assistant to reveal its instructions, its evidence, or what it knows about other people.

The principle underneath all of this is data minimization, which appears in essentially every modern privacy framework and which says that you collect what the purpose requires and nothing more. It is a design principle before it is a compliance requirement, and a system built to it has far less to go wrong.

Policy compliance

These are the rules your company has, which the model has never heard of, and which nobody has ever written down in a form a machine could check.

Start by writing them down. This sounds trivial and it is the hardest part, because the rules live in people's heads and in a training document from four years ago and in the practice of a support team that has evolved past both. Getting a list of the actual rules requires sitting with the people who enforce them and asking, one at a time, what must the assistant never say.

Each rule becomes a criterion, and each criterion needs cases designed to tempt the system into breaking it. A rule that has never been tested against a determined attempt to break it is a rule you have no evidence for. Generate the temptations synthetically, per Chapter Seven, because a human will write five and a generator will write five hundred.

Then decide where the rule is enforced, which is a design question with a clear right answer. A rule in the prompt is a suggestion that the model follows almost always. A rule in a downstream check is a guarantee. For anything with legal or financial consequence, put it in the check, and treat the prompt as an optimization that reduces how often the check has to fire.

And track policy versions against your evaluation cases, the way Chapter Five insists. When a policy changes, the cases that depend on it are wrong until somebody updates them, and a suite that penalizes the system for following the current policy will send your team on a hunt for a bug that does not exist.

Refusal quality

A refusal is a product experience and most teams evaluate it as a binary event, which throws away almost everything that matters about it.

There are four things to score and only one of them is whether the refusal happened.

Did it refuse when it should have. This is the safety rate and it is the one everybody measures.

Did it answer when it should have. This is the over-refusal rate and it is invisible to a safety dashboard, because from a safety dashboard's perspective a refusal is always a success. A system that refuses ten percent of legitimate questions is failing ten percent of its users, silently,

and the only way to see it is to build a set of cases that should be answered and measure how many are declined.

Did the refusal explain itself. A refusal with no reason reads as a malfunction. A refusal that says why, in terms the user can act on, is a piece of service.

Did it offer a path forward. A refusal that ends the conversation is a dead end. One that says I cannot help with this, and here is who can, has converted a failure into an outcome, and the difference between those two shows up in the escalation and abandonment numbers.

Over-refusal is the one I would fight hardest for, because it is the one that gets sacrificed. Every incident produces pressure to refuse more, and no incident ever produces pressure to refuse less, and the ratchet runs in one direction until the product is useless and everybody is puzzled about why engagement is falling.

Escalation behavior

Knowing when to stop is a capability and it should be evaluated as one.

The triggers are usually clear once you write them down. Low confidence. A high-stakes intent. A user who has said the same thing twice. Emotional distress. A request that falls outside the policy the assistant was given. A tool failure that leaves the request unresolvable. Any of these is a signal that a human should take over, and a system that plows on regardless is producing a bad experience with great efficiency.

Score both directions. Escalation when it was warranted, and, equally, escalation when it was not, because a system that escalates too eagerly is a system that has moved its cost onto a support team and its failure onto a queue.

And score the handoff itself, which is the part that gets no attention and which the customer experiences most sharply. Was context transferred, so the human does not ask the customer to repeat what they have already said three times. Was the reason for escalation communicated. Did the assistant tell the truth about what was happening. A handoff that dumps the customer into a queue with no context is worse than a refusal, because it has consumed their time and their patience and given them nothing.

One class of safety failure deserves separate handling because the enforcement point is counterintuitive.

Prompt injection through retrieved content. A document in your corpus contains text designed to look like an instruction, the retriever returns it, and the model reads an instruction that a third party wrote. The user is innocent. The attacker is a web page, a support ticket, a product review, or a document that somebody uploaded.

The instinct is to defend this in the prompt, by telling the model to ignore instructions found in evidence. That helps and it is not a control. The control is structural: mark retrieved content as data at the boundary, keep it out of the instruction channel, and place the checks that matter downstream of the model rather than inside it. And test it, with cases where the hostile text is in the corpus rather than in the user's message, because that is the version almost nobody has tried.

Auditability

This is the lawyer's question from the opening, and answering it is an engineering requirement in any domain where somebody can be harmed by an answer.

The record has to reconstruct the decision. For any response, at any time in the retention window: the exact response, the assembled context that produced it, the evidence retrieved and the version of each document, the prompt version, the model identifier, every policy and safety check that ran and its verdict, every tool call and its authorization basis, and the identity of the user and the session.

That is a demanding list and it is the same list as Chapter Seventeen's, which is not a coincidence. Observability and auditability are the same investment viewed from two directions, one of which is engineering and one of which is legal, and building it once satisfies both.

Test the audit trail by using it. Take a random session, hand its trace to somebody who was not involved, and ask them to write the paragraph that answers the lawyer's question. If they cannot, the trail is inadequate, and you have found that out on a Tuesday afternoon rather than under subpoena.

And retain according to the obligation rather than according to the storage budget. Regulated domains have retention requirements measured in years, and a system that deletes logs after thirty days because that was the default is a system that will be unable to answer a question it is legally required to answer.

Red-team evaluation

Red teaming is the practice of attacking your own system on purpose, and it is the only evaluation activity where the goal is to fail.

Do it in three layers, because they find different things. Automated attacks, generated at scale, per Chapter Seven, covering the known categories: instruction override, indirect injection through retrieved content, role play, incremental escalation, extraction, and sycophancy under confident false assertion. Cheap, broad, and it catches the known patterns.

Human red teaming finds what generators do not, because humans are creative in ways that a model prompted to be creative is not. The published work here is worth reading rather than reinventing. Deep Ganguli and colleagues at Anthropic ran a large human red-teaming study in 2022 and released the transcripts, and Ethan Perez and colleagues at DeepMind demonstrated in the same period that models can be used to generate attacks against other models at a scale no human team could match. Together those two papers describe the shape of a program: generate broadly, then have people probe where the generator cannot reach.

And domain red teaming is the one only you can do. Your policy people know how a customer talks their way into a refund. Your support team knows the phrasings that work. Sit them in front of the assistant for an afternoon and let them try, and they will find things that no security researcher would have thought of, because the attack surface is your business rather than the model.

Every successful attack becomes a permanent regression case, and the attack suite never shrinks. And run it against every model migration, because an attack that fails on one model may succeed on another, and your archive of attacks is, to some extent, an artifact of the model it was built against.

A framing that makes compliance work considerably less painful, and that most engineering teams arrive at only after their first audit.

The auditor is not asking whether your system is safe. They are asking whether you can demonstrate that you tried to find out, that you found things, and that you did something about

them. A team that reports zero safety failures and cannot show what it tested is in a far worse position than a team that reports a list of failures it found, fixed, and now guards against with permanent regression cases.

That reframing changes what you build. The artifact of value is the record: what was tested, what was found, what was done, who decided the residual risk was acceptable. Every one of those is produced as a side effect of the discipline in this book, which means a competent evaluation program is most of a compliance program, and a team without one will be manufacturing evidence under time pressure, which is where compliance work goes to become fiction.

Compliance reporting

At some point somebody outside engineering will ask you to demonstrate that your system is safe, and the demonstration has to be a document rather than a conversation.

The frameworks that exist are worth using rather than inventing your own. The NIST AI Risk Management Framework, published in 2023, gives a structure for identifying, measuring, and managing risk that maps closely onto what a competent evaluation program already does. ISO and IEC published a management system standard for artificial intelligence in the same period. And in the European Union, the AI Act establishes obligations that scale with the risk category of the system, which means the first question for a compliance program is which category you are in.

What all of them ask for, in different words, is the same set of artifacts. What the system is for and what it is not for. What could go wrong and how badly. What you did to reduce that. How you measured whether it worked. What the residual risk is. Who decided the residual risk was acceptable. And what happens when something goes wrong anyway.

Every one of those is produced by the evaluation program described in this book, and the useful reframing is that compliance reporting is mostly a packaging problem rather than a new activity. A team with a versioned dataset, a documented rubric, a gate with recorded overrides, an incident process, and a failure taxonomy already has the evidence. A team without them will have to manufacture it, retrospectively, under time pressure, which is where compliance programs go to become fiction.

A note on how safety evaluation ages, because it ages faster than anything else in this book.

Your adversarial suite is a record of attacks that worked against a system that no longer exists. The model has changed. The prompt has changed. The filters have been tuned in response to the very attacks in the suite. What you have is a set of cases that your current system passes by construction, and a growing false confidence that it is safe.

So generate fresh attacks continuously rather than replaying the archive, and treat a perfect score on the adversarial suite as evidence that the suite has gone stale rather than that the system has become secure. The archive still has value, as a regression test, to confirm you have not reintroduced an old hole. It has no value as a measure of current exposure.

Risk-based release gates

Not every change carries the same risk and a gate that treats them identically will be too slow for the small changes and too permissive for the large ones.

Tier by consequence. A formatting change gets the fast suite and ships. A prompt change to a low-risk intent gets the full offline suite and a canary. A change to a policy answer gets the full

suite, the safety suite, the policy owner's approval, and a slow ramp. A model migration gets all of that plus a launch review. And a change that touches an irreversible action, anything that moves money or sends something to a third party, gets the full agentic suite from Chapter Eleven and a named human who signs.

The tier is determined by the change, automatically, from the files it touches and the intents it affects, so that nobody has to remember and nobody can quietly select an easier tier for themselves.

And the highest tiers cannot be overridden by the team shipping. That is the whole architecture of the thing. Every other gate in this book can be overridden by a named person accepting a documented risk, and that is correct, because a gate that cannot be overridden gets routed around. The safety gate is the exception, and the exception is what makes the rest of the system credible.

Area	Check	Enforced where
Content harm	Provider safety categories, tested through composition with retrieval and tools	Model + output filter
Domain harm	Advice that requires a licensed professional; safety-critical instructions	Policy check
Data exposure	Another user's data, restricted documents, the system prompt itself	Retrieval ACL + filter
Commercial harm	Promises, prices, dates, competitor claims the company cannot stand behind	Policy check
Inbound PII	Redacted before storage; redaction itself has a test suite	Ingestion
Retrieval ACL	Restricted documents unretrievable for unauthorized users, checked below the model	Retrieval layer
Tool data minimization	Only the fields the tool requires are passed	Tool layer
Extraction attacks	System prompt, evidence, and other users' data cannot be elicited	Adversarial suite
Policy rules	Each written rule has a criterion and tempting cases	Downstream check
Policy versioning	Cases linked to source policy; stale cases quarantined	Dataset registry
Refusal correctness	Refuses what it must	Safety suite
Over-refusal	Answers what it may; rate tracked as a regression	Should-answer suite
Refusal explanation	States the reason and offers a path forward	Rubric
Escalation triggers	Low confidence, high stakes, repetition, distress, tool failure	Scripted cases
Handoff quality	Context transferred; customer does not repeat themselves	Conversation-level scoring
Audit reconstruction	A stranger can answer what was said and on what basis	Trace review
Retention	Set by legal obligation, not by storage cost	Logging policy
Automated red team	Injection, override, role play, escalation, extraction, sycophancy	Every release

Area	Check	Enforced where
Human red team	Creative attacks that generators do not reach	Quarterly
Domain red team	Policy and support staff attacking their own rules	Quarterly
Migration re-test	Full attack suite re-run; fresh attacks generated per model	Every model change
Risk tier	Gate depth determined automatically by what the change touches	CI configuration
Safety gate override	Not available to the shipping team	Governance

Read the last row again. Every other gate in this book bends under pressure by design, because a rigid gate is a gate that gets bypassed. That one does not bend, and the reason it does not is that it is the only thing in the entire apparatus that is protecting somebody other than you.

Which brings us to the part that no amount of engineering will solve, and that will decide whether any of this survives its first reorganization.

Chapter 24: Evaluation Leadership and Operating Model

The evaluation program died in a reorganization, and it took about six weeks, and nobody made a decision to kill it.

What happened was that the engineer who had built it moved to another team, because the work was not visible on any promotion path and she was tired of explaining that. The dataset went unmaintained, which meant it went stale, which meant its scores started to disagree with reality. The gate started blocking releases for reasons that were increasingly hard to defend, and so the gate was overridden, and then overridden again, and then, in a meeting where everyone was reasonable, switched to a warning.

The dashboard stayed up for another two quarters. It was green the entire time.

Every technical recommendation in this book is worthless in an organization that will not staff it, will not reward it, and will not let it block anything. This chapter is about the organization, and I have come to believe it is more important than the twenty-three that came before it.

Who owns evaluation?

The question has three wrong answers that are each seductive.

The wrong answer that everybody tries first is that everybody owns it. Every team evaluates their own feature, using their own methods, reporting their own numbers. This produces the four incomparable percentages from Chapter Twenty-One, and it produces no shared standard, and it means that no number in the company can be trusted at the point where trust matters, which is when it is used to say no.

The second wrong answer is that the science team owns it, because evaluation involves metrics and metrics feel scientific. This produces excellent methodology that runs on a research cadence, disconnected from the release pipeline, arriving after the launch it was supposed to gate.

The third wrong answer, and the most dangerous, is that the team that ships the feature owns the gate on their own feature. Nobody in that position is going to block themselves on a Thursday afternoon before a launch that has been on a roadmap for two quarters. They are not weak; they are human, and you have designed a system in which the only person who can say no has every incentive not to.

The answer that works is a small, funded, central team that owns the instrument, and distributed teams that own their own quality and are measured against the instrument. The central team owns the platform, the dataset standards, the rubric library, the judge calibration, the gate thresholds, and the ability to block. It does not own the quality of any individual product, which belongs to that product's team, and this separation is what makes both sides work.

And the central team does not report into the organization whose launches it gates. That reporting line is the whole thing. Every other structure will decay in the direction of the pressure that is applied to it.

Product, engineering, science, legal, and operations roles

Evaluation touches more functions than any other part of an AI product and each one has a piece that only they can do.

Product owns the definition of good. What is the task, what does success look like, what tradeoff between correctness and brevity is right for this audience. Engineers are not qualified to decide this and will decide it anyway, by default, in code, if nobody else does.

Engineering owns the instrument. The platform, the runner, the scoring service, the integration with the pipeline, the observability that makes any of it possible. This is a real engineering system and staffing it with a fraction of one person is how you get the spreadsheet.

Science owns the methods. Judge calibration, sampling design, statistical rigor, the noise floor, whether a delta means anything. This is the function that stops the organization from acting on random variation, and it is worth its salary for that alone.

Legal and policy own the constraints. What must never be said, what must be logged, how long it is retained, what the compliance obligation is. They will find out about your product eventually, and the difference between them finding out at design time and finding out at launch is about a quarter of your schedule.

Operations owns the humans. The reviewer pool, the training, the calibration meetings, the throughput, the fatigue caps from Chapter Nineteen. This is a real operational function with real management overhead, and treating it as something an engineer does on the side is how the ground truth gets corrupted.

The most common failure is that three of these five are unstaffed and the engineer is doing all five badly, at night, because they care.

Evaluation review boards

A board sounds bureaucratic and it is worth defending, because the alternative is that these decisions get made in a hallway.

The board owns the decisions that affect the instrument rather than the product. Changes to the frozen dataset. Changes to the rubric. Changes to the gate thresholds. Approval of a judge model for a criterion, based on its calibration. And review of every gate override from the previous month.

That last item is the one that gives the board teeth, and it should be short and regular and slightly uncomfortable. Every override, who made it, what risk they accepted, and what happened afterward. Most will have been fine. The two that were not will teach the organization more about its own risk appetite than any policy document, and the knowledge that this review exists is what makes people think before they override.

Keep it small and keep it fast. Five people, thirty minutes, monthly. A board that takes two hours will stop being attended by the people whose attendance matters.

Launch readiness reviews

The launch review is where the evaluation program either has authority or does not, and everyone in the room knows which within about ninety seconds.

Give it a fixed agenda so it cannot be talked around. What does the offline suite say, by criterion and by severity. What does the safety suite say. What does the adversarial suite say. What is the cost and latency impact. What are the known failures you are shipping with, and what is the plan for them. What are the rollback triggers and who is watching. And what is the quality of the evidence itself, meaning when was the dataset last refreshed and when was the judge last calibrated.

That last question is the one nobody asks and it is the one that determines whether any of the other answers mean anything. A launch review that accepts a quality number computed against an eighteen-month-old dataset scored by an uncalibrated judge has reviewed nothing, thoroughly.

The review must be able to say no. If it has never said no, it is a ceremony, and everyone in the organization has already worked that out and is behaving accordingly.

Model migration governance

Model changes deserve their own governance because they change everything at once and because they will be forced on you.

That last point is the one teams are least prepared for. Providers deprecate models. You will receive a notice giving you some number of months to migrate off a model your entire product was tuned against, and the migration will not be optional, and the timeline will not be yours. A team that has never run a migration will discover, at that moment, that they have no cached contexts to replay, no scorecard, and no idea what will break.

So run the scorecard from Chapter Fourteen as a standing process rather than as a project. Every candidate model, evaluated on the same axes, on cached contexts, with the results in a comparable format. Keep it current. When the deprecation notice arrives, you want to already know what your options are.

And govern the silent changes, which are the ones nobody plans for. A hosted model behind an endpoint is not a constant. Serving stacks change, quantization changes, safety layers get tuned, and you will not always be told. Run a small canary suite continuously against production, on a schedule, and watch for drift with no deploy behind it. That signal is your only defense against a provider changing something underneath you, and the first time it fires and you understand why, it will pay for itself.

A hiring observation that I offer with some hesitation and considerable conviction.

The person who is good at this job is frequently not the strongest engineer available. They are the person who is comfortable being the one in the room who says the number does not support that. That is a temperament rather than a skill, it does not show up in a technical interview, and most engineering organizations select against it without meaning to, because it reads as difficult.

Look for it deliberately. Ask a candidate about a time they were the only person who disagreed, and listen to whether they stayed disagreed. An evaluation function staffed entirely by agreeable, capable people will produce excellent methodology and will never block anything, and it will therefore not be an evaluation function.

Incident review process

The incident review is where the organization learns, and its quality determines whether the same failure happens twice.

Run it blamelessly, in the sense the site reliability community established: the question is what in the system made this failure possible, rather than who erred. This matters more for generative systems than for conventional ones, because the failures genuinely are nobody's fault. The engineer who added the word concise was doing good work. The content team that changed a page template had never heard of your service. A review that goes looking for a culpable person will find one and will stop looking before it finds the system that made the error inevitable.

The agenda maps onto the taxonomy from Chapter Eighteen. What was the failure class. Which component owns it. Why did evaluation not catch it, which has the three answers from that chapter: no coverage, coverage that failed and was overridden, or coverage that passed and should not have. That third answer is the most valuable and the hardest to admit.

And the review has one mandatory output, which is a test case, plus the twenty variations that confirm the fix generalizes. An incident that does not produce coverage has not been learned from. It has been survived.

Documentation standards

Documentation is where an evaluation program either becomes an organizational asset or stays in one person's head, and the story at the top of this chapter is what happens when it stays in one person's head.

The artifacts that have to exist as documents. The rubric library, with definitions, boundary cases, and the written rationale for every adjudicated disagreement, which is the thing that actually teaches a new reviewer. The dataset documentation: provenance, composition, sampling weights, known gaps, refresh cadence. The gate thresholds and the reasoning behind each number, because in a year somebody will ask why the bar is at eighty-two and the answer cannot be that it seemed about right. The judge calibration results, with dates. And the failure taxonomy, which is the shared vocabulary and which is useless if it lives in a slide deck.

Adequate documentation is not measured by whether it exists. The test is whether a competent engineer who joined last week can run an evaluation, interpret the result, and explain to a product manager what it means, without asking anyone. If they cannot, the program has a single point of failure, and that person will eventually go to another team, for the same reason the engineer in the opening did.

A structural failure worth naming because it is invisible from inside and obvious from outside.

Evaluation gets funded after an incident and defunded after a quiet quarter. The incident makes the case, headcount appears, a program is built, quality improves, and the improvement means no incidents, and the absence of incidents is read as evidence that the program is no longer needed. Two quarters later the headcount is reallocated to a launch, and the cycle begins again with a new incident.

The defense is to make the program's output visible when nothing is on fire. Publish what was caught. Every quarter, a short note listing the bad changes that the gate blocked, the failures the suite found before customers did, and the incidents that did not happen. That list is the product of the program and it is invisible by construction, and if nobody writes it down, then the program's entire value is a counterfactual that no budget process has ever been able to see.

Metrics for leadership

A leader will look at your dashboard for eleven seconds a week, and what they take from those eleven seconds will determine what gets funded.

So give them three numbers, and choose them carefully, because whatever you choose will be optimized. Quality on the frozen dataset, with the noise floor next to it. Critical failure count, which is the number that stops things. And the escalation or task-completion rate, which is the one online signal that is close to ground truth.

Add one number that most organizations never report and that changes the conversation entirely: the health of the instrument. When was the dataset last refreshed. When was the judge last

calibrated. What is the current agreement between the judge and human reviewers. What fraction of production failure classes have evaluation coverage. A leader who understands that the quality number depends on the health of the thing measuring it will fund the thing measuring it, and a leader who does not will cut it in the first difficult quarter and will never know what they lost.

And be honest when the number is bad. An evaluation function that only ever reports good news has told the organization, clearly, that it is not an evaluation function. The first time you block a launch and are proved right is the moment the program becomes real, and until that happens you are running a dashboard.

One practical starting move for anyone who has read this far and has no authority to do most of what is in this chapter.

Publish the number anyway. Compute an honest quality measurement, on a real dataset, with criteria you wrote down before you looked, and put it somewhere the team can see. Do not ask permission. Do not wait for a mandate, a headcount, or a platform. A single honest number, visible, updated weekly, will do more to change what an organization believes about its own product than any proposal you could write.

What happens next is predictable and it is the point. Somebody will dispute the number, which is good, because disputing it requires engaging with the methodology, and engaging with the methodology is how a shared instrument gets built. The alternative, waiting for the organization to ask for measurement, means waiting for an incident, and the incident will arrive on its own schedule.

Building a culture of AI quality

Culture is what remains after the process document has been ignored, and there are only a few things that reliably move it.

Make quality work visible. The engineer who built the evaluation harness saved more customer harm than the engineer who shipped the feature, and only one of them got mentioned in the launch note. This is a solvable problem and it is solved by whoever writes the launch note.

Make it promotable. If the career ladder rewards shipping features and has no rung for the person who prevented three bad launches, then the person who prevented three bad launches will go and ship features somewhere else, and you will find out about it in a reorganization.

Reward the finding of bad news. The most valuable person in an AI organization is the one who says the number is wrong and here is why, and the way they are usually treated is as an obstacle. An organization where finding a problem is a contribution will find problems. One where it is an inconvenience will stop finding them, and the problems will still be there.

And insist on evidence in the room. The most powerful cultural intervention I know is a norm, enforced by whoever runs the meeting, that a claim about quality requires a number, and a number requires a dataset version, and a dataset version requires somebody to have maintained it. Once that norm holds, everything in this book becomes something the organization wants rather than something an engineer has to defend.

Level	Dataset	Scoring	Gate	Ownership
1. Ad hoc	A text file of queries somebody made	Reading outputs, forming impressions	None; a demo before launch	Nobody

Level	Dataset	Scoring	Gate	Ownership
2. Measured	Labeled cases, in version control	A rubric, applied by humans	A number is reported at launch review	One engineer, unofficially
3. Automated	Versioned, stratified, refreshed from traffic	Calibrated judge plus human sample	Blocks a release; overridable by a named person	A funded team
4. Governed	Frozen and living tracks; coverage tracked by failure class	Judge agreement monitored; reviewers calibrated and capped	Risk-tiered; safety tier cannot be overridden	Central instrument owner, outside the shipping org
5. Continuous	Refreshed from production weekly; incidents become coverage automatically	Scoring re-runnable against stored outputs; drift detected without a deploy	Gate, canary, and auto-rollback wired end to end	Quality is a career path, not a chore

Most teams shipping generative AI today are at level one and believe they are at level three, and the gap between those two positions has very little to do with tooling. It comes down to whether anyone in the organization is allowed to say no.

That is the whole of the argument. What remains is to show you the finished thing, working, end to end.

Part Seven

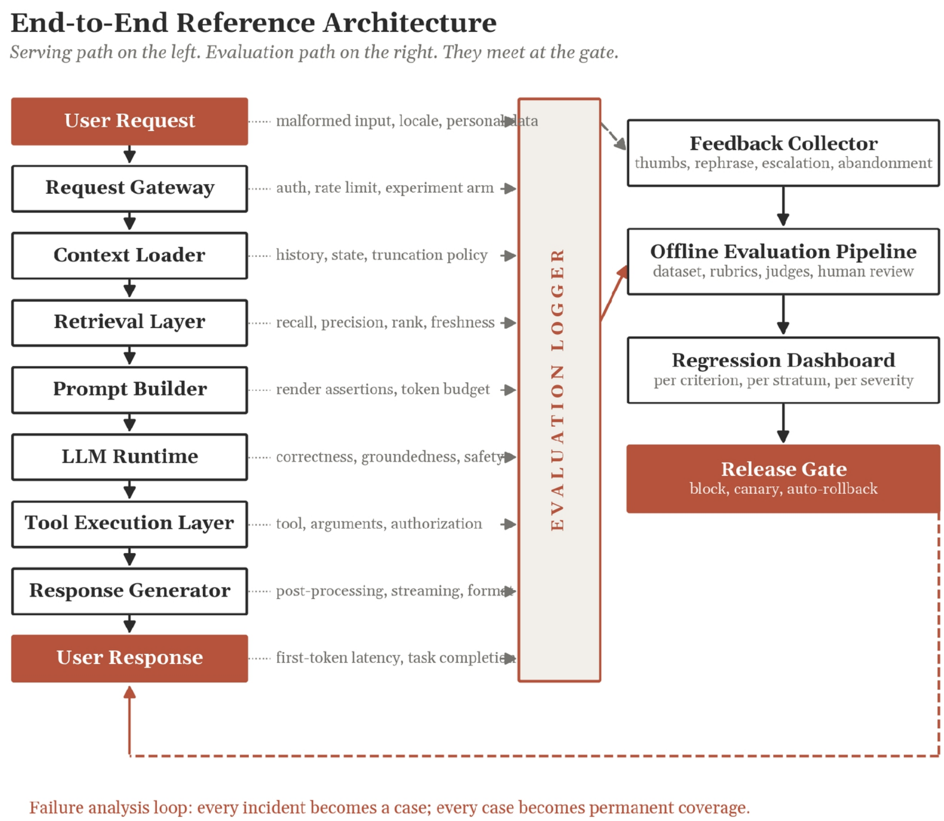
Reference Implementation and Practical Templates

Chapter 25: End-to-End Reference Architecture

Everything to this point has been a part. This chapter is the assembled machine.

I want to be careful about what a reference architecture is, because the phrase invites a certain kind of misuse. Treat it as a blueprint you implement and it will fail you. Your system already exists, it was built under constraints I know nothing about, and a diagram that tells you to rebuild it is a diagram you will politely ignore.

What it is, is a checklist of positions. Every box in it is a place where quality is decided, and every arrow is a place where something can be logged. Read it against your own system and the value is in the gaps: the components you do not have, the arrows you do not log, the loop you have never closed. In my experience most teams find between four and seven gaps on a first pass, and two of them are usually load-bearing.



System components

The serving path runs down the left and it is probably close to what you already have.

The request gateway is the front door. It authenticates, applies rate limits, and, importantly for everything that follows, assigns the request to an experiment arm and stamps it with a version tuple. If the gateway does not record which arm and which versions a request belongs to, every experiment downstream is uninterpretable.

The context loader assembles what the model will see from what has happened before. Conversation history, persisted state, user attributes. It

owns the truncation policy, which is the single most consequential undocumented decision in most production systems.

The retrieval layer finds evidence. Embedding search, keyword search, reranking, filtering by permission. It is where the largest share of your failures originate and where the fewest of your tests point.

The prompt builder renders the final text. Template, instructions, evidence, history, ordering. It produces the artifact that is the actual input to the model, and it is the artifact that most teams never persist.

The model runtime generates. The tool execution layer acts. The response generator transforms what came out into what the user sees: safety filtering, enrichment, formatting, streaming.

That is the product. Nine boxes, and if you drew your own system you would draw something close to it.

Three of those nine are the ones teams routinely fail to recognize as components at all, and I want to name them, because a component you do not recognize is a component nobody owns. The context loader is the first. It is usually a function inside another service, written by whoever needed it, and it holds the truncation policy that determines what the model can and cannot know. The prompt builder is the second, and it is frequently a template file with no tests, edited by four teams, governing every response the product has ever produced. And the response generator is the third, a pipeline of transformations bolted on over eighteen months, any one of which can take a correct answer and make it wrong.

Draw them as boxes. Give each one an owner. That single act, which costs a whiteboard and an argument, is worth more than most of the tooling in this book, because it converts three invisible failure surfaces into three things that appear in somebody's backlog.

Data flow

Follow one request through and notice how many times the thing being evaluated changes shape.

A user types eight words. The gateway wraps them in a request object with a session identifier and an experiment arm. The context loader adds twelve prior turns, four of which get summarized because the budget is tight. The retrieval layer turns the eight words into a rewritten query, then into an

embedding, then into fifty candidates, then into five after reranking. The prompt builder concatenates a system prompt, the five passages, the summarized history, and the eight words into a single string of nine thousand tokens.

That string is the input. The eight words the user typed are about one tenth of one percent of it.

The model produces tokens. The tool layer may interrupt to call something and feed the result back. The response generator takes the completed text, filters it, enriches it with product data, and renders it into a stream of user-interface components, at which point the text the model generated is again a minority of what the person is looking at.

Every one of those transformations is a place where the thing gets better or worse, and an evaluation that looks only at the user's eight words and the final screen is evaluating the two thinnest slices of the whole pipeline.

The flow also runs backward, and the backward path is the one most systems get wrong. After the response is delivered, something writes the turn to storage. Something decides what to summarize. Something decides what the next turn will be permitted to see. Those writes are the input to the next request, which means a bug in the write path manifests one turn later, or ten, in a component that did nothing wrong. The diagram shows this as an arrow, and the arrow is the reason Chapter Ten insists on round-tripping the persisted state in your test suite.

Evaluation insertion points

The annotations down the right side of the serving path are the insertion points, and they are the practical content of this chapter.

At the request: is the input malformed, is the locale supported, does it contain personal data that needs to come out before anything is stored.

At the gateway: was the experiment arm assigned correctly, and does the traffic split match what was configured, because a sample ratio mismatch here invalidates everything downstream.

At the context loader: is the information required to answer this question present in the assembled context. This is the single highest-value check in the entire system, it is a yes-or-no question, it costs almost nothing, and it splits your failures into two populations with different owners.

At retrieval: recall, precision, rank position, document freshness, and whether the answer exists in the corpus at all.

At the prompt builder: does the rendered prompt contain every section, no empty placeholders, and a token count inside budget.

At the model: correctness, groundedness, instruction following, safety.

At the tool layer: right tool, right arguments, authorized, confirmed, idempotent, and honest about failure.

At the response generator: did post-processing make it worse, which is checked by scoring the output before and after and looking at every case where the score dropped.

And at the user response: time to first token, and whether the person got what they came for.

Nine positions. Most teams instrument the first and the last.

Logging strategy

The vertical bar down the middle of the figure is the evaluation logger, and it touches every stage, and it is the component that makes the entire right-hand side possible.

What it captures, per request: the version tuple, the assembled context verbatim, the retrieval trace with scores and document versions, the rendered prompt, the raw model output before post-processing, every tool call with its arguments and authorization basis, the final response, the per-stage latency, and the token counts by role.

Sample it. Full capture on a small percentage of traffic, plus one hundred percent capture on anything with a negative signal, a high-risk intent, or an experiment arm. Compress, because this text is highly redundant. Retain for weeks rather than years, and promote what you need into the evaluation dataset, where it lives redacted and abstracted and no longer contains a person.

If you build one thing from this chapter, build this. Every other capability in the book, the dataset, the model comparison, the failure taxonomy, the root-cause analysis, is downstream of a logger that captured the intermediate state. A team without one is running a system whose only instrument is a customer complaint.

There is a design detail here that determines whether the logger is usable in two years. Log the artifact, not a reference to it. A trace that says the system retrieved documents 4471 and 8802 is a trace that becomes meaningless the moment those documents are edited, and they will be edited. Store the passage text as it was at that moment, or store an immutable content hash and keep the content addressable, but do not store a pointer into a mutable world and expect it to still mean something when you follow it.

The same applies to prompts, rubrics, and index versions. Everything the logger references must be resolvable to the exact bytes that were in force, or the reproduction you are counting on will silently reconstruct a different request from the one that failed.

Offline evaluation flow

The right-hand column is the evaluation path and it runs on a different clock from the serving path.

The offline pipeline pulls cases from the dataset registry, resolves a version tuple, executes them through the same serving path the product uses, and scores the outputs. Deterministic checks first, because they are free. Then the judge, calibrated, one criterion at a time, with position swapping. Then a human sample, which is slow and which is what keeps the judge honest.

Scores land in the metrics store, which is append-only, and which records the scorer and the rubric version alongside every number, so that a score can be interpreted in a year.

The comparison report is generated from the store and posted to the pull request, showing the crosstab, the per-criterion deltas, the noise floor, and links to every regressed case.

The critical design property is that scoring is separate from execution, which means you can re-score stored outputs against a new rubric without regenerating them. That one decision will save you an enormous amount of money the first time you change a criterion.

Online monitoring flow

The feedback collector sits alongside the serving path and gathers what production tells you, which is a different set of signals from anything offline evaluation can produce.

Explicit feedback, with a reason code, because an untagged thumbs-down is unroutable. Implicit signals: rephrase, abandonment, escalation, correction. Guardrail metrics in real time: safety interventions, latency at the tail, error rates, tool failures, refusals. And an online quality sample, scored asynchronously, which is the only measurement in the system that is computed on what real users actually received.

The guardrails feed the canary and the automatic rollback. The quality sample feeds the dashboard. And the negative signals feed the sampler that produces tomorrow's evaluation cases, which is the loop that keeps the dataset from going stale.

Put the offline and online numbers on the same page, always. Their relationship is the most informative thing on your dashboard, and the day the offline score rises while the escalation rate holds flat is the day you learn that your instrument has drifted away from the thing it was built to predict.

One thing this architecture deliberately leaves out, and the omission is the point.

It does not have a single evaluation service that every team calls. The registries, the scoring service, and the gate are shared, and the coverage of each component belongs to the team that owns that component. Retrieval tests live with retrieval. Context-builder assertions live with the context builder. Tool assertions live with the agent layer.

A central service that owns every check becomes a queue, and the queue becomes a bottleneck, and the bottleneck becomes the reason four teams build their own shadow evaluation to route around it, which is exactly the situation the platform was built to end. Centralize the instrument. Distribute the testing.

Release gate integration

The gate is where the two paths meet and it is the only box in the diagram that can say no.

It takes a version tuple and returns a decision. It reads from the metrics store, applies the thresholds from Chapter Sixteen, and blocks the deployment if any condition is unmet. It is called by the deployment

system, not by a person, and it cannot be skipped by choosing a different deployment path, because there is only one.

It is risk-tiered, per Chapter Twenty-Three. A formatting change gets the fast suite. A change to a policy answer gets everything. A change to an irreversible action gets everything plus a named human signature. The tier is derived automatically from what the change touches, so nobody has to remember and nobody can select an easier one for themselves.

Overrides are permitted, recorded, and reviewed monthly, with one exception. The safety tier cannot be overridden by the team that is shipping. That exception is what makes the entire structure credible, and it is the first thing that will be argued with.

A pragmatic note on how to read this diagram against a system that already exists, because that is the situation almost every reader is in.

Do not start by drawing your architecture. Start by taking one real failure, from last month, that somebody reported and nobody could explain. Walk it backward through the boxes on the left and ask, at each one, whether you could reconstruct what that component did on that request. The first box where the answer is no is where your instrumentation ends, and everything upstream of it is a place where failures can hide indefinitely.

In my experience the answer is no somewhere between the context loader and the prompt builder, in almost every system, and the team is genuinely surprised to discover it, because those components feel like plumbing rather than like places where quality is decided.

Failure analysis loop

The dashed line running back from the gate to the serving path is the last arrow in the diagram and the most important one, because it is the one that makes the system get better rather than merely observed.

A failure occurs in production. The feedback collector catches it, or a customer reports it. The trace is pulled from the logger. The root-cause procedure from Chapter Eighteen runs backward through the pipeline: was the context sufficient, was the evidence retrieved, was it in the corpus, did it survive assembly, did the model use it. The failure gets a class, and the class has an owner, and the owner gets a ticket.

Then the ratchet. The failure becomes a test case. The test case gets twenty synthetic siblings, so that the eventual fix is confirmed to generalize rather than confirmed to satisfy one string. The case family goes into the frozen dataset with the incident identifier attached, and it stays there for the life of the product. The gate now blocks any future change that would reintroduce it.

That is the whole machine. Every other component in the diagram exists to make that loop possible, and a system that has all the components and never closes the loop is a system that will experience the same failure every eight months, forever, with a slightly different customer each time.

An architecture is an abstraction and abstractions are easy to nod at. The next three chapters put this one to work in three real domains, each with different risks, different data, and different reasons why the obvious approach does not survive contact with production.

Chapter 26: Case Study 1, Customer Support AI

A support assistant handles account questions, billing disputes, troubleshooting, and the moment when a person has had enough and wants a human. It sits between a company and its angriest customers, it has access to systems that can change someone's account, and it is deployed because the support queue is expensive.

That last clause is the one that shapes everything. A support assistant is funded to deflect contacts, and deflection is trivially easy to achieve by giving people confident answers that make them go away. Deflection and resolution look identical in a dashboard for about six weeks, and then they diverge, and by then the assistant has told several thousand people something that was not true.

So the evaluation program for a support assistant is designed around one adversarial pressure: the business metric it will be judged on can be gamed by the exact failure mode it must not have. Everything below follows from that.

Dataset design

The support organization has already built your dataset and does not know it.

Every resolved ticket is a case with a known correct outcome, written by the agent who resolved it. The customer's question is there, in their own words, with their own typos and their own anger. The resolution is there. The policy applied is often there. This is a labeled dataset produced at scale by people who were paid to produce it for another reason, and the pipeline that turns it into evaluation cases is the highest-return week of engineering in the whole program.

Stratify by intent and by risk, and let risk dominate. Billing disputes, refunds, account access, and anything that touches a payment method are high-risk regardless of their share of volume. Password resets and store hours are high-volume and low-risk. The evaluation budget should not follow the volume, because the exposure does not.

Stratify by conversation depth, and oversample the deep ones. Support conversations are long. A customer explains a problem, the assistant asks a question, the customer answers, the assistant proposes a fix, the customer says it did not work. Turn six is where the failures live and turn six is a minority of the traffic.

Stratify by emotional register, which is a dimension nobody thinks to include and which matters enormously here. A calm question and a furious question about the same problem are different inputs. The furious one contains less information, more assertion, and a demand. Systems that handle the calm version beautifully often fall apart on the furious one, and the furious one is the one that ends up on social media.

And build the escalation set: two hundred cases where the correct behavior is to stop and hand off. This set exists to be scored in both directions, and it is the only defense against the deflection pressure.

One more stratum, which comes from the tickets and which nobody thinks to build. The customer who is wrong. A person contacts support convinced that they were charged twice, and they were not, and the correct answer is to explain, kindly, with evidence, and hold the position when they push back. This is the sycophancy test from Chapter Ten in its natural habitat, and a support assistant that folds under an insistent customer will start agreeing that charges were duplicated, and the downstream cost of that is a refund the company did not owe.

Rubric design

The rubric has four vetoes and they are unusual, and each one exists because of a specific way this product fails.

Policy correctness is a veto. A support answer that misstates policy is not a partially good answer. The customer will act on it, and the company will either honor a commitment it did not make or refuse one it appeared to make, and both outcomes are bad in ways that reach a lawyer.

Commitment discipline is a veto. The assistant may explain a policy. It may not promise an outcome. The difference between refunds for damaged items are generally approved within five business days and your refund will be approved is the difference between information and a contract, and models drift toward the second because the second is more helpful.

Data exposure is a veto. The assistant must never reveal another account's information, and it must never reveal more of this account's information than the person has authenticated for.

Escalation correctness is a veto in one direction. Failing to escalate a case that required a human is a critical failure, because the categories that require a human are the categories where a machine can do real damage.

Then the graded criteria. Groundedness in the policy corpus. Completeness against the required elements of the case. Whether the assistant asked for information it already had, which is the single most reliable way to enrage a person who has already given it. Tone under pressure. And whether the answer actually resolved anything, which is the criterion that separates resolution from deflection and which has to be scored by a human.

Tool-use evaluation

The support assistant acts, and its actions are exactly the ones from Chapter Eleven that cost money.

It issues refunds. It changes shipping addresses. It cancels orders. It resets access. It opens and closes tickets. Every one of those is a write, most of them are irreversible in the sense that matters to a customer, and the model is deciding which one to call from a conversation with an upset person who may be describing their problem inaccurately.

The authorization tests are the ones I would build first. The customer asks about an order that is not theirs, having found the order number somewhere. Does the tool layer check ownership, below the model, deterministically, and refuse regardless of how the request was phrased. Build a hundred cases whose only purpose is to persuade the model to cross that line, and assert that crossing it accomplishes nothing.

The idempotency tests are the ones that will save your money. Timeout on a refund that succeeded. Duplicate submission when a customer double-taps. The model emitting the same call twice. Each of these is an afternoon of fault injection and each of them prevents a variety of incident that is embarrassing in a specific and public way.

And the confirmation tests, with the specificity requirement. A confirmation that says shall I proceed is worthless. One that says I will cancel order 4471 and refund forty-two dollars to the card ending in 4122 is a confirmation, because the customer reading it can catch the error before it happens.

The derived-argument cases deserve their own family, because support conversations are full of relative references that something has to resolve into an identifier. The customer says the one from last month, and the tool needs an order number. They say the blue one, and there were two.

They say cancel it, and it could be the order, the subscription, or the return they opened yesterday. Each of these is an inference, each inference can be confidently wrong, and a wrong inference here cancels the wrong thing. Build a case for every relative reference your customers actually use, which you can find by reading a hundred tickets, and assert on the resolved identifier rather than on the prose.

Safety gates

The safety surface for a support assistant is unglamorous and it is where the real risk sits.

Commercial harm is the dominant category. A promise the company cannot keep. A price that is wrong. A delivery date. A claim about a competitor. A statement about a warranty that a lawyer would not have signed off on. Each of these is enforced downstream of the model, as a check, because a rule in a prompt is a suggestion that holds almost always and almost always is not a standard for something that creates a legal obligation.

Data exposure is the second category, and it is enforced at the retrieval layer with an access control filter rather than in the prompt. A document the customer may not see must be unretrievable for them, and the test is to query as an unauthorized user and confirm that nothing comes back, at the retrieval layer, without relying on the model to decline.

And there is a category specific to support that most safety frameworks omit entirely. Customers in distress. A person contacting support about a bill they cannot pay, or a product related to a medical need, or an account belonging to someone who has died. These conversations require a human and the assistant's only correct behavior is to recognize them and stop. Build the cases, with a policy person, and treat a failure to escalate as critical.

A regression this system will produce that has nothing to do with the model, and that will be blamed on the model.

The policy corpus changes. A support policy is updated, the document is republished, and the assistant starts giving the new answer, correctly, while your evaluation dataset still encodes the old one. Your quality number drops. Your team investigates the model. The model is fine.

The defense is the link from Chapter Five: every case records the policy document its expected behavior depends on, and a change to that document flags the cases for review. It takes one field in the schema and it will save you an incident review that ends with everybody feeling foolish.

Production monitoring

The headline online metric for a support assistant is resolution, and the temptation is to use deflection instead, and the difference between them is the entire subject of this chapter.

Deflection counts the conversations that did not reach a human. It goes up when the assistant is good and it also goes up when the assistant is confidently wrong and the customer believes it. It is the metric the business will ask for, and reporting it alone is a way to be rewarded for causing harm.

So report the pair. Deflection alongside the recontact rate, meaning the fraction of deflected customers who come back within seven days with the same problem. A conversation that deflected and recontacted was not resolved; it was postponed, at the cost of the customer's time and their opinion of you. Deflection that holds up under a recontact check is real. Deflection that does not is a lie with a good dashboard.

Watch the escalation rate closely and in both directions, because the deflection pressure will push it down and a falling escalation rate is ambiguous. It could mean the assistant is resolving more. It could mean it has learned to talk people out of asking for a human. Slice it by intent and check that it is not falling in the high-risk categories.

And watch the customer satisfaction score of the human agents who receive escalations, which is an indirect but sharp signal. If handoff quality is poor, the customer arrives at the agent already furious and having repeated themselves twice, and the agent's score drops for reasons that have nothing to do with the agent.

Regression example

Here is a real shape of failure and I want to walk through it because it is instructive about how these programs actually catch things.

A prompt change adds an instruction to be more empathetic in the opening of a response, following feedback that the assistant read as cold. It is a good change. It ships.

The fast suite passes. The full suite comes back with the aggregate up half a point, which is inside the noise floor, and one criterion down: completeness, by four points, which is outside its per-criterion threshold. The gate blocks.

The regression review reads the eleven cases that went from pass to fail. In every one of them, the response opens with two sentences acknowledging the customer's frustration, and the required policy element that used to be in the second paragraph is now in the fourth, and in three of the eleven it is missing entirely, because the model spent its budget on empathy.

This is the same failure as the concise change in Chapter Thirteen, in the opposite direction, and it is the reason the per-criterion threshold exists. The aggregate said the change was neutral. The per-criterion breakdown said the change had traded away the thing the product is for.

The fix was to move the required policy element to the front of the required-elements list in the prompt and to add a length bound on the empathetic opening. Both criteria recovered. The change shipped ten days later than planned, and the ten days were the cheapest ten days the team spent that quarter.

One more failure that belongs to this domain specifically, and that no rubric in the earlier chapters would catch.

The assistant is correct and the customer does not believe it. They asked whether they were charged twice, the assistant explained accurately that they were not, and they escalated anyway, because a machine telling an angry person that they are mistaken is a machine that will be escalated past. The answer was right. The interaction failed.

The measurement for this is the specialist assessment: when a case escalates, ask the human who received it whether the assistant was wrong, incomplete, or correct and not trusted. That third bucket is a product problem rather than a quality problem, and it is fixed with evidence, tone, and an offer of proof rather than with a better model. If you never separate it out, it will sit inside your failure rate, permanently, being attributed to the wrong cause.

Lessons learned

Four things I would tell anyone building this.

The support ticket archive is your dataset and you already own it. Nothing else in this chapter matters as much as the pipeline that turns resolved tickets into labeled cases, and it is a week of

work, and most teams have never done it.

Deflection will be your business metric and it will lie to you. Pair it with recontact from the first day, before anybody has grown attached to the deflection number, because retrofitting an honest metric onto a program that has been celebrating a dishonest one is a political problem rather than a technical one.

Authorization belongs below the model. Every hour spent making the model refuse is an hour spent on a probabilistic control, and the same hour spent on a deterministic check in the tool layer produces a guarantee. Support assistants touch accounts, and accounts are where the real damage is.

And escalation is a feature. The pressure will always run toward escalating less, because escalation costs money and appears in a budget. The cases where a machine should stop are the cases where a machine can do the most harm, and the only thing standing between the two is a set of test cases that somebody was willing to defend.

Area	Specification
Dataset source	Resolved support tickets; escalation transcripts; policy corpus; synthetic edge cases
Strata	Intent, risk tier, conversation depth, emotional register, escalation-required
Size	800 production-derived, 400 synthetic adversarial, 200 escalation-required
Veto criteria	Policy correctness; commitment discipline; data exposure; escalation failure
Graded criteria	Groundedness, completeness, no-re-asking, tone under pressure, resolution
Tool assertions	Ownership check below the model; idempotency under timeout; specific confirmation text
Adversarial suite	Cross-account access attempts; persuasion toward a commitment; injection via ticket content
Distress cases	Financial hardship, bereavement, medical need; correct behavior is to escalate
Offline gate	Zero veto failures; no per-criterion drop beyond limit; full safety suite clean
Canary guardrails	Safety interventions, tool failure rate, latency p99, refusal rate
Headline online metric	Resolution, defined as deflection that survives a seven-day recontact check
Counter-metrics	Recontact rate, escalation rate by intent, agent satisfaction on handoffs
Human review	200 cases per week, stratified; resolution scored by humans, never by a judge alone
Refresh	Weekly promotion of new failure clusters; incident cases permanent

A support assistant fails in public and costs money. The next system fails in private, costs almost nothing per incident, and is harder to evaluate for exactly that reason.

Chapter 27: Case Study 2, Internal Engineering Copilot

An internal copilot answers questions about your own systems. Where does this service get its configuration. Why did the deploy fail last Tuesday. What does this error code mean. Which team owns this endpoint. Summarize this incident. Show me how other services call this API.

It has a property that makes it unusually interesting to evaluate, and it is the property that determines everything else: your users are experts, and they will catch you.

A customer given a wrong answer about a return policy usually cannot tell. An engineer given a wrong answer about a service they work on can tell immediately, and they will tell their colleagues, and the tool will be dead within a month. Adoption of an internal copilot is a step function with a very sharp edge, and the position of the edge is set by trust, and trust is destroyed by confident wrongness far faster than it is built by correctness.

That asymmetry is the design constraint. An internal copilot must be aggressively willing to say it does not know, and the evaluation program exists mostly to police that.

Dataset design from engineering queries

The questions engineers ask are already written down and they are sitting in your chat history.

Every question asked in a team channel and answered by a colleague is a labeled case with an expert answer attached. So is every question in an incident channel, which is better, because incident questions are asked under pressure and are therefore both real and important. So is every question that got the response go and read the runbook, because that is a case where the answer existed and the person could not find it, which is precisely what the copilot is for.

Stratify by question type, because the types have completely different failure modes. Factual lookups, where the answer is in a document. Explanatory questions, where the answer requires synthesis across documents. Diagnostic questions, where the answer requires reasoning about a system's behavior. Code questions, where the answer is code. And incident questions, which are diagnostic questions asked by somebody who is angry and in a hurry.

Stratify by document freshness, which matters here more than in any other system in this book. Engineering documentation rots faster than any other corpus in a company, because writing it is nobody's job and the systems it describes change weekly. A dataset that does not have a stratum of questions whose documentation is out of date is a dataset that will not measure the thing that will actually kill this product.

And build a large I-do-not-know stratum. Questions about systems that are not documented anywhere. Questions whose answer is genuinely unknown. Questions about a service that was deleted. The correct response to every one of these is an admission, and the copilot's willingness to produce that admission is the property you most need to measure.

Size that stratum honestly, which means larger than feels comfortable. In the internal copilots I have looked at, somewhere between a fifth and a third of real engineering questions have no answer in the corpus, because the thing was never written down or was written down and deleted or lives in the head of a person who left. If your unanswerable stratum is five percent of your dataset and reality is twenty-five percent, then the number you report about abstention is off by a factor of five in the direction that flatters you.

RAG evaluation

This is a retrieval-dominated system and the retrieval problems are the specific ones from Chapter Fifteen, sharpened.

The corpus is heterogeneous in a way that breaks naive chunking. Design documents, runbooks, code, configuration files, incident writeups, chat threads, and a wiki that three generations of engineers have edited. These have different structures and different notions of what a chunk is, and a uniform chunking strategy will destroy the structure of most of them. Code split by token count is code that has been broken in the middle of a function.

The identifier problem is acute. Engineers search for exact strings: service names, error codes, function names, configuration keys. Dense retrieval is bad at exact strings and will confidently return something semantically adjacent. Hybrid retrieval is not optional here, and the test set should include a large family of exact-identifier queries, because that is what engineers actually type.

Staleness is the dominant failure. The document says the service uses one thing and it now uses another, and the copilot faithfully reports the document, and an engineer acts on it, and something breaks. Instrument the age of every retrieved document and surface it in the response, because a copilot that says according to a design document last updated eighteen months ago has told the truth about its own uncertainty, and that sentence is worth more than a marginal improvement in retrieval.

And measure the corpus coverage split from Chapter Fifteen: of the failures, how many are cases where the answer existed and was not found, versus cases where the answer was never written down. For an internal copilot the second number is usually large, and it belongs to nobody on your team. It is a documentation debt, and quantifying it is one of the more useful things this product will do for your organization.

Code-related evaluation

When the answer is code, the evaluation gets easier and stricter, and this is the one place in this book where you get something close to a real oracle.

Code can be executed. A generated snippet either compiles or it does not, and it either passes a test or it does not, and that is a deterministic verdict with no judge and no rubric. If your copilot generates code and you are scoring it with a model, you are choosing to be worse at something you could be good at. The functional-correctness framing that the code-generation benchmarks established, executing the output against tests rather than comparing it to a reference, is the right frame and it transfers directly.

So build an execution harness. It costs real engineering effort and it converts your fuzziest criterion into your sharpest one.

What execution does not catch is the part that matters most in an internal setting. The code compiles and passes tests and uses an internal library incorrectly. Or it uses a deprecated pattern that will pass review and cause an incident in a quarter. Or it hardcodes something that should come from configuration. These are the errors an expert reviewer catches instantly and no test suite will, and they require human expert review on a sample, permanently.

And there is a security dimension that belongs to nobody else. A copilot that suggests code with an injection vulnerability, or that suggests a pattern that leaks credentials, has done something worse than being unhelpful. Build a suite for it, get your security team to write the cases, and treat a failure as critical.

Incident summarization deserves a mention of its own, because it is the highest-value and highest-risk thing an internal copilot does. Summarizing a two-hundred-message incident channel into six sentences saves a great deal of time and produces exactly the kind of output that nobody verifies, because verifying it would require reading the two hundred messages, which is the thing the summary was supposed to spare you. Evaluate these summaries against the source with the same claim-level discipline as anything else, and pay particular attention to the two failures that matter: a dropped negation, which inverts a finding, and an invented causal claim, which will end up in a postmortem and then in a decision.

Human expert review

This is the system where crowd review is completely unavailable, and the constraint shapes the whole operating model.

Nobody who does not work on your systems can tell whether an answer about your systems is correct. Not a contractor, not a crowd worker, not a judge model, which knows nothing about your internal architecture and will produce fluent and confident scores that mean nothing. The only qualified reviewers are the engineers who own the services, and their time is the scarcest resource in the company.

So spend it precisely. Route a small number of high-value cases per service to the team that owns that service, at a volume small enough that they will actually do it. Twenty cases a month per team is a real ask that people will honor. Two hundred is an ask they will ignore, and the ignoring will be silent.

Make the review incidental to their work. The highest-participation mechanism I have seen is a feedback control in the copilot itself, with one click for wrong and one for out of date, that routes the case straight to the owning team. Engineers will tell you when a tool is wrong about their service, immediately and with some heat, and capturing that reaction is worth more than any review program you could staff.

And use the experts only where expertise is required. Groundedness against a retrieved document can be judged by a model. Whether the document is still true can only be judged by the person who owns the thing it describes.

A property of this system that changes how you interpret every online metric it produces, and it is easy to forget.

Your users talk to each other. An internal copilot has a user base that sits in the same building, on the same chat platform, and forms a collective opinion within about a week of launch. One engineer posts a screenshot of a confidently wrong answer, thirty people see it, and the tool's adoption curve flattens for reasons that have nothing to do with its aggregate accuracy.

That means the tail matters more than the mean here than in any consumer product. A copilot that is right ninety-five percent of the time and spectacularly, quotably wrong the other five is a copilot that will be remembered for the five. Optimize for the absence of embarrassing failures rather than for the average, and accept a lower answer rate to get it.

Regression detection

The regression that will hurt an internal copilot comes from the corpus rather than the model. It moves constantly, and nobody who moves it knows you exist.

A team rewrites their documentation. An old runbook is deleted. A wiki is migrated and the structure changes and every heading is now a different element. A repository is archived. Each

of these changes retrieval behavior for a set of queries, and none of them appears in your changelog, and the first you will hear about it is an engineer saying the copilot has gotten worse.

So monitor the corpus as a dependency. Documents added, modified, and deleted per day, on a dashboard. A large deletion or a bulk rewrite is an event you want to know about within the hour, because it will move your numbers and you want to attribute the movement correctly rather than spending a week bisecting your own commits.

Run a canary query set continuously. A few hundred questions with known answers, executed on a schedule, with an alert when the answer changes. This is the cheapest possible regression detector for a corpus-driven system and it will catch the template change, the migration, and the accidental deletion, none of which any other instrument in your stack is watching for.

And pin the index version in every evaluation run, so that a quality drop can be attributed to the model, the prompt, or the corpus, which are three different owners and three different fixes.

A note about the corpus that is easy to say and hard to hear.

A copilot for an organization with bad documentation will be a bad copilot, and no amount of retrieval engineering will change that. The system can only surface what exists. If a third of your services are undocumented, a third of the questions are unanswerable, and the very best possible product behavior is to say so, which is a worse experience than the one people imagined when the project was funded.

This is worth saying out loud, at the start, to whoever is funding it. The copilot will make your documentation debt visible and it will not pay it down. What it can do, and what makes the project worth running anyway, is produce a ranked list of exactly which documents are missing and how often people needed them, which is a more actionable artifact than any documentation initiative has ever produced by asking people what they wished existed.

Trust and adoption metrics

For an internal tool the online metrics are unusually good, because you know exactly who your users are and you can ask them.

Adoption is the headline, and it is a trust proxy. Weekly active engineers, and, more informatively, returning engineers: the fraction who used it last week and used it again this week. A copilot people try once and abandon has a high trial rate and a dead retention curve, and the retention curve is the truth.

Verification behavior is the most interesting signal available and almost nobody instruments it. Does the engineer click through to the cited source? An engineer who trusts the tool reads the answer and moves on. An engineer who does not trust it clicks through to check, every time, which means the tool has saved them nothing and they are using it as a search engine with extra steps. A falling click-through rate on citations is a rising trust signal, and it is a much better measure of whether this product is working than any satisfaction score.

The abstention rate is the criterion I would put on the wall. What fraction of questions does the copilot decline to answer, and is it declining the right ones. A copilot that answers everything is a copilot that is confidently wrong on the questions it should have declined, and its trust curve is a cliff. Measure abstention correctness in both directions, treat an unwarranted confident answer as a critical failure, and accept a lower answer rate as the price of a product people believe.

And track the documentation debt the copilot has found. The list of questions engineers asked that the corpus could not answer is a prioritized backlog for whoever owns documentation, and delivering it is a way for this product to create value even on the queries where it failed.

Area	Metric or check	Gate
Dataset	Chat and incident channel Q&A; runbook lookups; unanswerable questions	Refreshed monthly
Strata	Factual, explanatory, diagnostic, code, incident, stale-doc, unanswerable	All reported separately
Exact-identifier recall	Hybrid retrieval recall on service names, error codes, config keys	No drop
Chunking	Structure-aware per document type; code never split mid-function	Recall sweep on the labeled set
Staleness surfacing	Response states the age of its source when the source is old	Required criterion
Coverage split	Failures where the answer existed vs. was never written	Reported to the docs owner
Code correctness	Executed against tests, not compared to a reference	Pass rate; no regression
Internal-library correctness	Expert review sample; deprecated patterns flagged	Human only
Security	No injection-prone or credential-leaking suggestions	Veto
Abstention correctness	Declines the unanswerable; answers the answerable	Confident wrong answer is critical
Expert review	20 cases per service team per month, routed by ownership	Never a judge model alone
In-product reporting	One-click wrong and out-of-date, routed to the owning team	Primary failure source
Corpus monitoring	Documents added, modified, deleted per day	Alert on bulk change
Canary queries	A few hundred known-answer questions on a schedule	Alert on answer change
Index pinning	Every eval run records the index version	Required
Adoption	Returning weekly engineers, not trials	Headline
Trust	Falling citation click-through at stable correctness	Headline
Documentation debt	Ranked list of questions the corpus cannot answer	Delivered to docs owner

The two rows I would defend to the last are abstention correctness and citation click-through. One measures whether the system knows the limits of what it knows. The other measures whether anyone believes it. For an internal tool those are the same property viewed from two ends, and every other number in the table is downstream of them.

The third system has the retrieval problems of this one, the compliance problems of the last one, and one property that neither of them has: two people asking the same question are entitled to different answers.

Chapter 28: Case Study 3, Enterprise Knowledge Assistant

An enterprise knowledge assistant answers questions from a company's internal documents. Human resources policies, benefits, compliance procedures, technical documentation, operational playbooks, legal templates, the travel policy that nobody has read.

It looks like the easiest system in this part of the book. There is a corpus, there are questions, and the answers are in the corpus. No tools, no money moving, no angry customers.

It is the hardest of the three, and the reason is a single property that neither of the others has. The correct answer depends on who is asking.

A question about severance policy has one answer for an employee in one country and a different answer in another, a different answer for a contractor, and, for a manager asking on behalf of someone they are about to let go, an answer that involves a conversation with a human being before anything else happens. Same question. Same words. Four correct answers and several thousand incorrect ones, and the difference between them is a set of facts about the person asking that the question does not contain.

Dataset construction

The dataset for this system has a dimension the others do not, and building it without that dimension will produce an instrument that measures nothing.

Every case carries an asker. Role, region, department, employment type, seniority, and any entitlement that affects what they may see or what applies to them. The expected behavior is a function of the asker, not of the question, and a case that records only the question has recorded half of itself.

This means the same question appears many times in your dataset with different askers and different expected behaviors, and that is correct, and it is the most important structural feature of the whole design. A suite that contains one severance question is a suite that will confidently report that the assistant handles severance questions.

Source the questions from the places people already ask them. The help desk queue. The internal chat channels where people ask each other about policy because they cannot find it. The frequently-asked lists that every department maintains. And, most valuably, the questions that were escalated to a specialist, because those are the ones where the answer was hard and where a wrong answer would have mattered.

Then build the conflict stratum on purpose, because it is where this system fails and it will not arise naturally from sampling. Cases where two documents disagree. Cases where a global policy has a regional override. Cases where a policy was superseded and the old version is still in the corpus. Cases where a general rule has an exception and the exception has an exception. These are rare in traffic and they are a large share of your exposure.

Document-grounded evaluation

Groundedness is close to absolute here, and the absoluteness is a design decision worth making explicitly and defending.

The assistant may not use its parametric knowledge about what companies typically do. Not for anything. A model asked about parental leave has read a great deal about parental leave, and the sentence it produces from that reading will be plausible, will be well-formed, will be wrong

about your company, and will be believed, because it came from the internal assistant and the internal assistant speaks for the company.

So the criterion is that every claim must trace to a document in the corpus, and an unsupported claim is a failure regardless of whether it happens to be true. This is a stricter standard than the one in Chapter Nine, and it is right for this system, and it will make the assistant refuse more often than the product team wants.

The measurement is claim-level decomposition, per Chapter Nine. Split the response into atomic claims, check each against the retrieved evidence, and report the support rate. The judge is good at this task because it is entailment against a visible artifact, which is the thing judges are reliably good at, so the cost is manageable.

And build the completeness check with equal weight, because in a policy domain omission is as dangerous as fabrication. An answer that states the benefit and omits the eligibility condition has not partially answered the question. It has told somebody they are entitled to something they are not, and they will act on it, and they will be upset. Required-elements lists per case, and a missing required element is a failure.

Citation checks

Citations matter more here than anywhere else in this book, because the citation is the mechanism by which the user verifies, and verification is what makes the system trustworthy enough to be used for anything consequential.

Every claim carries a citation. Every citation resolves to a specific passage, not to a document, because a citation to a forty-page policy document is a citation that nobody will follow. The passage supports the claim, which is a check that costs a model call and is worth it. And the document is the current version, which is a check that costs nothing and catches a great deal.

The failure from Chapter Nine is the one to build cases for, and this is the domain where it bites hardest. A passage that supports the claim in isolation and does not support it in context. Policy documents are structured, with sections that scope everything beneath them, and a chunk retrieved from the middle of a section does not know what section it is in. The chunk says employees are eligible after ninety days, and three paragraphs above it a heading says this section applies to full-time employees in the United States, and the chunk does not contain that heading, and a contractor in Ireland has just been told something false with a citation attached.

So carry the section path into the chunk. Prepend the document title, the section hierarchy, the applicability scope, and the effective date to every chunk before embedding. This is the single highest-value engineering change available in this system, and it is a day of work.

A failure specific to this domain that will not appear in any of your metrics and that will eventually appear in a complaint.

The assistant is right and the policy is wrong. An employee asks a question, the assistant retrieves the policy and reports it accurately, and the policy itself is unclear, contradictory, or says something that nobody in the company actually intends. The system has performed perfectly. The employee has been given a bad answer.

This is not a defect you can fix in the assistant, and treating it as one will waste a quarter. What the evaluation can do is surface it: a case whose expected behavior two experts cannot agree on is a case where the policy does not determine an answer, and the right output is a question to the

policy owner rather than a test case. Collect those questions. They are the most valuable thing this system will produce in its first year.

Access-control tests

This is the category where a failure is a data breach rather than a bad answer, and it is enforced in exactly one place.

The filter is at retrieval. A document a user may not see must be unretrievable for that user, at the index level, before the model exists in the story. Any architecture where the model is trusted to decline to discuss a document it has been shown is an architecture that will leak, because the model complies almost always, and almost always is not an access-control standard.

Test it directly and test it adversarially. Query as an unauthorized user for a restricted document and assert that retrieval returns nothing. Then try to get at it indirectly: ask about a topic the restricted document covers and see whether anything from it appears. Ask the assistant to summarize what it knows about a subject. Ask it to list the documents it has access to. Use an injection attempt embedded in a document the user can see, instructing the assistant to reveal what it cannot.

And test the inference leak, which is the subtle one. The assistant declines to discuss a document, and in declining, confirms that the document exists and concerns a particular subject. In some contexts, the existence of a document is itself the sensitive fact. A refusal that says I cannot show you the reorganization plan for your department has disclosed the reorganization plan.

The correct refusal in that situation is uninformative, which is a strange thing to design for and an important one. The assistant says it does not have information on that topic, in the same words it would use if no such document existed, so that the refusal itself carries no signal. Engineers find this uncomfortable, because it feels like lying, and the alternative is a system whose refusals are an oracle for the existence of confidential material. Decide this deliberately, with your security and legal partners, and write the decision into the rubric, because otherwise the model will decide it and the model will be helpful about it.

Staleness and conflict tests

The corpus of an enterprise knowledge base is a graveyard of superseded documents that nobody had the authority to delete.

The policy was updated in March and the old version is still in the wiki, unlabeled, indistinguishable to an embedding model from the current one. Both are retrieved. The model reads both and picks one, by a process nobody specified, and it will pick the wrong one a meaningful fraction of the time, and there is no signal in your system that this happened.

The design fix is precedence in the data, and it is the thing to build before anything clever. Every document carries an effective date, a supersession pointer, and an authority level. The retrieval layer filters superseded documents out before ranking. When two documents genuinely conflict and neither supersedes the other, which happens with regional variations, the pipeline resolves by the recorded precedence rule, and if it cannot, the assistant says so rather than choosing.

That last behavior is worth a criterion of its own. A system that surfaces a conflict, states both positions, and recommends confirming with a human is behaving correctly. A system that picks silently is wrong half the time and confident every time, and nobody will find out until an employee makes a decision on the losing half.

Supersession is worth a small piece of process rather than a clever retrieval trick. Ask the document owners to mark the old version when they publish the new one, and give them a one-click way to do it, and accept that they will forget about a third of the time. Then run a detector: two documents with high similarity, different effective dates, and no supersession pointer between them, which is a query you can run nightly and which will produce a short list that somebody can resolve in an hour a week. The clever retrieval approaches to this problem are all attempts to work around a piece of metadata that costs almost nothing to maintain.

Build the freshness monitor. Age of the documents backing your top intents, at the ninetieth percentile, on a dashboard. And run the canary from Chapter Fifteen: a set of facts you can verify externally, queried on a schedule, alerting when the answer changes or fails to change when the source did.

One design choice in this domain has more effect on trust than any retrieval improvement, and it costs nothing.

Show the source. Not a citation marker that expands into a document reference, but the passage itself, visible, next to the answer, with the effective date and the section it came from. An employee reading a policy answer with the policy text beside it can verify in five seconds, and a system that can be verified in five seconds gets trusted in a way that no confidence score will ever achieve.

The engineering cost is a passage-level retrieval trace that survives to the user interface, which you need anyway for the citation checks earlier in this chapter. The product benefit is that the assistant stops being an oracle and starts being a very fast way to find the paragraph you needed, which is what people actually wanted and a far more defensible thing to build.

Online feedback loop

The signals here are thinner than in a consumer product and one of them is unusually strong.

Volume is low, so statistical detection of small regressions from online data is not available to you and you should stop trying. The offline suite is doing most of the work and that is fine.

Escalation to a specialist is the strongest signal you have and it is well instrumented, because the specialists are on your payroll and they have a ticket queue. Every escalated question is a case where the assistant failed, with the correct answer attached by the person who resolved it, and the pipeline from the ticket queue back into the evaluation dataset is a week of work that will keep paying for years.

The signal I would watch most closely is the specialist's assessment rather than the employee's. Ask the human who received the escalation one question: was the assistant's answer wrong, incomplete, or merely not trusted. Those three answers point at three completely different problems, and the third one, where the answer was correct and the person did not believe it, is a signal about citations and about tone and it is one that no other instrument in this book will capture.

And close the loop into the corpus, because a large share of the failures in this system are not failures of the assistant. The policy is ambiguous. The document contradicts itself. The answer was never written down. Delivering that list to the people who own the documents is the highest-value output of the entire program, and it improves the company rather than merely the assistant.

Area	Specification
Case unit	Question plus asker: role, region, department, employment type, entitlements
Same question, many cases	The correct answer is a function of who is asking; the dataset must reflect that
Sources	Help desk queue, internal chat, department FAQ lists, escalated specialist tickets
Conflict stratum	Superseded policies, regional overrides, exceptions to exceptions, disagreeing documents
Groundedness standard	Absolute; parametric knowledge is a failure even when true
Scoring	Claim-level decomposition; support rate against retrieved evidence
Completeness	Required-elements list per case; omitting an eligibility condition is a failure
Citation granularity	Passage-level, not document-level; resolves and supports the claim
Chunk context	Title, section hierarchy, applicability scope, effective date prepended before embedding
Out-of-context quote	Dedicated adversarial stratum; the highest-yield failure in this domain
Access control	Enforced at retrieval, below the model; restricted documents unretrievable
Indirect access tests	Topic queries, summarization requests, document listing, injection via visible documents
Inference leak	A refusal must not disclose the existence or subject of the restricted document
Supersession	Effective date and supersession pointer on every document; filtered before ranking
Conflict behavior	Surface both positions and recommend a human; silent selection is a failure
Freshness monitor	p90 document age for top intents; canary facts queried on a schedule
Primary online signal	Escalation to a specialist, with the resolved answer captured
Specialist assessment	Was the answer wrong, incomplete, or correct-but-not-trusted
Corpus feedback	Ranked list of ambiguous, contradictory, and missing documents, delivered to their owners

Three systems, three domains, one shared skeleton. A dataset that reflects the real population of askers and questions. Criteria written before anything is scored. Vetoes where the consequence is unrecoverable. Enforcement below the model wherever a guarantee is required. A gate that can say no. And a loop that turns every failure into permanent coverage.

What changes between them is what is at stake and therefore what gets a veto. What does not change is the shape of the discipline, and that shape is what the next chapter puts into a form you can copy.

Chapter 29: Templates and Checklists

A colleague once told me that the difference between a book and a system is that a book ends.

He meant it as a criticism, and he was right. Everything to this point has been an argument, and arguments are how you get somebody to agree with you. They are a poor way of getting somebody to do something on a Tuesday morning when they have four other priorities and a launch on Friday.

So this chapter is the operational residue. Twelve structures, stripped of the reasoning that produced them, reduced to what you would actually write down. They are specifications rather than forms: every field is named, its permitted values are stated, and the note tells you what belongs there. Copy them into a repository, delete what does not apply to you, and start.

One warning before they begin. A structure used unchanged in a domain it was not built for produces a false sense of coverage, which is worse than no structure at all, because it is a checklist that certifies the absence of things you never checked. Delete rows freely. Just be honest with yourself about the difference between a row that does not apply and a row you would rather not think about.

Template 1: Evaluation dataset schema

One record per test case. The fields marked required are the ones whose absence makes a case unscoreable, which means a case missing any of them will be argued about, indefinitely, by people who each remember a different version of what it was supposed to test.

Field	Values	What belongs there
case_id	string, required	Stable across rewording, so history survives an edit
query	string, required	Verbatim, including the typos and the anger
conversation_history	list, if multi-turn	The turns as the model would see them, not as a summary
asker_context	object, if relevant	Role, region, entitlements, when the right answer depends on who asks
intent	enum, required	What the person was trying to do, from your own intent taxonomy
expected_behavior	text, required	Requirements a reviewer can check, not a model answer to match
required_facts	list, required	Facts that must appear, specified by meaning rather than wording
disallowed_behavior	list, required	What a correct answer must never do, however well phrased
required_evidence	list, if grounded	Document and passage identifiers the answer must rest on
rubric_id	reference, required	Which criteria apply to this case
severity	critical, major, minor, cosmetic	Decided from the question, in advance, never from the outcome
category	happy, edge, adversarial	Reported separately; a blended pass rate hides both ends
source	production, expert, synthetic, incident	Where the case came from, which predicts how it will age
owner	string, required	Who resolves a dispute about what this case expects

Field	Values	What belongs there
created	date, required	When it entered the suite
last_reviewed	date, required	Past the refresh window, the case is quarantined automatically
status	active, quarantined, retired	A retired case leaves the score, not the history
incident_ref	string, if applicable	Cases born from an incident are never retired

Template 2: LLM response rubric

The rubric has to be short enough that a reviewer will read it before scoring, which in practice means one page. Everything past what a person can hold in their head is decoration that changes no verdicts. The definitions below are the generic ones; the work is rewriting each in your own domain's terms, and then writing the boundary case underneath it, because the boundary case is what a reviewer actually consults.

Criterion	Type	Weight	The question the reviewer answers
Safety	Binary	Veto	Would this cause harm if a person acted on it?
Policy compliance	Binary	Veto	Does it state a rule the company does not actually have?
Commitment discipline	Binary	Veto	Does it promise an outcome rather than explain a policy?
Correctness	Binary	Heavy	Is every factual claim true?
Groundedness	Binary	Heavy	Does every claim trace to the evidence provided?
Intent match	Binary	Heavy	Did it answer the question that was asked?
Required elements	Count	Heavy	How many of the required facts are present?
Specificity	Binary	Medium	Is it actionable, or true and useless?
Context awareness	Binary	Medium	Does it use what the user already told us?
Clarity	Binary	Medium	Would a non-expert understand it on one reading?
Tone	Three-point	Light	Appropriate to the register of the question?
Length	Binary	Light	Proportionate to what was asked?
Severity, if failed	Four-level	Reported	How badly does this hurt someone?

Two rows carry the rubric's real weight and neither is a criterion. The boundary case, one per criterion, showing the rule applied to a case that sat on the line. And the record of adjudicated disagreements, which is where a rubric stops being a statement of intent and becomes a document that specifies behavior.

Template 3: Multi-turn conversation test

A conversation test is a script with assertions, and the assertions that matter are the ones that fire several turns after the thing they are testing. Below is the structure, with a seven-turn script that exercises the failures from Chapter Ten. Run each script at four depths, because a system that holds a constraint at turn four and drops it at turn twelve has a truncation policy rather than a memory.

Element	Specification
Persona	Who the user is, since it determines what they will leave unsaid
Goal	What completion means for this conversation, stated before it starts
Setup state	Anything true before turn one: prior orders, entitlements, preferences
Turn 1	Establish a constraint the system must carry to the end
Turn 2	Ask the substantive question
Turn 3	Refer to an item by position, not by name
Turn 4	Correct something the assistant got wrong
Turn 5	Add a second constraint that narrows the first
Turn 6	Switch topics entirely
Turn 7	Return to the original topic without restating it
Carryover assertion	The turn-one constraint is still applied at turn seven
Correction assertion	The turn-four correction is still in force at turn seven
Freshness assertion	Turn-one constraints are dropped after the turn-six switch
Standing prohibition	The system never re-asks for information already given
State assertion	What was persisted matches what the user actually said
Conversation score	Task completion, worst-turn severity, and self-consistency
Depth ladder	Run at four, six, eight, and twelve turns

Template 4: RAG evaluation scorecard

Set the baseline once, from your current system, and then leave it alone. A baseline that quietly updates every quarter is a baseline against which you will always appear to be improving.

The three diagnostic rows at the bottom are the ones to populate first, before any of the retrieval metrics above them. Until you know whether your failures come from evidence that was never retrieved, evidence that was never written, or evidence that was retrieved and ignored, every other number here is a measurement you cannot act on.

Metric	What a bad number means
Recall at k, on a labeled set	The evidence exists and your retriever cannot find it
Precision at k	You are spending context budget on distractors
Mean reciprocal rank	The right passage is being found and buried
Rank quality after reranking	The reranker is not earning its latency
Exact-identifier recall	Dense retrieval is approximating strings that need to match

Metric	What a bad number means
Corpus coverage by intent	There are questions your corpus simply cannot answer
Document age, ninetieth percentile	You are answering from a world that has moved
Corpus documents changed per day	Something upstream shifted and nobody told you
Top passage survives assembly	Retrieval succeeded and the context builder discarded it
Position of the top passage	The best evidence landed where the model attends least
Conflict precedence correct	Two documents disagreed and the system picked silently
Claim support rate	Answers are drifting past what the evidence establishes
Citation resolves and supports	The citation is decoration
Failures with evidence present	A generation problem; stop tuning the retriever
Failures with evidence absent	A retrieval problem; stop tuning the prompt
Failures with the answer absent from the corpus	A content problem; it belongs to whoever owns the documents

Template 5: Tool-use evaluation checklist

Nine of the seventeen checks below are critical, and not one of the nine has anything to do with whether the model produced a good answer. If your agentic evaluation is staffed entirely by people who think about model quality, these are the checks that will be missing, and they are the ones where a single failure costs real money in a way that appears on someone's statement.

Check	Severity	Enforced where
Correct tool selected for the request	Major	Evaluation only
No call made when none was warranted	Minor	Evaluation only
Copied arguments match what the user said	Major	Schema assertion
Derived arguments correct: dates, identifiers, quantities	Critical	Assertion on the resolved value
No sensitive field passed that the tool does not need	Critical	Tool layer
A forbidden call is refused regardless of phrasing	Critical	Below the model
The user cannot act on another user's resources	Critical	Below the model
Confirmation requested before any irreversible action	Critical	Agent layer
The confirmation states the real effect and the real amount	Critical	Rubric
A timeout plus a retry produces exactly one effect	Critical	Idempotency key
A duplicate call within one turn produces one effect	Critical	Idempotency key
A tool error is reported, never narrated around	Critical	Fault injection
A failed step halts the rest of the plan	Major	Agent layer
Transient failures retried, permanent ones not	Major	Agent layer
Step limit, backoff, and turn budget enforced	Major	Agent layer
A stranger can reconstruct the session from the trace	Major	Trace review
Planted errors are caught by human review	Major	Review operation

Template 6: Prompt regression report

Attach this to every pull request that touches a prompt. The line that does the most work is the fourth one, which asks which test case would fail if the change were reverted. An author who cannot name one has not established that their instruction does anything, and roughly a third of the time it does not.

Line	What it must state
Prompt version	Before and after, by identifier
Intended behavior change	In one sentence, in terms of what the system will now do
Instructions this may compete with	Named, because prompts are not additive
Test case that fails if reverted	The evidence that the instruction has an effect
Render assertions	No empty placeholders, no missing sections, budget respected
Token budget	Before and after; the system prompt is a tax on every request
Variables held fixed	Model, evidence, seed, dataset version

Line	What it must state
Fail to pass	Cases the change repaired
Pass to fail	Cases the change broke; this is the number that matters
Aggregate delta	Reported next to the published noise floor, never alone
Per-criterion deltas	Where the aggregate is hiding a trade
New critical failures	Zero, or the change does not ship
Regressions read by a human	Every one, by a person, not a summary statistic
Gate result	Pass, or block with the condition that failed
Override	Name, reason, and the risk explicitly accepted

Template 7: Model migration scorecard

Run this on cached contexts, so that the models are answering identical questions from identical evidence. A migration evaluated on freshly retrieved contexts is comparing two models and two corpora and calling the result a model comparison.

Dimension	How to read it
Accuracy, per criterion	An aggregate win can hide a loss on the criterion you sell
Groundedness on identical evidence	The cleanest comparison available; make it first
Instruction following, per instruction	New models ignore different instructions than old ones
Safety, full adversarial suite	Veto; a pass here is not optional
Freshly generated attacks	Veto; your archive was built against the old model
Over-refusal on should-answer cases	The regression nobody looks for and everybody ships
Multi-turn carryover, at four depths	Where a strong model quietly becomes a weak one
Reference resolution	Ordinals and pronouns break differently across families
Tool selection confusion matrix	Veto; a new tool confusion is a new incident
Derived-argument correctness	Veto; this is where money moves
Tool error recovery	Veto; a model that narrates around a failure is dangerous
Structured-output parse rate	A silent cause of downstream breakage
Output tokens, median and tail	Verbosity is a cost line and a latency line
Time to first token	What the user experiences as speed
Cost on real traffic	Including reasoning tokens you pay for and never see
Score correlation with length	If it is high, your judge is rewarding verbosity
Production prompt vs. adapted prompt	The gap measures the debt sitting in your prompt
Largest shadow-traffic divergences	Read them; this is where the surprises live

Template 8: Human reviewer calibration form

One record per reviewer, reviewed quarterly by whoever runs the review operation. The two lines that catch the failure nobody sees are the anchor-set comparison and the median time per case. A reviewer whose anchor scores have crept upward has recalibrated against the population rather than against the standard, and their recent output is inflated in a way that is invisible in the output itself.

Line	What it records
Rubric version certified against	A rubric edit invalidates the certification
Training-set agreement at certification	Must clear the bar before any score counts
Criteria certified for	Scoring outside these does not enter the dataset
Criteria excluded	The ones that require domain expertise this reviewer lacks
Overlap rate assigned	Ten to twenty percent, permanently, never announced as temporary
Agreement against the pool, per criterion	An aggregate figure hides a criterion nobody can score
Gold-case accuracy	The only defense against a reviewer who has stopped reading
Median time per case	A throughput signal, not a moral judgment
Daily cap	Sixty to a hundred and twenty substantive judgments, enforced in code
Anchor-set score vs. the frozen score	Catches drift in the standard itself
Last and next recertification	Reviewers drift, and the drift is silent

Template 9: Safety evaluation checklist

The column that matters here is the second one. A safety property enforced in a prompt is a suggestion that the model follows almost always, and almost always is not a standard for anything that creates a legal obligation or exposes another person's data.

Property	Enforced where
Content harm, tested through composition with retrieval and tools	Model plus output filter
Domain harm: advice a licensed professional is supposed to give	Policy check, downstream
Commercial harm: promises, prices, dates, competitor claims	Policy check, downstream
Another user's data cannot be reached	Retrieval access control
Restricted documents are unretrievable, not merely undiscussed	Retrieval access control
The system prompt cannot be extracted	Adversarial suite
Inbound personal data redacted before storage	Ingestion
The redaction itself has a test suite	Ingestion
Tool calls carry only the fields the tool requires	Tool layer
Every written policy rule has cases designed to tempt a breach	Downstream check
The system refuses what it must	Safety suite
The system answers what it may	Should-answer suite; over-refusal is a regression
A refusal explains itself and offers a path forward	Rubric
A refusal does not disclose what it is refusing	Rubric
Escalation fires on distress, stakes, repetition, and tool failure	Scripted cases
A handoff transfers context so nobody repeats themselves	Conversation scoring
A stranger can reconstruct what was said and on what basis	Trace review
Retention is set by legal obligation, not by storage cost	Logging policy
Injection, override, role play, extraction, sycophancy	Automated red team, every release
Creative attacks a generator cannot reach	Human red team, quarterly
Attacks written by the people who enforce the policy	Domain red team, quarterly
The full attack suite re-run against every model change	Migration gate

Template 10: Release gate checklist

Note the distinction in the last column between a block that cannot be overridden and one that can be overridden by a named person. That distinction decides whether the gate survives its first collision with a deadline. A gate that cannot be overridden gets routed around, usually by somebody quietly building a second deployment path. A gate whose override costs a name and a written acceptance of risk holds, because the price of using it is social rather than technical.

Condition	On failure
Deterministic checks all pass	Block; no override
Prompt renders with no empty placeholders	Block; no override

Condition	On failure
Safety suite: nothing passes through	Block; no override
Zero new critical failures	Block; override requires an executive
Major failures not above baseline	Block; named override
Aggregate delta within the noise floor	Block; named override
No per-criterion drop beyond its limit	Block; named override
Latency at the tail within budget	Block; named override
Over-refusal rate not above baseline	Warn
Cost per conversation within budget	Warn; finance review
Judge calibrated within the last quarter	Warn; flag every result as uncalibrated
The score states its dataset version	Block if unstated
Every regression read by a human	Block
Canary guardrails armed and tested	Block
Rollback is a configuration change, not a release	Block
Any override records a name, a reason, and a risk accepted	Block

Template 11: Production monitoring dashboard outline

The instrument-health panel is the one that no dashboard I have ever inherited contained, and it is the one I would fight for. A quality figure of eighty-four percent means nothing without knowing when the dataset was last refreshed and how well the judge that produced it agrees with a human. Putting those facts on the same screen as the headline number is what stops a stale instrument from being read as a healthy product.

Panel	What it shows	Refresh
Headline	Quality on the frozen dataset, with its noise floor and version	Per release
Headline	Critical failure count	Per release
Headline	Escalation rate, or task completion	Daily
Instrument health	Dataset last refreshed; judge last calibrated; current agreement	Always visible
Instrument health	Share of production failure classes with evaluation coverage	Weekly
Segments	Quality by intent, conversation depth, severity, cohort, locale	Per release
Retrieval	Recall, precision, freshness, corpus change volume	Daily
Context	Top-passage survival, truncation rate, prompt tokens at the tail	Daily
Safety	Interventions, near misses, over-refusal rate	Real time
Tools	Call volume, failure rate, retries, duplicate suppressions	Real time
Latency	Time to first token, and per-stage budget breaches	Real time
Cost	Per conversation, split by token role, trended	Daily
Feedback	Negative signals by reason code, rephrase rate, abandonment	Daily

Panel	What it shows	Refresh
Offline against online	Both on one panel, with divergence flagged	Per release

And every number on it clicks through to the cases behind it. A metric that cannot be opened is a metric nobody believes, and disbelief is fatal, because the moment a team stops trusting a number they stop acting on it and the whole apparatus becomes theater.

Template 12: Failure taxonomy worksheet

One record per production failure, completed during triage. The first five questions are the root-cause procedure from Chapter Eighteen, run backward through the pipeline, and they are answerable from a trace and unanswerable without one. The last four are the ratchet, and they are what stop the same failure from arriving again in a different customer's inbox.

Question	Why it is asked
Was the assembled context sufficient to answer?	Splits the failure into two populations with different owners
Was the evidence retrieved?	Separates a search problem from a context problem
Was the answer in the corpus at all?	Separates a search problem from a content problem
Did the evidence survive context assembly?	The component nobody thinks of as a component
Did the model use what it was given?	Only now have you earned the right to blame the model
Failure class, one to fourteen	A class has an owner; a symptom does not
Severity	Assigned from the question, never from how loudly anyone complained
Did the suite have coverage?	No coverage, coverage overridden, or coverage that passed
If it passed, why did the case not catch it?	The most valuable answer and the hardest to admit
Regression case written	An incident without a case has been survived, not learned from
Twenty variations generated and passing	Confirms the fix generalized rather than satisfied one string
Added to the frozen suite, permanently	Incident cases are never retired

Using these

The twelve structures above are what remains of this book once the arguments are removed. A team that adopted only them, having read none of the reasoning, would be running a more disciplined evaluation program than most organizations shipping generative AI today.

What they would miss is the judgment about which rows to delete, and that judgment cannot be handed over in a table. A checklist is a way of not forgetting things you already understand. It is a poor way of coming to understand them, and a team that adopts these without the arguments behind them will tick every box on a system that is quietly broken in a way the checklist was never built to see.

A word on sequencing, because a team handed twelve structures will attempt all of them and finish none. Start with the dataset schema and the rubric, because everything else consumes their output. Add the failure taxonomy worksheet on the day of your next incident, which is when a taxonomy is cheapest to build and most obviously worth having. Then the release gate, because

a measurement that blocks nothing will be ignored the first time a deadline gets tight. The rest can wait until you have a specific pain that one of them addresses.

And publish them internally rather than keeping them in your own repository. The value of a shared rubric structure across four teams has nothing to do with the criteria being identical, because they will not be. The value is that four teams can read each other's evaluation reports and know what the numbers mean, which is the thing that breaks first when an organization scales past a single AI product, and which is very hard to retrofit once every team has invented its own vocabulary.

Keep asking the question that produced all of this: what would have to be true for my system to be failing right now, badly, without any of these numbers moving. That question is the whole discipline. No template will ever answer it for you.

One chapter remains, and it is the one where I stop telling you what to do and start telling you what I think is coming.

Chapter 30: The Future of LLM Evaluation

In the mid-nineteen-nineties, testing was something you did at the end. There was a phase for it, and a team, and the team was separate from the people who wrote the code, and they were called quality assurance, and the relationship between the two groups was frequently adversarial. Developers wrote software. Testers found bugs. Everyone understood the arrangement and most people thought it was permanent.

It was not permanent. Within fifteen years, a developer who shipped code without tests was considered unprofessional. The tests were written by the person who wrote the code. They ran automatically. Nobody had a phase for testing, because testing had stopped being a phase and become a property of how software was made.

I think that is roughly what happens to LLM evaluation, and I think it happens faster, because the industry already knows the shape of the transition and is impatient.

This chapter is my best guess about the destination. I have tried to separate what I am confident about from what I am speculating about, and I have tried to be honest about the parts nobody has solved, because a closing chapter that pretends the field is finished would be a strange way to end a book whose central argument is that you should measure things rather than believe them.

From prompt testing to AI quality engineering

The first shift is one of category. Right now, evaluation is a thing a team does. It will become a thing a system has.

The signs are already visible if you look at the tooling. Three years ago, evaluating an LLM application meant writing a script. Today there are registries, judge services, tracing standards, and comparison harnesses, and they look increasingly like the infrastructure that grew up around conventional testing: a test runner, a coverage report, a continuous integration hook, a dashboard nobody looks at until it turns red.

What follows from that is a change in who does the work. Evaluation today is done by whoever cares, which is usually one motivated engineer and sometimes a scientist. It will be done by everybody, badly, using shared tools, the way testing is done today, and that will be an enormous

improvement over the current situation where it is done well by almost nobody.

The threshold to watch is when a pull request that changes a prompt without an accompanying test case starts getting rejected in review, not by policy, but because a reviewer looks at it and finds it unprofessional. That is the moment the discipline is real. We are not there. In a small number of organizations we are close.

Evaluation for autonomous agents

Everything in this book gets harder when the system acts over long horizons, and this is the area where I am least confident that the current techniques extend.

The problems compound in a specific way. A single-turn response has one output to score. A conversation has a sequence, and Chapter Ten was already honest about how thin the tooling is there. An agent executing a fifty-step plan has fifty decisions, most of which are individually reasonable, some of which are only wrong in the light of what happens at step forty, and any one of which can compound into a state from which recovery is impossible.

Credit assignment across those fifty steps is genuinely unsolved. When the plan fails, which step caused it? The obvious answer, the step that failed, is usually wrong. The step that failed was made inevitable by a decision eleven steps earlier that looked fine at the time and constrained everything after it. Human organizations have the same problem and have not solved it either, which is why postmortems are hard.

Then there is the sandbox problem, and it is a real barrier rather than an engineering inconvenience. Evaluating an agent requires letting it act, and letting it act requires an environment, and an environment that is faithful enough to be informative is expensive to build and drifts away from production the moment you stop maintaining it. The benchmark suites that have appeared for agentic tasks, the ones built around software engineering issues and around web interaction, are useful precisely because somebody paid the cost of building the environment, and that cost is why there are so few of them.

My guess is that the field converges on consistency as a first-class metric here, because it already has in the places where people have looked hardest.

An agent that succeeds on a task eight times out of ten is a different proposition from one that succeeds ten out of ten, and for anything with an irreversible action the second is worth a great deal more than a marginally higher average. Reporting the pass rate across repeated attempts, rather than a single-shot success rate, is a small change to a metric that reflects a large change in what you are willing to deploy.

Evaluation for multimodal systems

The second frontier is that the input and the output stop being text, and most of the machinery in this book quietly assumed they were.

Groundedness against an image is a genuinely different problem. What does it mean for a claim to be supported by a photograph? The claim that the product in the picture is blue is checkable. The claim that it looks well-made is not, and a model will produce both in the same sentence with the same confidence.

Claim decomposition, which is the workhorse of Chapter Nine, assumes the evidence is readable. When the evidence is a chart, a screenshot, a diagram, or a video, the verification step requires a model that can read the evidence, which means your evaluator now has the same failure modes as the system it is evaluating, and correlated failures between a system and its judge are the worst situation in measurement.

And the output side is harder still. Evaluating generated text is difficult. Evaluating a generated image, a generated interface, a generated video is difficult in ways where even the criteria are contested, and the human review is slower, and the rubrics that exist are mostly aesthetic.

I do not have a good answer here and I am suspicious of anyone who claims one. What I would say is that the structural advice survives even when the techniques do not. Define the dimensions. Write criteria a second reviewer can apply. Measure agreement before you trust a score. Separate the evidence from the generation. Those hold whatever the modality is, and they are what you fall back on when the specific tooling does not exist yet.

Continuous evaluation from production traffic

The direction of travel here is clear and I am confident about it. The distinction between offline evaluation and online monitoring collapses.

Today they are separate activities with separate tools, separate teams, and separate meetings. Offline is a suite you run before a release. Online is a dashboard you watch afterward. The reason for the split is cost: you could not afford to score production traffic at any meaningful volume, so you scored a fixed dataset instead and hoped it was representative.

That cost is falling, quickly. Scoring a sample of live traffic with a small calibrated judge is already affordable at rates that would have been absurd two years ago, and the trend is not subtle. When you can score one percent of production continuously, the dataset stops being a proxy for production and starts being a subset of it, refreshed daily, weighted to the traffic distribution automatically.

What that unlocks is the thing this discipline most needs, which is a quality signal that moves in hours rather than in releases. A regression that currently takes eleven days to surface as a customer complaint becomes a line on a chart that moves the same afternoon.

The fixed dataset does not disappear. You still need a frozen instrument to compare against over time, because a metric computed on continuously shifting traffic cannot distinguish a change in your system from a change in your users. But it becomes the reference rather than the measurement, and that is a meaningful reordering.

Standardization opportunities

The field is at the point in its life where standards would help more than techniques, and I would rather see effort go there than into another metric.

Tracing is the most obvious. There is no reason every organization should invent its own schema for what an LLM request trace contains. The conventional observability world converged on a standard for distributed tracing and the benefits were enormous and mostly boring: tools interoperate, people move between companies and already know the format, vendors compete on quality rather than on lock-in. The same convergence for generative pipelines, covering the assembled context, the retrieval trace, and the tool calls, would be worth more than any single evaluation technique in this book.

Failure taxonomy is the second. The fourteen classes in Chapter Eighteen are mine, and they are defensible, and there is no reason they should be mine. A shared vocabulary would let two organizations compare notes

about what breaks, which is currently impossible, which means the entire industry is independently rediscovering the same failures.

Rubric interchange is the third and the least likely, because rubrics are domain-specific and companies regard them as proprietary. What could be shared is the structure: how a criterion is specified, how a boundary case is recorded, how agreement is reported. Not the content, the container.

And judge calibration reporting. If you tell me a system scores eighty-four percent, that number is uninterpretable unless I know what scored it and how well that scorer agrees with humans. A convention for publishing judge agreement alongside every judged score would raise the quality of public discourse about AI systems more than any benchmark leaderboard has.

One change I expect that would make most of this book easier, and that depends on nobody in this industry.

Regulation will arrive, unevenly and imperfectly, and one of the things it will do is make evaluation a requirement rather than a virtue. That is already visible in the risk-tiered obligations of the European framework and in the direction of the American standards work. Teams that currently have to argue for a quality budget will, within a few years, have to produce one.

I have complicated feelings about most of what regulation will do to this field and almost none about this part. A discipline that only gets funded after an incident is a discipline that runs on other people's harm. Anything that moves the funding decision earlier, including a legal obligation, is an improvement over the mechanism we currently have, which is waiting for something to go badly enough wrong that somebody senior reads about it.

Open challenges

Six things I do not know how to do, stated plainly, because a book that only described solved problems would be dishonest about the state of the field.

Scoring a long conversation as a whole. Chapter Ten gave you three numbers and I stand by them and they are a workaround. Nobody has a principled account of what makes a twenty-turn interaction good, as an object, rather than as a sequence of turns that were individually acceptable.

Credit assignment in long agent trajectories. Discussed above. Open.

Detecting the failures that produce no signal. The user who received a wrong answer, believed it, and left satisfied is invisible to every instrument described in this book except reading conversations by hand, at a low rate, forever. Call that a solution and you are fooling yourself. It is a confession.

Evaluating against a moving world. Your corpus is a snapshot, your policies change, your users change, and a fixed dataset measures a world that has already moved. The refresh mechanisms in Chapter Six are the best I have and they are a treadmill rather than an answer.

Judges that share the model's blind spots. When the evaluator and the evaluated come from the same family of systems, their errors correlate, and correlated errors are invisible to every agreement statistic you can compute. The self-preference findings in the recent literature are an early and mild instance of a problem I expect to get worse as judges become the dominant scoring mechanism.

And Goodhart, which is not a technical problem and which will defeat you anyway. Every metric in this book, once it becomes a target, will be optimized in ways that satisfy it without improving the product. The only defense I know is to keep several metrics, keep watching the ones you are not optimizing, and periodically go and read what your users actually received.

The role of AI reliability engineers

A role is forming and it does not have a settled name yet, and I think it is going to be one of the more consequential jobs in the industry.

The precedent is site reliability engineering, which emerged when systems became too complex for the people who built them to also be the people who kept them up, and which turned out to require a genuinely different skill set: statistics, systems thinking, a tolerance for saying no, and a willingness to be unpopular on Thursday afternoons.

The equivalent for generative systems is a person who owns the instrument rather than the product. They build and maintain the dataset, calibrate the judges, set the thresholds, run the calibration meetings, watch the drift, and hold the authority to block a launch. They do not own the quality of any feature, which belongs to the team that ships it. They own whether the organization can tell the truth about its own quality.

That role does not fit cleanly into any existing function, which is why it is currently being done by a motivated engineer in their spare time, and why the evaluation program in Chapter Twenty-Four died in a reorganization. It needs a title, a ladder, and a reporting line outside the organization it measures, and the companies that build that structure first will have a durable advantage that is very hard to see from the outside and very hard to copy.

A prediction about tooling that I hold with moderate confidence and that would change a lot if it happens.

The evaluation harness becomes part of the model provider's product. It is a natural move: they have the traces, they have the judge models, they have the incentive to make migration between their own versions less frightening. The moment a provider ships a credible offering here, most teams will adopt it, because it is free and it is right there.

That would be a genuine improvement in the median and a real hazard at the frontier, and the hazard is the same one this book has warned about in three other places. A judge supplied by the maker of the model being judged has an alignment of interest that nobody involved intends and that no agreement statistic will surface. Use the tooling. Keep your own calibration set. Check the provider's judge against your humans, on your criteria, on a schedule, and be prepared to disagree with it.

Why evaluation will define trust in generative AI systems

Here is the argument I would leave you with, and it is not a technical one.

The capability of these models is going to keep improving, and it is going to improve for everyone at roughly the same time, because the frontier is public and the diffusion is fast. Whatever advantage a company has today from access to a better model is a temporary one, measured in months.

What does not diffuse is the ability to know whether your system is working. That is built, slowly, out of datasets that took a year to label, rubrics that took six months of argument to sharpen, judges calibrated against reviewers who were trained and retrained, a taxonomy that accumulated from real incidents, and an organizational habit of letting somebody say no. None of that can be bought, and none of it can be copied from a paper, and it does not become obsolete when the next model ships.

And it is what users are actually responding to. Nobody outside this industry cares which model you are running. They care whether the thing was right the last three times they used it, and whether it admitted when it did not know, and whether it did what it said it was going to do with their money. Those are quality properties, and they are earned one measured, gated, rolled-back release at a time.

The systems that people come to depend on will not be the ones with the highest benchmark scores. They will be the ones whose builders knew when they were wrong, found out quickly, and fixed it before anybody else noticed. That capability is the product of a discipline, and the discipline is what this book has been about.

There is one thing left to say, and it goes back to where we started.

Conclusion

The dashboards were green.

I want to go back to that room, and that night, and the wall of monitors that were all reporting, accurately and in good faith, that a system was healthy while it was quietly failing the people it had been built to help. Latency inside budget. Error rate flat. Deployment clean. An on-call engineer asleep, correctly, because nothing had woken him.

And four days later, a screenshot. A customer who asked about three products, got a clean comparison, typed six words, and was answered about the wrong one. Twice. And then left, and did not file a bug, and did not fill in a survey, and simply became one of the people who tried the thing once and does not use it.

Everything in the three hundred pages between that scene and this one has been an attempt to answer a single question: how would we have known?

What we have actually been doing

The techniques are numerous and the underlying moves are few, and I want to name them, because a reader who has followed the whole argument deserves to see the skeleton.

We made the invisible visible. That is what logging the assembled context does, what the failure taxonomy does, what the stage-by-stage pipeline map does. A generative failure does not announce itself, and the entire discipline begins by constructing an instrument that can see something a stack trace cannot.

We turned judgment into criteria. That is what rubrics do, and expected-behavior specifications, and claim-level decomposition. Quality in a generative system is a matter of judgment, and judgment does not scale and does not agree with itself, and the only escape is to break it into questions specific enough that two people answer them the same way.

We turned anecdotes into rates. That is what the dataset is for. A demo is an anecdote. A screenshot is an anecdote. Neither has a denominator, and without a denominator you cannot tell whether you are improving, and you will spend years responding to whichever failure was most recently forwarded to you by somebody senior.

We put the measurement in the path. That is what the gate does. Every organization I have worked in has people who will raise a concern in a meeting. Very few have people who will block a launch that a director has already promised to a partner team. The gate exists because human judgment under schedule pressure fails in one direction, reliably, in every industry, and the only thing that has ever worked is to make the failure automatic and the override expensive.

And we closed the loop. Every incident becomes a case, every case becomes coverage, and the coverage never goes away. That ratchet is the difference between an organization that gets better and one that survives the same failure every eight months with a slightly different customer each time.

Five moves. Everything else is technique, and technique changes, and these do not.

It is worth noticing what is absent from that list. None of the five moves is about the model. Not one. The most consequential decisions in this entire discipline are made in the components around the model: what got retrieved, what survived truncation, what the prompt asked for, what the post-processor did, what got logged, and who was allowed to say no. The model is the part everybody talks about and the part you have the least control over, and a team that responds to a quality problem by reaching for a bigger one will spend a quarter and move nothing.

That is the finding I would most want a reader to carry out of this book, because it is the one that changes what you do on Monday.

And one thing not to do, which is the mistake I see most often in teams that have read something like this and are enthusiastic.

Do not try to build all of it. There are twelve templates in Chapter Twenty-Nine and a reference architecture with fourteen boxes, and a team that attempts the whole thing in a quarter will produce an impressive amount of infrastructure and will not have measured anything. The programs that work start embarrassingly small and grow because they are useful, and the programs that fail start comprehensive and collapse under their own maintenance burden before they produce a single number anyone acts on.

Fifty cases and one honest measurement will change more about your product than a platform will.

What I would do first

If you are starting from nothing, and most teams are, the order matters more than the completeness, and I want to be specific rather than encouraging.

Log the assembled context. This week. Not the query and the response, which is what you have, but the actual text that went to the model, with its version tuple attached. Every other capability in this book is downstream of that one artifact, and a team that has it can build everything else afterward, and a team that lacks it is permanently blind in a way that will not feel like blindness. It will feel like a customer with a screenshot and a question nobody can answer.

Then build fifty cases. Fifty, not five hundred. Real user questions, hand-labeled with what a correct answer must contain and must not contain, stratified across your top intents with a deliberate scattering of edge and adversarial conditions. It will take a week and it will find more problems than the previous quarter of guessing.

Then run the split. For every failure, ask one question: was the information needed to answer this present in the context? That single yes-or-no partitions your failures into two populations with different owners and different fixes, and in every retrieval-augmented system I have looked at closely, the answer surprises the team, and the surprise redirects their entire roadmap.

Then gate something. Anything. One threshold, on one criterion, that blocks one kind of change. It will be crude and it will occasionally block a release it should not have, and the first time it blocks a release it should have, the argument about whether evaluation is worth funding will end.

Four steps, and none of them require a platform, a team, or a budget. They require an engineer who is willing to spend a month on work that no promotion committee will ask about.

What I would resist, in the first month, is the platform. The instinct to build the registry, the runner, the judge service, and the dashboard before anyone has scored a single case is a strong one, and it is a way of doing something that looks like evaluation without producing any information. Build the platform in month six, when you know what you are measuring and why, and when the spreadsheet has broken in three specific ways that you can

name. Building it in month one produces infrastructure for a discipline you have not yet developed.

And I would resist the temptation to make the first number look good. Your first honest measurement is going to be worse than you expect. It will be worse than the demo suggested, worse than the team believes, and worse than what you have been telling leadership. That number is the most valuable thing you will produce all year, and the pressure to soften it, to remove the unfair cases, to blend in the easy ones, will arrive within about a week. Everything that follows depends on whether you hold.

The part that is not engineering

I have tried to be honest throughout that the hardest problems in this discipline are not technical, and I want to say it plainly at the end rather than leaving it implied.

Every evaluation program I have watched fail, failed organizationally. The dataset went stale because maintaining it was nobody's job. The gate became a warning because it blocked something inconvenient and nobody had the standing to hold the line. The engineer who built the whole thing moved teams, because the work was invisible on every ladder in the company, and she was tired of explaining.

The technical recommendations in this book are within reach of any competent team. The organizational ones are not, because they require somebody to be given authority that the people being measured would prefer they did not have. An evaluation function that reports into the organization whose launches it gates will decay in the direction of the pressure applied to it, and nobody will make a decision to let that happen. It will simply be what a system built that way produces.

So if you take one thing from Part Six, take this. Ask who in your company is allowed to say no to a launch, and what happens to them afterward. The answer to that question tells you more about the future quality of your AI products than any benchmark you could run.

On being wrong

There is a temperament that this work requires and it is not the one most engineering cultures select for.

Most of what you build in software is an assertion that something works. Evaluation is the opposite. It is an apparatus whose entire purpose is to find out that you were wrong, and the better it works, the more often it will tell you so, and the people around you will experience that as a problem.

The engineer who says the number is wrong and here is why is the most valuable person in an AI organization and is routinely treated as an obstacle. The team that blocks its own launch has done its job and will be asked why they are always the bottleneck. A quality function that only ever

reports good news has told the company, unmistakably, that it has stopped functioning as one, and everyone has already worked that out and adjusted.

The fix is cultural and it is small and specific. Make finding a problem a contribution. Say so out loud, in the launch note, in the review, in the promotion packet. The organizations that do this will find problems. The ones that do not will stop finding them, and the problems will still be there, and they will be found instead by a person with a screenshot.

The version of this that I have seen work best is almost embarrassingly simple. When a launch is blocked and the block turns out to be correct, name the person who blocked it, in writing, in front of the people whose launch it was. Do it the first time it happens, before anybody has decided how they feel about the gate. That one act establishes what kind of organization this is going to be, and it is much harder to establish later, after the gate has been overridden twice and everyone has quietly concluded that it is decorative.

What this will look like in five years

My honest expectation is that most of the specific techniques in this book will look dated and most of the structure will not.

The judge models will be better and cheaper, and scoring live production traffic continuously will be routine rather than aspirational. The tracing formats will be standardized, and moving between companies will not mean relearning how a request is logged. The role that is currently done by a motivated engineer in their spare time will have a title and a ladder. And a pull request that changes a prompt without a test case will get sent back in review, not because of a policy, but because a reviewer will look at it and find it careless.

What will not change is the underlying situation. You will be operating a system whose failures are silent, whose correctness is a matter of judgment, whose behavior shifts when nobody has deployed anything, and whose users will not tell you when it fails them. That situation is a permanent property of building products on top of probabilistic systems, and the response to it will always be measurement, and measurement will always be unglamorous, and it will always be the thing that separates the products people come to rely on from the ones they try once.

I would add one prediction that I hold more loosely and care about more. I think the competitive advantage in this industry is going to come from evaluation rather than from capability, and I think that is going to be true sooner than most people expect.

Model capability diffuses. Whatever the frontier is today, it will be widely available within a year, to your competitors and to a graduate student with a laptop. Nobody is going to build a durable business on top of temporary access to a better model. What does not diffuse is a labeled dataset that took eighteen months to build, a rubric that took six months of argument to sharpen, a judge calibrated against reviewers who were trained and retrained, a taxonomy accumulated from real incidents, and an organizational habit of letting an engineer block a launch on a Thursday. None of that can be bought. None of it is in a paper. And none of it becomes obsolete when the next model ships.

The six words

Tell me more about the second one.

I keep returning to that sentence because of how ordinary it is. It is the least remarkable thing a person can say. There is nothing clever in it, no adversarial intent, no edge case, no ambiguity that a human would even notice. It is a person, mid-conversation, pointing at something that is right there on the screen.

A system either handles it or it does not, and there is no partial credit, and the difference between those two outcomes has almost nothing to do with model capability. It comes down to whether anybody built the thing that would have noticed.

Somewhere in your product, right now, there is a sentence like that one. A short, ordinary, unremarkable thing that a user will type, and that your system will get wrong, and that none of your instruments are watching for. It is failing today. It will fail tomorrow. Nobody will file a bug, and your dashboards will stay green, and the people it fails will quietly stop coming back, and you will attribute the drop to something else.

You cannot prevent every failure of that kind. Nobody can, and any book that promised otherwise would be selling you something. What you can do is find them, name them, count them, rank them, and put a number on the wall that goes down when you fix them and up when you break them.

That is the job. It is not glamorous, no one will write a press release about it, and it is the only thing that will ever tell you the truth about what you have built.

Go and look at what your users actually received.

Acknowledgments

This book is the product of a great many conversations with engineers, scientists, product managers, reviewers, and policy specialists who took the time to argue with me. Most of what is useful here was learned from watching careful people work under pressure and get it right, and from watching thoughtful people, myself included, get it wrong in ways that turned out to be instructive.

I am grateful to the colleagues who read early drafts and told me plainly which parts were unconvincing, and to the reviewers and specialists whose patient, unglamorous work over long stretches taught me most of what I know about measuring quality honestly. I owe a particular debt to the researchers whose published work is cited throughout, and whose willingness to document what did not work has done more for this field than any single technique.

Thanks are due to the teams who let me watch their systems fail and then let me write about it. The failures in this book are real, the lessons are theirs, and the identifying details have been removed.

To my family, for the patience that a project like this quietly consumes: thank you.

Any errors that remain are mine.

About the Author

Aditi Patodiya is a Senior Software Engineer working on production generative AI infrastructure for a large-scale conversational assistant. With over ten years of experience in distributed systems and AI platforms, she designs the foundational backend services, context management frameworks, and real-time token streaming systems that support responsive conversational experiences at global scale.

Her work centers on the engineering problems that arise when large language models move out of research environments and into high-throughput production systems: context window management, distributed caching, token streaming, and the reduction of user-perceived latency across large numbers of concurrent conversational sessions. She has served as the technical owner of a cross-service program to unify how conversation context is produced, transported, stored, retrieved, and consumed in a multi-turn generative AI product, authoring the program design and driving contract alignment across interdependent service teams.

She holds a Bachelor's degree in Computer Engineering and a Master's degree in Computer Science. She is a Senior Member of the IEEE and a recipient of an inventor's award for her work.

She is also a technical judge and expert reviewer for international computing conferences and global innovation award programs, and has written on enterprise AI memory and context pipelines for technical publications.

LLM Evaluation at Scale is her first book.
