

# DeepEval

*Building an Automated LLM Evaluation Harness That  
Engineers Trust*

Trex Team

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by NobleTrex Press



© 2026 by NOBTREX LLC. All rights reserved.

For permissions and other inquiries, write to:

P.O. Box 3132, Framingham, MA 01701, USA

This book is for educational and informational purposes only. The author and publisher make no guarantees about the accuracy or completeness of its contents and accept no liability for any loss or damage resulting from its use. Software tools such as large language models have been applied to assist in the writing and editing of this book. All product names, logos, and trademarks are the property of their respective owners. References to third-party products or names are for identification only and do not imply endorsement.

# Contents

- 1 Framing DeepEval as an Engineering Trust System 5**
  - 1.1 Why LLM Systems Need an Evaluation Harness . . . . . 5
  - 1.2 What DeepEval Actually Provides . . . . . 13
  - 1.3 Trustworthy Evaluation Principles for Experienced Teams 22
  - 1.4 From Product Risks to Evaluation Requirements . . . . . 30
- 2 Modeling Evidence with Test Cases, Datasets, and Goldens 41**
  - 2.1 LLMTestCase as the Atomic Unit of Evaluation . . . . . 41
  - 2.2 ConversationalTestCase and Turn Semantics for Multi-Turn Evidence . . . . . 48
  - 2.3 Designing Goldens That Stabilize Evaluation Over Time 56
  - 2.4 Choosing Coverage for Real Product Behavior . . . . . 63
- 3 Selecting Metrics and Understanding LLM-as-a-Judge 71**
  - 3.1 Metric Families as Failure-Mode Instruments . . . . . 72

|          |                                                                                            |            |
|----------|--------------------------------------------------------------------------------------------|------------|
| 3.2      | How LLM-as-a-Judge Metrics Really Work . . . . .                                           | 81         |
| 3.3      | Designing Robust Custom Metrics with G-Eval . . . . .                                      | 90         |
| 3.4      | Metric Composition, Weighting, and Conflict Resolution                                     | 98         |
| <b>4</b> | <b>Engineering Trustworthy RAG Evaluation</b>                                              | <b>107</b> |
| 4.1      | Evaluating RAG as a Decomposed Retrieval-to-Answer Pipeline . . . . .                      | 107        |
| 4.2      | Measuring Answer Relevancy and Faithfulness in Grounded Generation . . . . .               | 114        |
| 4.3      | Interpreting Contextual Precision, Recall, and Relevancy for Retriever Diagnosis . . . . . | 121        |
| 4.4      | Building RAG Benchmark Suites That Expose Root Causes                                      | 130        |
| <b>5</b> | <b>Evaluating Multi-Turn and Agentic Systems</b>                                           | <b>139</b> |
| 5.1      | Conversation-Level Quality Beyond Single-Turn Accuracy                                     | 139        |
| 5.2      | Designing Multi-Turn Metrics and Scenario Suites . . .                                     | 149        |
| 5.3      | Tool Use, Plans, and Intermediate Execution State . . .                                    | 158        |
| 5.4      | Task Completion, Tool Correctness, and Step Efficiency                                     | 167        |
| <b>6</b> | <b>Scaling the Harness with Simulation, Synthetic Data, and Workflow Automation</b>        | <b>177</b> |
| 6.1      | Generating Synthetic Goldens Without Corrupting Benchmark Validity . . . . .               | 178        |
| 6.2      | Simulating Multi-Turn Conversations for Targeted Failure Coverage . . . . .                | 186        |

- 6.3 Building Local Evaluation Loops and CI/CD Gates That Engineers Will Actually Use . . . . . 196
- 6.4 Operating Historical Comparisons and Release Governance from Stable Evidence . . . . . 204
- 7 Calibration, Debugging, and Evolving a Trustworthy DeepEval Program 213**
  - 7.1 Reading Scores as Evidence Instead of Truth . . . . . 214
  - 7.2 Debugging Failure Signals Across Prompts, Data, Metrics, and System Boundaries . . . . . 225
  - 7.3 Adapting the Harness to DeepEval Capability Milestones and Interface Shifts . . . . . 236
  - 7.4 Operating an Evaluation Program Engineers Continue to Trust . . . . . 246

# Introduction

The development of Large Language Model (LLM) applications requires transitioning from manual spot checks to systematic, automated testing. When engineering teams deploy LLMs into production environments, they require rigorous, repeatable frameworks to verify system behavior. This book provides a comprehensive guide to building an automated evaluation harness using DeepEval. It frames LLM evaluation strictly as an engineering trust system. Rather than aggregating arbitrary benchmark scores, the text demonstrates how to establish repeatable evidence, define measurable quality criteria, and implement risk-aligned quality gates for operational decision-making.

A reliable evaluation system depends on precisely structured evidence. The initial chapters establish the core abstractions of the DeepEval framework, defining how single-turn generations and multi-turn conversations must be modeled as discrete test cases. Readers will learn how to design golden examples and construct stable benchmark datasets. By encoding realistic scenarios, expected behaviors, and explicit failure surfaces into these datasets, engineering teams can guarantee that longitudinal comparisons remain valid even as underlying foundation models and application prompts are modified.

Once the evidentiary baseline is established, the text examines the metric systems used to generate operational signals. It details the internal

mechanics of LLM-as-a-judge methodologies, analyzing prompt sensitivity, model variance, and rubric interpretation. Readers will learn to implement custom metrics that encode strict, domain-specific quality standards. The book applies these measurement principles directly to Retrieval-Augmented Generation (RAG) systems. By treating RAG as a decomposable architecture, the text provides exact methodologies to isolate defects, measuring retrieval precision, contextual relevancy, and factual faithfulness as distinct, verifiable components.

As applications increase in complexity, evaluations must account for extended interactions and autonomous processes. The book extends the DeepEval harness to encompass conversational interfaces and agentic workflows. For multi-turn systems, it addresses the evaluation of dialogue memory, system consistency, and conversational progression. For agentic applications, the evaluation model shifts to inspect process artifacts. The text demonstrates how to evaluate execution traces, API tool calls, intermediate arguments, and step efficiency, ensuring that the system is assessed on its operational logic rather than its final terminal output alone.

To integrate these capabilities into standard software engineering workflows, the evaluation harness must operate at scale. The concluding chapters detail the application of synthetic data generation and dialogue simulation to expand test coverage. The text outlines precise strategies for executing evaluations within continuous integration and continuous deployment pipelines, establishing automated regression checks and release-blocking failure thresholds. Furthermore, it provides systematic workflows for score calibration, false-positive management, and harness debugging.

Ultimately, this book provides the technical specifications and operational patterns necessary to maintain a mature evaluation program. By mapping product risks directly to testable criteria, engineering teams can utilize DeepEval to deploy LLM applications with demonstrable,

documented reliability.



# Chapter 1

## Framing DeepEval as an Engineering Trust System

*This opening chapter starts from an uncomfortable truth: most teams can demo an LLM system long before they can justify trusting it. The chapter's core message is that DeepEval matters not as a score generator, but as an engineering trust system that turns product risk into evidence, evidence into evaluation, and evaluation into operational judgment.*

### **1.1. Why LLM Systems Need an Evaluation Harness**

A language model can look polished in a demo and still fail as a product component. The failure mode is rarely dramatic. More often, the

system answers confidently on the five prompts you remembered to try, then drifts when the phrasing changes, the retrieval index updates, or a tool call resolves to an edge case you did not anticipate. That gap between apparent competence and sustained reliability is what forces evaluation to become an engineering control rather than a research accessory.

A practical way to see the difference is to compare ad hoc testing with a harnessed workflow:

```
deepeval test run \  
  --dataset eval_cases.jsonl \  
  --repeat 5 \  
  --cache \  
  --ignore-errors
```

The command itself is not the point. The point is the discipline it implies: you preserve test cases, rerun them under controlled conditions, compare repeated outcomes, and keep the evaluation traceable. Without that structure, every prompt trial becomes a one-off judgment call. With it, you can ask whether a change improved the system, whether a regression is real or just stochastic noise, and whether a pass is strong enough to trust in production.

That distinction matters because LLM systems are not stable in the way conventional deterministic software is stable. A database query returns the same rows until the data changes. An API either honors a contract or violates it. An LLM application, by contrast, is sensitive to prompt wording, sampling variance, context ordering, retrieval composition, and hidden changes in the model provider's backend. Even when the code around the model is unchanged, the observed behavior can shift. The model may still be useful, but usefulness is not the same as reliability. Reliability requires a repeatable way to observe behavior under defined conditions.

Trust, in this setting, is not a feeling about intelligence. Trust is a claim

about observed behavior under controlled circumstances. If you say the system is trustworthy, you need to specify for which inputs, under which constraints, with what tolerance for error, and relative to which baseline. That makes trust measurable rather than rhetorical. It also makes trust testable: if the system cannot satisfy the same expectation twice in a row, under the same evaluation setup, you do not yet have a quality signal you can build on.

Repeatability is the first requirement of that trust model. A repeated evaluation does not need to produce identical token sequences, but it must produce stable enough judgments that you can distinguish real changes from noise. That means preserving the input set, freezing the evaluation criteria, and isolating the variables that you intentionally changed. If you modify the prompt, the retriever, and the judge rubric at the same time, you have no attribution path. The result might improve, but you will not know why. In engineering terms, you have lost causal visibility.

Regression detection is the second requirement. LLM products almost always evolve in ways that create cross-effects. A prompt revision that improves factual precision may worsen tone. A new retrieval filter may reduce hallucinations while silently dropping useful context. A model upgrade may improve one language and harm another. A multi-turn assistant may remain acceptable on single-turn examples but accumulate enough subtle failures to become frustrating over time. A harness gives you a baseline and a comparison method, so the team can see when a gain in one dimension came with a loss in another. That is especially important because average scores can hide failures that matter to users. A system that performs well on most examples and catastrophically on a minority may still report a respectable mean.

Evidence quality is what keeps the evaluation from collapsing into opinion. A screenshot of a benchmark run is weak evidence because it is hard to reproduce, hard to audit, and easy to cherry-pick. A folder of

hand-picked “good examples” is only slightly better if nobody knows why those examples were selected. Strong evidence includes the test case definition, the observed output, the relevant context, the metric result, the threshold, and the reason for failure or pass. When a disagreement arises between product, research, and engineering, that evidence becomes the common reference point. The discussion shifts from whether someone “likes” the output to whether the system satisfied the agreed criteria.

This is why the evaluation harness matters as a control surface. It lets you preserve examples, rerun them, compare runs, and record the basis for the decision. In practice, that means the harness sits between the application and the release decision. It does not replace the model, the retriever, or the orchestration layer. It makes their behavior observable in a form that supports judgment. That distinction is easy to miss when a team first becomes productive with LLMs, because the early wins are often qualitative: a clever prompt, a useful answer, a satisfying demo. But once the system enters a product environment, the question changes from *can it impress?* to *can you defend its quality over time?*

**The trust deficit in informal testing** Ad hoc prompt trials are useful for exploration, but they do not establish reliability. They are biased toward memorable successes and against the boring failures that matter in production. If a teammate tries five prompts and the outputs look reasonable, that does not tell you much about coverage, robustness, or error rate. It tells you that the system can work in a narrow slice of the input space.

Manual spot checks suffer from the same weakness. Human review is essential for calibration, but it is expensive, inconsistent at scale, and vulnerable to context drift. Reviewers also tend to overestimate systems that produce fluent prose. Fluency creates a false sense of

correctness. The model sounds authoritative, the answer appears well structured, and the mistake is only visible if you already know the domain. That makes informal review a poor substitute for structured evaluation.

A particularly dangerous anti-pattern is to validate a system with a curated benchmark screenshot. A static image of a high score can hide benchmark overfitting, stale test data, or a configuration that no longer matches production. Another anti-pattern is changing too many variables at once. If you revise the prompt, swap the model, and modify retrieval settings in one commit, you may improve the output while destroying your ability to diagnose future regressions. The system becomes less legible even if the immediate score goes up.

The same problem appears in long-lived conversational products. A single turn may look fine, but small failures compound across a session. The assistant forgets a constraint, contradicts itself, overuses a tool, or loses the thread of the user's goal. No single demonstration captures that accumulation unless the evaluation framework is designed to expose it. That is the central reason the harness exists: it makes the hidden failure modes visible.

**From intuition to enforceable expectations** A harness becomes useful when you can express quality as an expectation with observable consequences. “Good output” is too vague unless you can say what good means for the task at hand. For a factual answer, good may mean that the response is supported by retrieved evidence. For a support agent, good may mean that the response follows policy and avoids unnecessary escalation. For a tool-using agent, good may mean that the right tool was called with the right arguments and no unsafe side effects occurred. The wording changes by system, but the logic stays the same: define the property, capture the evidence, and evaluate against a rule.

That rule needs explicit failure criteria. Without them, a team can always reframe a weak result as acceptable. The answer was “close enough.” The failure was “only in one edge case.” The issue was “not important enough to block release.” Those judgments may be correct, but they are not portable unless the team records how much deviation is tolerable and under what conditions. Explicit thresholds convert judgment into policy. They tell you when to pass, when to fail, and when to escalate to review.

This is also where the notion of risk alignment enters the picture. Not every defect deserves the same operational response. A cosmetic formatting issue should not carry the same weight as a privacy leak or a tool call that performs the wrong action. If the harness treats all failures equally, it becomes noisy. If it ignores serious failures because the average score still looks fine, it becomes dangerous. The better approach is to connect evaluation criteria to the actual product risk and the user harm that follows from it.

The strongest teams treat evaluation as a governed practice. They keep the test set stable enough to compare runs, but they also version it deliberately when the product definition changes. They record the reason for each threshold. They identify who reviews ambiguous failures. They preserve the evidence needed to understand whether a change represents improvement, drift, or a new class of bug. That practice turns evaluation into a durable organizational memory rather than a temporary experiment.

**Why the harness is an engineering control** Engineers already use control systems wherever failure is costly. They add circuit breakers, retry limits, schema validation, alarms, and rollback procedures because they know that complex systems misbehave under load. LLM systems deserve the same treatment. An evaluation harness is the quality-control counterpart to those runtime safeguards. It does not

prevent every error, but it provides a structured way to detect, classify, and respond to them before they reach users.

The harness also reduces disagreement. Product managers want evidence that the user experience improved. Researchers want a way to compare behaviors across models and prompts. Engineers want a reproducible signal that can be tied to code changes. A shared harness gives each group the same artifacts to inspect. That shared evidence matters because language model behavior is often multidimensional. One stakeholder notices correctness, another notices tone, and another notices latency or cost. Without a common evaluation layer, those perspectives stay fragmented.

The engineering benefit becomes clearer when you consider release decisions. A team that relies only on intuition tends to ship on the basis of confidence rather than evidence. The model “feels better.” The answers “look more polished.” The demo “went well.” Those are not invalid observations, but they do not scale into a release process. A harness lets you define a release gate: the system must satisfy the chosen criteria on the preserved cases, under the agreed thresholds, with the expected level of evidence. If it does not, the team has a reason to pause. If it does, the team has a reason to proceed.

That gate does not need to be binary in every case. Some failures should block deployment immediately. Others should generate warnings, track trends, or route to human review. The important point is that the decision is explicit. A harness makes that possible because it records the behavior in a form that can be compared over time. The decision moves from tribal knowledge to documented evidence.

The vocabulary that supports this process is simple but precise. *Repeatability* means you can rerun the same cases under the same rules and get comparable results. *Regression* means a later change made the system worse on an agreed measure. *Evidence* means the artifacts

that let another competent reviewer reconstruct the judgment. *Thresholds* mean the boundaries that separate pass, fail, and review. *Risk alignment* means the criteria reflect the actual consequences of failure. Once those terms are in place, the rest of the evaluation stack becomes easier to design because the team is no longer arguing about abstractions.

```
{
  "input": "Summarize the policy in one paragraph.",
  "retrieval_context": [
    "Policy v3.2, section 4: refunds require..."
  ],
  "actual_output": "Refunds are available within 30 days..."
}
```

A compact record like this looks mundane, but it captures the minimum ingredients of accountable evaluation: the user input, the evidence the model saw, and the output it produced. Once those artifacts are preserved, the team can attach metrics, compare runs, and explain failure cases without reconstructing the entire interaction from memory. That is the operational value of the harness. It turns quality from a subjective impression into an observable property of the system.

**The cost of not having one** Without a harness, product teams often optimize the wrong thing. They spend time polishing prompts that only work on favored examples. They celebrate isolated successes that do not survive broader input variation. They treat model quality as a property of the latest demo instead of a property of the system under conditions of change. The result is fragile confidence: the team feels ahead until the first serious regression arrives.

The failure is not just technical. It is organizational. If no one can point to the evidence, then every later debate about quality becomes a debate about memory, not data. If no one has preserved the test cases, then every regression has to be rediscovered manually. If no one has set thresholds, then release decisions depend on the loudest voice in

the room. A harness eliminates that ambiguity by making evaluation a routine part of the engineering process.

That is why the right mental model is not “extra testing for AI.” It is a trust infrastructure for systems whose outputs are probabilistic, context-sensitive, and expensive to judge informally. The harness gives you a place to encode expectations, preserve evidence, compare behavior, and make release decisions with more discipline than intuition alone can provide.

### 1.2. What DeepEval Actually Provides

DeepEval is easiest to understand if you treat it as a controlled evaluation surface rather than as a score generator. That distinction matters because teams often reach for the first available tool that can print a number, then discover that the number is hard to interpret, impossible to reproduce, or attached to the wrong abstraction. DeepEval gives you a place to encode cases, run them consistently, attach metrics, and preserve the evidence that supports a release decision. It does not replace the application under test, and it does not decide quality for you. It gives you the structure needed to make quality claims defensible.

A minimal harness invocation makes that role concrete:

```
deepeval test run \  
  --dataset eval_cases.jsonl \  
  --repeat 3 \  
  --cache \  
  --ignore-errors
```

That pattern already reveals the framework’s shape. You are not launching a model in isolation. You are replaying known cases, controlling repetition, and collecting results in a way that supports comparison. The value is not the command itself; the value is that the command encodes a repeatable evaluation contract. Once you have that

contract, you can ask whether a model change, prompt edit, retrieval adjustment, or agent policy revision improved behavior without relying on memory or anecdote.

**Where DeepEval sits in the tooling landscape** DeepEval is not a benchmark leaderboard. A leaderboard compares scores across systems, but it rarely tells you whether the score reflects your product’s actual risks or whether the underlying test set is still relevant to the current system. DeepEval is also not just an observability tool. Tracing tools are excellent at showing what happened in a live run, but they do not by themselves define pass/fail criteria or provide a stable replay surface for regression testing. And DeepEval is not a one-off testing script, because scripts are usually fragile, hard to version, and difficult to scale across multiple categories of behavior.

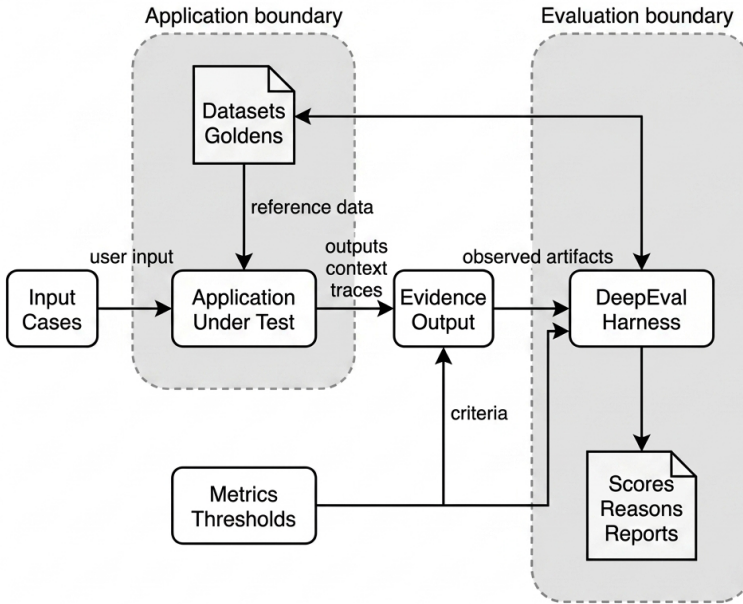
The useful comparison is with the evaluation harness pattern used in mature software engineering. You define inputs, expected properties, and acceptance criteria; you run them repeatedly; and you keep the results as evidence. DeepEval packages that pattern for LLM systems. The important word is *package*: the framework organizes the workflow, but the quality of the outcome still depends on how carefully you define the cases and metrics.

That boundary is why the harness should be understood as infrastructure for judgment rather than judgment itself. You can use it to compare two prompts, evaluate a retrieval pipeline, assess a chat agent, or check a tool-using workflow, but the framework only becomes trustworthy when the artifacts it consumes are well modeled. If you feed it noisy examples, vague criteria, or shifting datasets, the harness will faithfully produce noisy, vague, shifting results. The tooling does not compensate for weak evaluation design.

**The application boundary matters more than the metric name** A mature team usually fails on evaluation not because it lacks metrics, but because it attaches the metrics to the wrong layer. DeepEval evaluates application behavior. That behavior may be a single-turn generation, a retrieval-augmented pipeline, a multi-turn conversation, or an agent that calls tools and accumulates state across steps. The framework is useful across all of those cases, but the evidence you must capture changes by scope.

For a single-turn generator, the core artifacts are straightforward: the input, the model output, and any reference or rubric you want to compare against. For retrieval-augmented generation, you need more than the final answer. You need the retrieved context, because the answer only makes sense relative to what the model saw. For multi-turn systems, you need the conversation history, because a failure may arise from a contradiction introduced three turns earlier. For agentic workflows, you need tool traces and intermediate decisions, because the final answer can look plausible even when the agent took the wrong path.

This is the central architectural point: DeepEval is most useful when the application under test exposes the evidence required to judge behavior at the right level of abstraction. If you evaluate only the final answer of a tool-using system, you may miss unsafe tool selection, unnecessary tool calls, or hidden dependency failures. If you evaluate only the retriever in isolation, you may miss how the generator uses the retrieved material. Good evaluation starts with evidence modeling, not with a metric catalog.



The dashed boundary in the figure matters conceptually. The application boundary contains the system that generates behavior: prompts, retrieval, orchestration, tools, memory, and the model call itself. The evaluation boundary contains the machinery that interprets that behavior: datasets, metrics, thresholds, scoring logic, and reports. Confusing those boundaries leads to brittle evaluation programs. For example, if you hide retrieval evidence inside the application and never surface it to the harness, you make faithfulness hard to measure. If you bury tool traces in logs that the evaluator cannot access, you make agent safety hard to assess. The harness is only as good as the evidence

you let it see.

**What kinds of systems you can evaluate** DeepEval covers a useful range of LLM application types because it is built around the idea that tests are atomic interactions with different scopes.

- Single-turn generation is the simplest case. You provide an input and evaluate the model's response against a reference, rubric, or property such as relevance, correctness, style, or policy adherence. This is often the right starting point because it isolates the output layer. If the behavior is already unstable here, there is little point in adding orchestration complexity.
- Retrieval-augmented pipelines introduce a second layer of evidence. Now you care not only about the generated answer, but also about whether the retrieved context was relevant, sufficient, and actually used. This changes evaluation from a pure text comparison problem into a composite judgment about retrieval quality and answer quality. A model can produce a fluent answer that is not supported by the retrieved material; conversely, it can retrieve excellent context and still fail to synthesize it. The harness needs to observe both halves of the pipeline.
- Multi-turn conversational systems add state and temporal dependence. A single response can be correct in isolation but wrong relative to the conversation history. In that setting, the harness must preserve the dialogue context, because you are evaluating continuity, memory, and consistency as much as you are evaluating the latest turn. If a product claim depends on retaining constraints across turns, the evaluation case must carry those constraints forward explicitly.
- Agentic workflows with tools are the most demanding scope because they involve process as well as output. A tool-using sys-

tem may retrieve documents, call APIs, query databases, or invoke internal functions before producing a final answer. The final answer can hide a flawed decision path. If the system called the wrong tool, called it with the wrong arguments, or called it when no tool was needed, the output alone may not reveal the error. Evaluation at this scope depends on tool traces, intermediate states, and clear criteria for what counts as appropriate action.

The important point is not that all of these system types require different frameworks. The important point is that they require different evidence. DeepEval’s job is to give you a consistent place to evaluate each scope without collapsing everything into a single opaque score.

### **How DeepEval differs from observability and monitoring**

Observability tools answer a live debugging question: what happened in this run? DeepEval answers a controlled quality question: does the system satisfy this expectation across this set of cases? Those questions are related, but they are not interchangeable.

A tracing stack may show token usage, latency, model calls, retrieved documents, and tool invocations. That data is valuable, and in many systems you should use it as part of the evidence fed into evaluation. But observability alone does not tell you whether a given behavior should pass or fail. It does not freeze the cases. It does not enforce thresholds. It does not compare the current run against a baseline under repeatable conditions. DeepEval adds those controls.

The reverse confusion is also common. Teams sometimes build a script that runs a prompt, prints a response, and decides the output is “good enough.” That is not observability, and it is not evaluation infrastructure either. It is a convenience wrapper around manual judgment. A harness requires more discipline than that: stable cases, explicit criteria, repeatable execution, and recorded reasons. If the output is good,

the harness should tell you why. If it is bad, the harness should tell you what failed.

**The role of datasets, metrics, and execution flow** DeepEval organizes evaluation around a small set of operational primitives: test cases, datasets, metrics, and execution flow. Those terms are worth separating carefully.

A test case is an atomic interaction, not a vague user story. You choose the minimal slice of behavior you need to evaluate a specific risk. A dataset is the collection of those cases, usually curated to represent the product surface or a failure mode you care about. A metric is the rule or model used to assess the case. An execution flow is the mechanism that runs the cases, computes the metrics, and aggregates the results.

That structure is important because it keeps evaluation auditable. If a result changes, you can usually tell whether the change came from the application, the dataset, the metric, or the execution settings. Without that separation, results become hard to interpret. A score can rise because the model improved, because the test set became easier, or because the judge became more permissive. Only a harness with clear component boundaries can distinguish those possibilities.

This also explains why version discipline matters. DeepEval capabilities and interfaces can evolve across releases, so you should think in stable conceptual terms rather than depending on one exact method name or CLI flag. The stable concepts are the ones that matter operationally: test cases, datasets, metrics, simulations, reasons, thresholds, and comparative runs. You can validate the precise API surface against the version you use. What should not change is the evaluation logic itself.

**Metrics are instruments, not truth machines** A common mistake among experienced teams is to treat a metric as if it were a direct measurement of quality. In practice, a metric is an instrument with a scope, bias, and failure mode. Some metrics are deterministic and rule-based. Others rely on a judge model. Some are better for retrieval quality. Others are better for response quality, safety, or process correctness. DeepEval provides the structure for running these metrics, but it does not remove the need to understand what the metric can and cannot see.

That distinction matters because the same application can require different kinds of evaluation. A factual assistant may need answer correctness and source grounding. A support bot may need tone and policy compliance. A tool-using agent may need action validity and trace correctness. No single metric captures all of those concerns cleanly. DeepEval's value is that it lets you compose the right metrics around the right evidence instead of forcing everything into one score.

Consider the practical difference between a score and a reason. A score tells you whether a case passed a threshold. A reason tells you what the evaluator saw. The reason is often more useful than the number, because it lets you debug the failure. If a system is failing because it cites unsupported claims, the reason should expose that. If it is failing because the retrieval context is insufficient, the reason should reflect that. The harness earns its keep when it makes failure legible.

```
from deepeval.metrics import GEval

metric = GEval(
    name="Answer Relevance",
    criteria="Judge whether the answer addresses the input.",
    threshold=0.8,
    strict_mode=False,
    include_reason=True,
)
```

That style of metric configuration illustrates the boundary between

framework and application. You define the criterion, the pass threshold, and whether you want a reason. DeepEval executes the evaluation and records the outcome. The framework does not invent the product requirement; you do. It turns the requirement into a repeatable test.

**Why this matters for release decisions** The operational value of DeepEval is not that it produces more testing activity. Busy does not mean trustworthy. The value is that it creates a decision surface for quality. You can compare runs, inspect failures, and decide whether a change is acceptable in the context of the risk you actually care about.

That matters because LLM systems tend to improve unevenly. A model upgrade may reduce hallucinations but increase verbosity. A prompt change may tighten style but reduce retrieval fidelity. A new retriever may improve freshness but weaken ranking stability. If you only look at the latest answer, those trade-offs disappear. If you run a harness with stable cases and reasoned outputs, they become visible. That visibility is what allows engineering teams to make release decisions that are grounded in evidence rather than optimism.

DeepEval is therefore best understood as an evidence framework for LLM applications. It provides a disciplined place to encode expectations, run them consistently, and preserve the artifacts needed to defend a quality claim. The framework is useful exactly because it respects the boundary between the system that produces behavior and the harness that judges it. That boundary is where evaluation becomes reliable enough to influence production decisions.

### 1.3. Trustworthy Evaluation Principles for Experienced Teams

A busy evaluation suite is not necessarily a trustworthy one. You can accumulate dozens of tests, multiple judge prompts, and several dashboards, yet still fail to answer the only question that matters operationally: should you trust the result enough to change the product? Experienced teams know that trust comes from disciplined design, not test volume. The suite has to stay interpretable when the model changes, reproducible when the prompt shifts, and relevant when the product starts failing in ways the original demo never exposed.

A useful mental model is to treat the evaluation program as a measurement system. If the measurements drift, the thresholds are arbitrary, or the failure reasons are opaque, the resulting scores become decorative rather than decision-grade. A harness built on that foundation can still run, but it cannot justify a release gate. The practical problem, then, is not how to collect more signals. It is how to make the signals credible.

#### **Measurable expectations come first**

The first principle is measurability. If a quality expectation cannot be observed, it cannot be enforced. “The answer should be helpful” is too vague unless you can translate it into something a reviewer or metric can inspect. “The answer must address the user’s question, avoid unsupported claims, and cite relevant context” is far better because each clause creates an observable criterion. The same logic applies whether you are judging a single-turn answer, a retrieval-augmented response, or an agent’s tool-using behavior.

Measurability does not require every property to be numeric. It requires that every property be inspectable. Some expectations are best expressed as reference comparisons; others as rubric-based judg-

ments; others as structural checks on format, tool usage, or policy adherence. The important thing is that the metric or reviewer can test the expectation against actual evidence rather than against taste. If a criterion depends on a vague sense of quality, it will fail under disagreement. If it depends on evidence the harness can preserve, it becomes operational.

This is where experienced teams often overgeneralize from model fluency. A fluent answer can still be wrong, incomplete, or unsafe. A metric that rewards elegance without checking substance will systematically mislead you. If the product risk involves factuality, traceability, or correct tool use, then the expectation must name that risk directly. Do not hide behind a generic score when the real concern is specific.

#### **Stable datasets make comparisons meaningful**

The second principle is dataset stability. If the benchmark itself changes unpredictably, a score delta tells you almost nothing. You need the evaluation set to be stable enough that a later run is comparable to an earlier one. That does not mean the dataset can never evolve. It means evolution must be deliberate, versioned, and traceable. If you silently edit cases after seeing the results, you destroy the ability to tell whether the system improved or the benchmark became easier.

Stable datasets matter for the same reason controlled experiments matter in any other engineering domain. If you want to understand regression, you hold the test conditions steady. If you want to understand model drift, you preserve the reference set. If you want to know whether a new retriever is actually better, you must evaluate it on the same cases the old retriever faced. Otherwise, comparison collapses into guesswork.

This is also why a single aggregate score is a weak authority signal. Averages hide distribution shifts. A system can improve on common cases while degrading on important edge cases, and the average may

still rise. The more critical the application, the more you need a stable suite that includes representative cases, difficult cases, and failure-triggering cases. A noisy or constantly edited benchmark may still be useful for exploration, but it should not be the basis of a release decision.

Stable datasets are also easier to audit. If a team disputes a result, you can point to the exact case set and say what changed since the previous run. That traceability turns evaluation from a debate about memory into a debate about evidence. The control is not only technical; it is organizational.

### **Thresholds make decisions explicit**

The third principle is threshold explicitness. A score without a threshold is a measurement without a policy. It may describe behavior, but it does not support an action. Teams need to know what counts as pass, fail, and gray-zone review. Otherwise every release decision becomes a negotiation after the fact.

Thresholds should reflect product risk rather than statistical convenience. A lightweight formatting issue may justify a permissive threshold, while a safety-sensitive action may require a much stricter one. If the same threshold governs every failure mode, the suite becomes blunt. If thresholds are chosen after inspecting the results, they become self-serving. Set them before the run, justify them in terms of user harm and operational cost, and revise them only with the same discipline you would apply to any other production policy.

Strict thresholds reduce false confidence, but they also create reviewer burden when the task is inherently ambiguous. That is where gray-zone handling matters. Not every borderline case should automatically fail a release, and not every ambiguous result should be waved through. A good harness lets you route cases into manual review when the evidence is incomplete or the cost of a false decision is high. That design

is usually better than pretending every case is binary.

You should also be careful not to let thresholds become theater. If they are so loose that they never fail, they do not constrain anything. If they are so strict that they fail most normal variation, they lose operational value. The useful threshold is one that aligns with the product's tolerance for error and the team's ability to act on the result.

#### **Failure reasons are part of the measurement**

The fourth principle is diagnostic richness. A score tells you whether the case passed. A reason tells you why. You need both, but the reason often carries the real engineering value. When a test fails, a terse "false" is not enough to guide action. You need to know whether the failure came from unsupported claims, irrelevant retrieval, missing context, incorrect tool selection, or a policy violation. The better the reason, the faster the fix.

This matters especially when the same failure pattern can surface through multiple metrics. A RAG answer may fail faithfulness because the retriever missed key context, because the generator ignored the context, or because the context itself was insufficient. Without reasons, those cases collapse into the same score. With reasons, the team can localize the problem and assign ownership correctly. That cuts down on blame-shifting and reduces diagnosis time.

Reason-rich failures also help calibrate the evaluation suite itself. If the metric repeatedly flags cases that humans judge as acceptable, the metric may be too strict or poorly aligned. If it consistently misses failures that reviewers care about, the metric is too permissive or blind to the wrong signal. The explanation attached to the score is therefore not cosmetic. It is part of the measurement loop.

DeepEval's configuration model reflects that reality by allowing reason capture and by constraining the evaluation payload to the parameters

actually referenced by the metric criteria or evaluation steps. That restriction is easy to underestimate. It prevents you from smuggling irrelevant inputs into the judge prompt and then attributing the result to evidence the metric was never instructed to use. In practice, that is a validity safeguard: the more tightly you specify what the metric may inspect, the easier it is to defend the result.

### **Risk alignment keeps the suite honest**

The fifth principle is alignment with application-specific risk. An evaluation suite becomes untrustworthy when it measures the wrong thing very well. Eloquence is not safety. Fluency is not correctness. High agreement with a judge model is not the same as low product risk. You need the suite to reflect the consequence of failure in the actual application.

That means the suite should be shaped by the question “What can go wrong in production, and how would we know?” If the product involves tool use, you need criteria for whether the right tool was called at the right time. If the product surfaces factual content, you need criteria for grounding and unsupported claims. If the product includes long conversations, you need criteria for continuity and state retention. If the product handles regulated or sensitive tasks, you need criteria that reflect those constraints directly. A general score may still be useful, but it should not be the only thing you trust.

Risk alignment also determines what deserves automation. Some risks should fail automatically; others should only raise a warning or route to human review. A policy-violating tool call may warrant an immediate block. A tone issue may warrant review. A mild drop in retrieval recall may warrant monitoring unless it affects a high-value path. The point is to match the evaluation response to the severity and reversibility of the failure.

This is where a sophisticated team resists the temptation to optimize

for elegance alone. A beautifully neat benchmark can still be operationally irrelevant if it misses the real failure modes. The credible suite is the one that measures the failures the product actually cannot afford.

### **Calibration keeps judgment from becoming dogma**

The fifth principle only works if you calibrate the evaluation suite continuously. Calibration is the process of checking whether the metric, the dataset, and the thresholds still reflect reality. You do that by comparing machine judgments with selective human review, especially on edge cases, disagreements, and cases close to the threshold. If the suite is never checked against human judgment, it can drift into false authority.

A practical calibration loop usually starts with curated goldens: a small but high-quality set of examples that encode what good looks like for a particular behavior. You then run the suite repeatedly, inspect disagreements, adjust the rubric or metric where needed, and freeze the rule set long enough to compare later runs. That frozen period is important. If you keep changing the rules, you cannot tell whether the system changed or the evaluator changed.

Repeated runs are also useful because LLM systems are stochastic. A single pass can understate variance or overstate confidence. Repetition gives you a better sense of stability, especially when the metric itself uses an LLM as a judge. If the score swings widely from run to run, the issue may not be the application at all. It may be the evaluator. That is why experienced teams treat the judge as an instrument to be calibrated, not an oracle to be obeyed.

```
metric = GEval(  
    name="Grounded Answer",  
    criteria="Judge only facts supported by context.",  
    threshold=0.8,  
    strict_mode=True,  
    include_reason=True,  
)
```

A configuration like this is only useful if you can also explain why the threshold is 0.8, what “strict\_mode” means operationally, and which evidence the judge is allowed to inspect. If you cannot justify those choices, the metric is just a shell around preference. If you can justify them and verify them against curated examples, the metric becomes part of an evidence chain.

### **Evaluation parameters must match the rubric**

A subtle but important validity check is that the evaluation inputs must match the rubric. If the criteria mention only answer relevance and context use, do not feed the metric a long list of unrelated fields and then assume the extra data improved the judgment. When you overpopulate the evaluation payload, you risk creating hidden dependencies that are hard to debug and impossible to explain. The cleaner practice is to pass only the fields the rubric or evaluation steps explicitly require.

That discipline also improves reproducibility. If you know exactly which parameters the judge may inspect, you can compare runs more confidently and isolate the source of change. If the metric is sensitive to undocumented context, then the score becomes difficult to interpret. In a production setting, that is not a small flaw. It is a validity problem.

This rule applies equally to human review. Reviewers should not be asked to inspect undefined context or make judgments on criteria that were never written down. If the team cannot state the rubric clearly enough for a human reviewer, the machine evaluator will not be stable either. Good evaluation design starts with a criterion that can survive translation between human and machine review.

### **Trade-offs are not defects; they are design constraints**

Every credible evaluation suite has to balance competing goals. Broad coverage competes with stability. Strict thresholds compete with re-

viewer burden. Generic metrics compete with domain specificity. Sensitive judges catch more failures, but they also create more false alarms. These are not signs that the suite is failing. They are the normal constraints of evaluation design.

You need a policy for handling those trade-offs. For high-risk failures, you can tolerate more reviewer effort and more false positives. For low-risk or noisy dimensions, you may accept looser thresholds and track only trends. For domains where human judgment remains essential, you can use the harness to narrow the cases that need review instead of trying to automate everything. The mistake is not having trade-offs. The mistake is pretending they do not exist.

One useful operating pattern is to separate *release gates* from *diagnostic metrics*. Release gates protect users from unacceptable changes. Diagnostic metrics help the team understand behavior and guide improvements without necessarily blocking deployment. Not every signal should stop a release, but every important signal should be visible. That distinction reduces operational friction while preserving accountability.

#### **A repeatable evaluation flow**

A disciplined team can evaluate a new risk or product change with a simple sequence.

First, define the failure hypothesis in user terms. Say exactly what can go wrong and why it matters. Second, encode that hypothesis as a measurable expectation using the smallest evidence set that can support the judgment. Third, select or design the metric that can inspect that evidence. Fourth, choose a provisional threshold and state the reason for it. Fifth, run the suite against a stable dataset and inspect disagreements, especially on edge cases. Sixth, calibrate the rubric, the threshold, or the test set where the evidence shows a mismatch. Seventh, freeze the rule set long enough to compare future runs against it.

That workflow sounds straightforward because it is. The difficulty lies in resisting shortcuts. If you skip the calibration step, the metric may drift. If you skip the stable dataset, comparisons become meaningless. If you skip human review entirely, you may never notice systematic blind spots. If you skip the threshold rationale, you cannot defend the release decision. The sequence is less about bureaucracy than about preserving interpretability.

The strongest evaluation programs are not the ones with the most instrumentation. They are the ones that preserve enough discipline for the signals to stay meaningful as the model, prompt, retriever, or tool chain changes. That discipline gives the suite authority. Without it, the evaluation may still run, but the team will eventually stop trusting what it says.

#### **1.4. From Product Risks to Evaluation Requirements**

The most effective evaluation programs do not begin with metrics. They begin with failure modes. If you start from a generic quality goal such as “make the assistant better,” the resulting suite usually becomes a collection of weak proxies: a few output comparisons, a few subjective judgments, and a lot of uncertainty about what a score actually means. If you start instead from a concrete product risk, you can determine exactly what evidence the harness must capture, which metric class can observe it, and what operational response a failure should trigger. That translation step is where evaluation becomes an engineering control.

A compact way to express the mapping is to encode the risk at the smallest useful scope:

```
case = {  
  "risk": "hallucinated answer in support flow",
```

```
"evidence": {  
  "input": "...",  
  "retrieval_context": [...],  
  "expected_output": "...",  
},  
"metric": "groundedness",  
"threshold": 0.85,  
"on_fail": "manual_review",  
}
```

The snippet is intentionally plain. It forces you to name the risk, identify the evidence, and declare the operational policy in the same artifact. That discipline prevents the common failure mode where a team claims to care about safety, factuality, or tool correctness but never writes down the observable criterion that would prove the system satisfied that concern.

### **Start with the risk, not with the metric**

A product risk is a statement about harm or degradation in a real use case. “The assistant may invent refund policy” is a risk. “The system is not accurate enough” is not yet a risk; it is an aspiration. To become actionable, the statement has to include the affected user, the failure consequence, and the component where the failure is likely to appear. That decomposition gives you a path from product language to testable behavior.

NIST’s generative AI risk vocabulary is useful here because it names the classes of failures that actually matter in deployment: confabulation, harmful bias, privacy leakage, information integrity failures, and IP-related risks. Those categories are not metrics. They are prompts for requirement design. If your product touches customer support, knowledge retrieval, or external tool use, those risks map naturally to evaluation requirements. For example, confabulation in a knowledge assistant is not just “bad text”; it is unsupported factual assertion in a context where the user may rely on the answer. That distinction tells you that the relevant evidence is the input, the retrieved or supplied

context, and the final answer, not just the final answer alone.

The practical workflow is consistent across domains. First, define the risk in user and system terms. Second, localize the component most likely to cause it. Third, define the atomic interaction that exposes the behavior. Fourth, choose the metric inputs required to judge that interaction. Fifth, set thresholds and escalation rules. Sixth, assign ownership for what happens when the case fails. Once you do this repeatedly, the suite stops being a loose collection of tests and becomes a map from business concern to control surface.

### **Localize the component before you define the case**

Experienced teams often make the mistake of writing a test before they know what part of the system they want to observe. That produces overbroad cases that are hard to debug and easy to argue about. If the risk lives in retrieval, the test case must expose retrieval context. If the risk lives in tool selection, the test case must expose tool calls. If the risk lives in answer generation, the test case needs the user input and the final output, and possibly a reference answer or rubric. The stronger the component boundary, the cleaner the evaluation.

DeepEval’s test-case design reflects this requirement. The schema is useful precisely because it lets you pass only the minimum data needed for the relevant risk. For a RAG system, `retrieval_context` is essential because answer quality depends on the evidence the model saw. For an agentic workflow, `tools_called` or equivalent tool traces are essential because the final response may hide a poor action path. For correctness or policy checks, `expected_output` or rubric criteria are the right anchor because they let you compare the model output to an explicit expectation. That is a tighter design than dumping every artifact into every case and hoping the metric finds the right signal.

This is also where many teams discover that the application boundary matters more than the metric family. A metric can only evaluate

what the application exposes. If the retriever is opaque, you cannot meaningfully judge whether the answer was grounded. If the agent hides tool traces, you cannot determine whether it behaved safely or efficiently. Good requirements therefore include evidence-modeling decisions, not just metric selection.

### **Use the smallest atomic interaction that can prove the risk**

An atomic interaction is the smallest evaluable unit that can support a decision. For a single-turn generation task, one prompt-response pair may be enough. For retrieval, the atomic unit may be the prompt, retrieved passages, and answer. For a multi-turn assistant, the atomic unit may be a whole turn window rather than a single response. For an agent, the atomic unit may include intermediate actions and tool calls. The goal is not to make the case as small as possible in an abstract sense. The goal is to make it as small as possible while still preserving the causal structure of the failure.

This discipline helps you avoid noisy overreach. If you evaluate a support agent only on the final answer, you may miss the fact that it performed an unnecessary account lookup. If you evaluate an entire conversation when the failure occurred on one turn, you may mask the root cause in context. If you evaluate a retrieval system at the answer layer only, you may confuse generator mistakes with retriever mistakes. Atomicity is what keeps evaluation local and actionable.

LangSmith's guidance to start with a small curated set of examples per component is compatible with this principle. Five to ten carefully chosen examples that show what good looks like for a critical behavior often teach you more than fifty loosely curated prompts. The reason is not that small datasets are sufficient forever. The reason is that they force clarity. You learn what evidence actually matters, what variation is acceptable, and which edge cases need to be encoded later as the suite matures.

### **Choose the metric class that matches the evidence**

Once the risk and atomic interaction are clear, the next step is metric class. The class matters because not every risk is best evaluated the same way. Reference-based comparison works well when the task has a clear expected output or a close canonical answer. Referenceless judgment is more appropriate when quality depends on rubric interpretation, policy adherence, or nuanced synthesis. Retrieval metrics focus on whether the right evidence was retrieved, while conversation metrics focus on continuity, state retention, and turn-level coherence. Process-oriented agent metrics inspect the action path, not just the final result.

This distinction is practical, not academic. If the risk is factual hallucination in a knowledge task, answer correctness alone may be insufficient; you may need a groundedness or faithfulness-style metric that checks whether the output is supported by the provided context. If the risk is unsafe tool use, the metric should inspect action sequence and parameters, not just the textual answer. If the risk is format compliance, a deterministic structural metric may be better than a judge model. When the evidence is objective and bounded, a simpler metric is often preferable. When the evidence is open-ended or qualitative, a rubric-based judge may be necessary.

DeepEval’s metric design makes this separation explicit. A metric can be deterministic or LLM-backed, and that choice should follow the nature of the evidence. Deterministic checks are cheaper, faster, and easier to reason about when the property is structural or rule-bound. LLM-backed judges are more flexible when the property requires semantic interpretation. The mature approach is not to prefer one universally. It is to choose the smallest metric class that can observe the risk reliably enough for the operating context.

### **Turn thresholds into release policy**

A metric without a threshold does not guide action. The threshold is the boundary between observation and decision. If you never specify it, you cannot tell whether a failing score should block release, trigger review, or simply inform monitoring. For product risks, that distinction matters as much as the metric itself.

Thresholds should reflect severity, reversibility, and confidence. A privacy leak or unsafe tool action should usually have a strict threshold and an automatic fail path. A minor tone deviation or a borderline relevance issue may justify a review path instead. A soft drop in retrieval quality may be acceptable if the product can absorb variance, but not if the degraded behavior affects a critical workflow. In other words, thresholds are policy, not decoration. Set them according to consequence, not convenience.

It helps to separate three operational responses:

- **Automatic fail:** the risk is high enough that the case blocks release.
- **Manual review:** the evidence is ambiguous or the consequence is severe enough to warrant a human decision.
- **Trend monitoring:** the case does not block release, but you track it over time for drift.

That triage keeps the suite from becoming either too lax or too noisy. If every failure blocks release, reviewers drown. If no failure blocks release, the suite loses authority. The strongest teams attach each risk to one of these paths before they run the first evaluation.

### **Treat escalation and ownership as part of the requirement**

A requirement is incomplete until you specify who responds to failure. If a retrieval test fails, does the owner belong to search, ranking, prompt design, or the product team? If a tool-call test fails, does the

issue belong to orchestration, policy, or application code? If a conversational continuity test fails, does the fix belong to memory management or dialogue design? Ownership boundaries matter because they determine whether failures become action items or just dashboard clutter.

Escalation policy should be written alongside the threshold. A strong requirement says not only “fail if unsupported claims exceed the threshold,” but also “route failures to the model quality owner and require sign-off before launch.” That makes the suite operational. It links the observed behavior to the team that can change it. Without ownership, the harness becomes a reporting layer. With ownership, it becomes part of the release process.

This is especially important for cross-functional systems. A RAG pipeline may involve retrieval engineering, prompt design, application logic, and content policy. An agent may span orchestration, tool integration, and safety controls. If you do not define the ownership boundary, every failure becomes a debate about jurisdiction. Evaluation should reduce that ambiguity, not amplify it.

### **Use curated goldens to anchor what good looks like**

A good requirement maps risk to evidence, but a good suite also needs examples of success. Curated goldens are the easiest way to express that. A golden is not just a favorite sample. It is a case that encodes the behavior you expect under a known condition. If the system is supposed to ground answers in a specific policy passage, the golden should include that passage and an answer that uses it correctly. If the system is supposed to avoid a certain tool call, the golden should show the correct refusal or alternative action.

Goldens do two things. They define the positive target, and they give the team a stable reference for calibration. When the model changes, you can rerun the goldens and see whether the same behavior still holds. When the judge changes, you can check whether its interpre-

tation still matches human expectation. When the product changes, you can update the goldens deliberately rather than letting them drift informally.

Because goldens are expensive to produce well, they should be concentrated on high-value risks and critical paths. You do not need a golden for every possible input. You need enough carefully chosen examples to anchor the behavior you care about most. That is why small, curated sets are so effective early on: they reveal the shape of the problem before you scale the suite.

### **Calibrate with repeated runs and selective human review**

A trustworthy requirement is one that survives calibration. Repeated runs help you distinguish true regressions from stochastic variance. Human review helps you catch blind spots in the metric or judge prompt. Together, they keep the suite honest.

Repeated execution is especially important when the evaluation uses an LLM judge. A judge model may be useful, but it is still an instrument with variance. If you run the same case several times and the outcome changes materially, you have learned something important about the evaluator itself. That does not invalidate the harness. It tells you where to tighten the rubric, narrow the evidence, or switch to a more deterministic check.

Selective human review is the other half of calibration. You do not need humans on every case forever. You need them on the edge cases, the disagreements, and the cases near the threshold. That review loop lets you refine the requirement until the machine judgment and the human judgment converge enough to support release decisions. The goal is not perfect agreement. The goal is adequate and documented agreement for the risk level involved.

This is where versioned history becomes valuable. If you record the

TEVV history of a metric, a dataset, and a threshold, you can explain why a change in score occurred and whether it reflects product behavior or evaluation behavior. That history is part of the evidence chain. It makes the suite legible after the team, the model, or the product has changed.

### **Include execution conventions in the requirement**

Operational requirements are not only about metrics. They also include execution conventions such as repeats, caching, and error handling. If the system is stochastic, you may need repeated runs to estimate variance or surface unstable behavior. If the evaluation is expensive, you may use caching to keep iteration practical. If some parameters are missing, you need a clear policy for whether the case fails, skips, or warns. If an evaluation error occurs, you need to know whether the harness should stop, continue, or mark the case as invalid.

These choices affect trust as much as the metric does. A suite that silently ignores missing parameters can give a false sense of coverage. A suite that suppresses errors without reporting them can create phantom passes. A suite that reuses stale cached results without making the cache visible can hide behavior changes. Mature teams therefore write execution conventions into the requirement set. They treat reproducibility as part of the control plane, not as an implementation detail.

This is also why requirement design should start from stable failure concepts rather than from a single API surface. DeepEval's interfaces may evolve, but the requirement should still say what evidence the case needs, how many times it should run, how missing data should be handled, and what happens when evaluation fails operationally. Those are durable design decisions. The exact syntax can follow the version you are using.

### **Translate the workflow into a release gate only when the risk justifies it**

Not every requirement should block a release. That is a temptation worth resisting. Some failures are too noisy, too low-risk, or too expensive to gate automatically. Others are severe enough that the harness must act as a hard control. The decision depends on user harm, reversibility, failure frequency, false-positive cost, and false-negative cost. If the harm is high and the evidence is reliable, make the requirement a gate. If the signal is useful but noisy, make it diagnostic or advisory. If the risk is still poorly understood, use the harness to collect evidence before assigning a stronger policy.

That calibration keeps the program credible. Teams trust evaluation suites that are selective, not ones that block every change. A gate should represent a meaningful constraint. If it blocks trivial variation, it trains people to work around it. If it never blocks anything serious, it becomes ornamental. The right requirement is one that reflects real deployment risk and can be operated sustainably.

The end result is a requirement chain that runs from business risk to observable evidence, from evidence to metric class, from metric to threshold, and from threshold to ownership. Once you can write that chain clearly, you are no longer discussing quality in the abstract. You are specifying a control system that can survive model changes, product churn, and the normal friction of production work.



## Chapter 2

# Modeling Evidence with Test Cases, Datasets, and Goldens

*A strong evaluation program does not begin with metrics; it begins with evidence modeled well enough that metrics can mean something stable over time. This chapter teaches readers how to encode LLM behavior into test cases and benchmark assets that survive framework changes, expose real product risk, and let future chapters discuss scoring, RAG diagnosis, and workflow automation on solid ground.*

### **2.1. LLMTestCase as the Atomic Unit of Evaluation**

Two teams can run the same model against the same prompt and still reach opposite conclusions because they did not preserve the same evi-

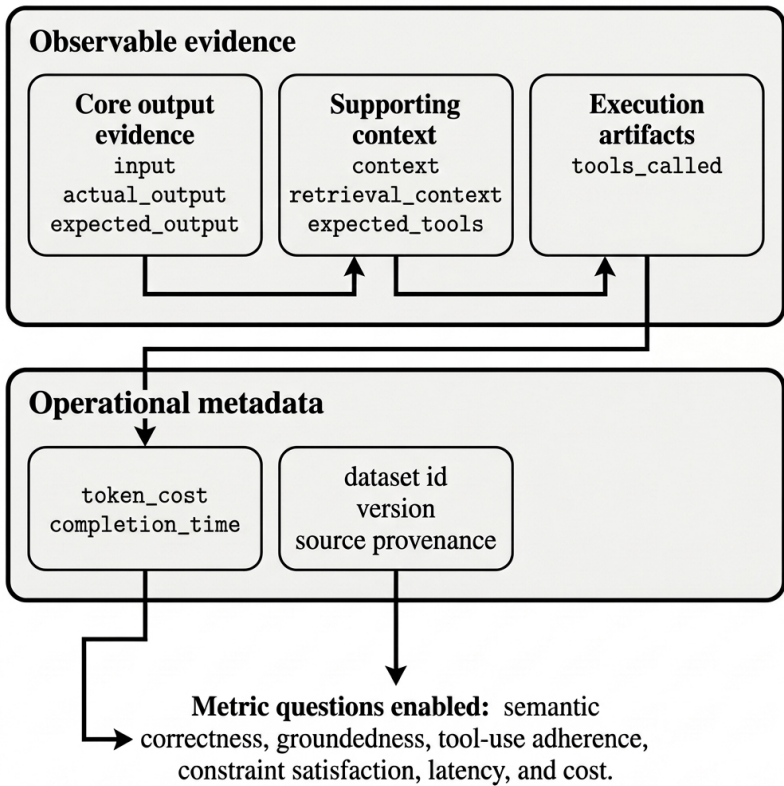
dence. One team keeps only the input and final answer; another keeps the expected behavior, retrieval artifacts, and the cost and latency context that explain why the answer matters. The score may be identical, yet the judgment is not. That difference is not a reporting nuisance; it is a failure of the evidentiary record. In DeepEval, `LLMTestCase` is the record you evaluate, not a convenience wrapper you pass to a metric. If the record is incomplete, the metric can still return a number, but the number no longer tells you what you think it tells you.

**A concrete test case establishes the evidentiary contract.** A minimal single-turn record carries three statements at once: what the application received, what it produced, and what behavior you expected. In practice, you encode that directly in the test object:

```
from deepeval.test_case import LLMTestCase

case = LLMTestCase(
    input="Summarize the refund policy in one sentence.",
    actual_output="Refunds are available within 30 days.",
    expected_output="Refunds are available within 30 days.",
    context=[
        "Policy excerpt: refunds are available within 30 days "
        "of purchase with receipt."
    ],
    token_cost=0.0184,
    completion_time=1.27,
)
```

That small object already fixes the boundary of interpretation. `input` records the user-facing request. `actual_output` records the model behavior under test. `expected_output` records the reference behavior the evaluator is allowed to compare against. Everything else is supporting evidence or operational metadata. Once you separate those roles, you can ask sharper questions: is the answer semantically right, is it grounded in supplied context, does it satisfy a structured constraint, did it do so within a latency budget, and did it do so at an acceptable cost?



**The atomic unit is a contract, not a transcript.** Once you treat the test case as a contract, each field gains a specific semantic boundary. `input` is the stimulus. It should represent the user request as received by the application layer, not a cleaned-up prompt reconstructed after the fact. If the application performs hidden prompt assembly, keep the user-facing portion distinct from internal scaffolding so you can localize failures correctly. A metric comparing `actual_output` to `expected_output` cannot diagnose a broken upstream prompt template if you have folded that template into the user text and erased the boundary.

`actual_output` is the observed behavior, and that sounds trivial until normalization enters the picture. You often need to decide whether to preserve formatting exactly, strip harmless whitespace, canonicalize JSON, or retain raw text for safety and hallucination checks. Make that decision explicit. A test case that silently normalizes away punctuation can hide regressions in code generation or structured extraction. A test case that preserves every formatting artifact can generate noise for semantic metrics. The field should therefore retain the output at the granularity required by the strongest intended metric, while normalization rules live beside it rather than inside it.

`expected_output` is not a rubric, and it is not a product wish list. It is the reference behavior the harness can compare against. If you stuff multiple meanings into this field, the evaluation becomes ambiguous very quickly. An expected answer that mixes the target content, grading instructions, and exception handling guidance forces the metric to guess which part matters. That is how teams get false confidence: the model appears to satisfy a bundled expectation even though no single requirement is actually met cleanly. Keep the reference behavior separable from the grading logic. If the correct answer is open-ended, encode the expected behavior as structured constraints or as a richer context object, not as a prose paragraph that tries to do every job at once.

**Supporting context tells the metric what reality the answer lived in.** The core triad is not enough for many applications. You often need `context` to preserve user-visible state that materially changes the meaning of the answer. In a support workflow, that may include the account tier, a policy excerpt, or a prior confirmation message. In a product workflow, it may include a selected plan, an address, or an appointment slot. Keep `context` limited to the facts that were actually available to the application at decision time. If you mix hidden retrieval content into the same field as user-visible context, you remove the abil-

ity to determine whether the model used the right evidence or merely benefitted from a stronger test fixture.

`retrieval_context` serves a narrower purpose. It preserves what a retriever supplied, often with source texts or passages that support a groundedness metric later on. That field matters because retrieval quality and answer quality are separable failure modes. A good answer can arise from poor retrieval by luck, and a bad answer can arise from good retrieval because the generator ignored the evidence. Without the retrieval payload, you cannot localize the failure. With it, you can distinguish a missing-document problem from a reasoning problem. That distinction is exactly why the record should include provenance for any externally supplied context.

`expected_tools` belongs to agentic workflows even when the final task is single-turn. It specifies which tool interactions should have occurred, or at least which tools were permitted or intended. If you are evaluating a support agent that should fetch account status before answering, the test case should make that requirement explicit. Otherwise, the metric may reward a fluent but unauthorized answer. The same logic applies to `tools_called`, which captures what actually happened. Together, the pair creates a simple but powerful diagnostic split: did the agent choose the right action, and did it execute the right action correctly? When those fields are absent, you collapse a workflow bug into a text-only judgment and lose the ability to pinpoint the fault.

**Operational metadata turns semantic evaluation into release engineering.** Semantic correctness is not the only axis that matters in production. `token_cost` and `completion_time` convert a qualitative test into an operational one. A model that is correct but too expensive to ship, or fast but consistently within a latency budget only because it truncates output, still fails product requirements. Keeping those values in the test case lets you evaluate performance as part of the same evidence object rather than as an external dashboard disconnected from

the content judgment. That matters when a release decision depends on a trade-off between answer quality and system efficiency.

Use these fields carefully. Cost and latency are observational facts, not labels of correctness. They should not be used to explain semantic failures unless the system constraint itself is part of the test. A short response is not automatically better because it is cheaper, and a long response is not automatically worse because it takes longer. The evidence object should record the resource usage so the evaluator can compare it to budget thresholds or regressions, but the metric must still interpret that data in the context of the product requirement.

Versioning and provenance belong in the same mental category. Even when the framework exposes them through dataset abstractions rather than the raw test case, your test asset should still carry identity and origin. A case without provenance is difficult to review, difficult to retire, and impossible to compare across time. If the underlying document changed, if a prompt template changed, or if a normalization rule changed, the score may be numerically comparable while the evidence is not. Stable IDs and version tags keep the object from being mistaken for a timeless truth.

**Field discipline prevents misleading metrics.** Most evaluation failures start before the metric runs. A common anti-pattern is to encode evaluation instructions into `expected_output`. That makes the test case look richer than it is, but it also contaminates the reference with rubric language. Another is to flatten the entire execution trace into one free-form text field and call it context. That destroys the distinction between what the application knew, what the retriever returned, what the tool executed, and what the user actually saw. Once those layers collapse, a metric cannot tell whether the model failed to answer, the retriever failed to supply evidence, or the tool layer failed to execute the action.

Another failure mode is under-specification. If the test case contains only a prompt and a final answer, you can still run a similarity check, but you cannot localize many important classes of failure. You cannot tell whether the model should have called a tool, whether it answered from stale context, whether it used unsupported reasoning, or whether it ignored a specific constraint. The score may still pass. The harness may still look healthy. Yet the evaluation tells you very little about the product behavior that actually matters.

You also need discipline around normalization. If the application emits JSON, decide whether the test case stores canonicalized JSON or the raw serialized string. If the application emits code, decide whether insignificant whitespace should be normalized or preserved. If the application emits citations, decide whether the evaluation should see citation text as content or as supporting metadata. Each choice changes what a metric is allowed to inspect. The safest pattern is to normalize only the properties that you explicitly do not want to evaluate and to keep the rest intact. Hidden normalization is one of the easiest ways to create false confidence because it makes failing outputs look closer to the reference than they really are.

**The object should support several metric families without being rewritten.** A good `LLMTestCase` is durable because it supports multiple future questions. A semantic similarity metric may inspect `input`, `actual_output`, and `expected_output`. A groundedness metric may add `context` and `retrieval_context`. A tool-use metric may require `expected_tools` and `tools_called`. A performance gate may inspect `token_cost` and `completion_time`. The same record should carry enough evidence for each of those metric families without being reshaped every time your evaluation strategy changes.

That is the practical reason to keep the object atomic. If the evidence layer is stable, you can swap metrics, refine thresholds, and add new failure analyses without rebuilding the dataset. If the evidence layer

is brittle, each new metric forces a schema migration or a rewrite of old records. The harness then becomes tied to one evaluation philosophy instead of one product reality. Mature teams avoid that trap by preserving a record that is broad enough to support later judgment but narrow enough to avoid pre-deciding the conclusion.

**Author the record from the behavior you actually need to judge.** A useful workflow starts with the product question, not the available logs. First identify the behavior under test. Then decide which artifacts a later evaluator must see to judge that behavior honestly. Separate user stimulus from internal evidence. Preserve any reference material that grounds the expected behavior. Attach tool and performance metadata only if the product requirement depends on them. Finally, check that each field has a plausible consumer in the harness. If no metric or reviewer will ever inspect a field, leave it out.

That discipline also clarifies when a single-turn case is the wrong abstraction. If success depends on an evolving dialogue, a correction, or a memory trace across turns, the single-turn record will be too coarse. But when the behavior under test is one decision, one answer, or one tool-mediated action, the atomic test case is exactly the right size. It captures the evidence needed for reproducible judgment without pretending that a longer interaction has already been modeled. That restraint is what keeps the evaluation honest.

## **2.2. ConversationalTestCase and Turn Semantics for Multi-Turn Evidence**

A conversational system can look competent at each individual reply and still fail the user as a whole. It answers the first question correctly, forgets the constraint introduced on the second turn, and then repairs itself only after the user has already lost trust. That failure mode is

invisible if you treat a conversation as a pile of independent prompts. The evidence record has to preserve temporal structure, because the meaning of each response depends on what came before it. In DeepEval, `ConversationalTestCase` exists for exactly that reason: it models the interaction as ordered turns rather than as a flattened transcript, so later evaluation can inspect memory, correction, and recovery without guessing.

**A working conversational record starts with ordered turns.**

The core abstraction is simple: you represent the dialogue as a sequence of `Turn` objects, and each turn preserves at minimum the speaker role and the message content. A minimal construction looks like this:

```
from deepeval.test_case import ConversationalTestCase, Turn

case = ConversationalTestCase(
    turns=[
        Turn(role="user", content="Book a 3pm dentist appointment."),
        Turn(role="assistant", content="Sure, what date?"),
        Turn(role="user", content="Next Tuesday."),
        Turn(role="assistant", content="Booked for next Tuesday at 3
pm.")
    ],
    expected_outcome="An appointment is scheduled for next Tuesday at
3pm.",
)
```

That object is more than a transcript container. It preserves the order in which commitments were made, clarifications were requested, and constraints were introduced. Once you have that order, you can ask whether the assistant stayed aligned with the evolving task. Without it, a metric can still compare the final response to a reference, but it cannot tell whether the assistant was consistently wrong, temporarily confused, or ultimately successful after repair.

**Turn order is part of the meaning, not just the sequence.** In a conversation, the same sentence can mean different things depending on when it appears. A user saying “next Tuesday” only becomes resolvable because a prior turn established the task. A model saying “I already booked it” only makes sense if a previous action actually occurred. A correction such as “no, I meant Wednesday” is not a new request unrelated to the old one; it is an update that changes the state of the interaction. That is why an ordered turn list is the canonical representation. Reordering turns destroys causality, and collapsing them into one prompt destroys the distinction between prompt construction and conversational behavior.

The practical implication is that you should treat each turn as a state transition. The conversation begins with incomplete information, accumulates constraints, and then either resolves them or fails to do so. A single bad turn may be tolerable if the system repairs itself. A locally acceptable final turn may still be unacceptable if it contradicts earlier commitments. Multi-turn evidence therefore evaluates a trajectory, not a snapshot. This trajectory view is what lets you distinguish a graceful recovery from a brittle one.

**Role attribution constrains what the evaluator may infer.** A robust conversation record does not merely store text; it stores who said what. `role` preserves the semantic distinction between user, assistant, system, and, when relevant, tool. That distinction matters because the same content can have different evidentiary weight depending on the speaker. User turns establish intent. Assistant turns establish behavior. System turns establish policy and hidden instruction boundaries. Tool turns establish external facts or actions. If you flatten roles into a single text stream, you erase the source of each statement and make it impossible to diagnose instruction-following failures.

Role modeling becomes especially important when hidden system in-

structions affect the interpretation of a response. If the assistant disobeys a system constraint, the failure is not captured by the user-facing transcript alone. Conversely, if a user request conflicts with a system policy, the assistant may be correct to refuse. The test record has to preserve enough structure for the evaluator to tell the difference. That is one reason conversation data should remain typed rather than free-form whenever you can manage it.

**Conversation evidence must preserve memory, repair, and intermediate state.** Many failures only appear after the assistant has already committed to a path. A calendar assistant can answer the final turn correctly and still forget that the user excluded weekends. A support bot can handle an initial billing question and then lose the thread after a clarification. A booking assistant can recover after a typo, but only if it preserves the corrected details across turns. These are not edge cases of transcript formatting; they are the central semantics of multi-turn interaction.

You therefore need more than a turn list when the application depends on state carryover. DeepEval’s conversational model allows additional evidence to be attached where the turn structure alone is insufficient, such as tool-call information or retrieval-related context at the turn level. That extra structure helps you localize whether the failure arose from memory loss, retrieval failure, tool misuse, or a malformed user correction. If you omit it, a metric may still pronounce the conversation acceptable, but you will not know why the assistant succeeded or failed.

A good rule is to preserve any intermediate state that a later turn could depend on. If the assistant learned a delivery address on turn two, and the final answer depends on that address, the address belongs in the record. If the assistant invoked a tool to fetch account status, the tool result belongs in the record. If the user corrected a prior request,

the correction should remain visible rather than being rewritten into a cleaned summary. That is the difference between a faithful conversation record and an editorial reconstruction.

**Turn-level acceptability is not the same as trajectory-level success.** A conversation can contain one weak reply and still succeed. Suppose the assistant initially misunderstands an abbreviation, asks a clarification, and then gets the final answer right after the user repairs the ambiguity. That conversation may deserve a pass if the system recovers quickly and correctly. The inverse is also common: each individual response looks plausible, but the assistant gradually drifts away from the user’s actual goal. If you evaluate only the final turn, you miss the drift. If you evaluate each turn in isolation, you miss the trajectory.

That distinction is why conversation-level evidence should carry an expected outcome for the whole interaction, not just a reference answer for the last assistant turn. The expected outcome describes the successful end state: a resolved booking, a completed recommendation, a confirmed cancellation, or a clearly appropriate refusal. It does not need to over-specify every utterance. In many cases, the exact wording of intermediate turns should remain flexible because the user experience depends on whether the assistant converged, not on whether it reproduced a reference transcript verbatim.

This is also where you should resist the temptation to reduce the conversation to a concatenated prompt. Concatenation makes one-turn metrics easier to apply, but it changes the object under evaluation. The metric no longer sees distinct turn boundaries, so it cannot determine whether the assistant responded appropriately to a clarification or simply matched a long text blob. Once that happens, you lose the ability to distinguish local fluency from conversational correctness.

**A well-formed conversational case keeps the benchmark intent separate from the execution trace.** The benchmark intent belongs at the scenario level: what kind of interaction should happen, what outcome counts as success, and what constraints the assistant must respect. The execution trace is the concrete conversation that occurred in one run. Keeping those apart prevents the benchmark from becoming brittle when the model or prompt changes. DeepEval’s conversational abstractions support that separation by allowing a scenario-oriented golden or simulation input to instantiate an actual conversation record.

In practice, that means you should describe the task in terms of user intent and expected outcome, then preserve the generated or replayed turns as evidence. For example, a scheduling benchmark may say that the user wants an afternoon appointment after a certain date and that the assistant should confirm a slot without violating the stated constraint. The actual turn sequence can vary while still satisfying the benchmark intent. That flexibility is crucial if you want the suite to remain useful across model upgrades. A benchmark that demands one exact transcript is too fragile to survive even small wording changes.

**Granularity choice determines what failures you can localize.** Not every interaction belongs in a conversational case. Some tasks are genuinely single-turn: translate this sentence, classify this text, summarize this paragraph. For those, forcing a multi-turn structure adds noise without adding evidence. Conversely, any task that depends on memory, repair, or path-dependent reasoning belongs in a conversational case because the relevant behavior only exists across turns. The decision signal is not the number of messages; it is whether earlier turns materially affect the meaning of later ones.

You should model the interaction as one conversation case when at least one of these conditions holds: the assistant must remember a

constraint across turns, the user can correct or refine the task, recovery matters after ambiguity, or the business risk is attached to the full interaction rather than the last reply. If none of those conditions apply, linked single-turn cases may be cleaner. The wrong granularity hides the wrong failures. A conversation split into isolated prompts will miss memory decay. A set of single-turn tests merged into one long dialogue may obscure which step actually broke.

**Anti-patterns usually come from flattening structure too aggressively.** The most common mistake is to collapse every turn into a single prompt string and then score the output against one expected response. That makes the harness look simple, but it destroys the semantics of turn order and role attribution. Another mistake is to ignore user edits and corrections because they complicate the transcript. Once you remove the correction, the assistant appears more stable than it really was. A third mistake is to score only the final assistant message. That rewards clean endings even when the interaction was confusing, circular, or unsafe along the way.

A subtler anti-pattern is to omit explicit dissatisfaction signals. If the user says “that’s not what I meant” or “you missed the point,” that is evidence. It should not be hidden inside a generic conversational log. Those signals are often the most direct proof that the system failed to maintain user intent. If you discard them, you can no longer detect recovery failures reliably. The same applies to interruptions and reformulations. They are part of the behavior under test, not incidental text.

**Tool and retrieval artifacts belong at the turn where they matter.** Conversation-level evidence often needs more than speaker text. If the assistant used a tool, preserve the invocation and the returned result at the turn where it happened. If a retrieval system injected back-

ground passages, preserve those passages in the conversational record rather than folding them into a synthetic summary. The reason is diagnostic precision. A later failure may be caused by the wrong document being retrieved, the right document being ignored, or the tool returning a malformed result. If all of that is reduced to prose, you cannot localize the fault.

This does not mean you should dump every internal trace into the test case. Preserve only the artifacts that a later evaluator or reviewer might legitimately need to understand the interaction. The record should stay legible. When tool calls, retrieval evidence, and user turns are separated cleanly, you can check whether the assistant's answer was grounded in the right source, whether the action was taken at the right time, and whether the follow-up turn respected the prior state. That is the operational value of structured conversational evidence.

**Build the record from the user story, not from the transcript alone.** When you author a conversational test case, start with the product behavior. What should the assistant do when the user clarifies, contradicts, or revises the request? What state must persist? What outcome counts as successful completion? Then decide which turns, tool events, and context objects are necessary to evaluate that behavior faithfully. If a field has no plausible consumer in the harness, leave it out. If a field is essential for diagnosing stateful behavior, preserve it explicitly.

A strong conversational record is compact enough to review and rich enough to explain itself. It keeps role boundaries intact, preserves the order of commitments and corrections, and captures the artifacts needed to distinguish memory failure from retrieval failure, or recovery from drift. Once you do that, a metric can inspect the conversation with the right semantics instead of treating it as a longer prompt. That is what makes the evidence multi-turn rather than merely verbose.

### 2.3. Designing Goldens That Stabilize Evaluation Over Time

A benchmark suite can grow quickly and still become less trustworthy with every release. The prompts change, the model changes, the retrieval corpus changes, and the examples that once anchored a score begin to drift away from the behavior you actually care about. At that point, a good number on the dashboard no longer means much because it measures a moving target. Goldens solve that problem only if you treat them as durable benchmark commitments rather than disposable test cases. A golden is the piece of evidence you expect to remain interpretable after the surrounding system changes.

**A golden encodes intent, not a frozen transcript.** A useful golden records the scenario you want to preserve, the behavior you expect, and enough supporting context to judge the interaction consistently across versions. You do not want a golden that merely reproduces one phrasing of one successful run. You want one that captures the product obligation underneath the phrasing. In practice, that means the golden should preserve the user profile, the task shape, the source context, and the expected outcome, while leaving room for implementation details that may vary without changing correctness.

A simple example shows the distinction:

```
from deepeval.test_case import LLMTestCase

golden = LLMTestCase(
    input="Cancel my subscription at the end of this cycle.",
    context=[
        "User is on a monthly plan.",
        "Cancellation takes effect after current billing cycle.",
    ],
    expected_output=(
        "Confirm cancellation at cycle end and explain "
        "any remaining access."
    ),
)
```

The value of that record is not that the exact sentence will always be emitted. The value is that the benchmark now states, in durable form, what successful behavior looks like: the assistant must recognize the request, preserve the billing constraint, and respond with the right cancellation semantics. If the wording changes but the intent remains intact, the golden still serves its purpose. If the product logic changes, the golden will reveal that change as a semantic shift rather than a superficial text difference.

**Promotion into a golden suite should follow operational risk, not convenience.** Not every test case deserves canonical status. You promote cases that recur in production, expose important failure modes, or encode contractual behavior. Regression history is a strong signal: if a behavior has broken more than once, it belongs in the stable suite. So do scenarios tied to compliance, user safety, billing, account modification, and other high-impact paths. Toy examples and hand-picked easy wins are poor candidates because they inflate confidence without protecting users.

A practical selection rule is simple: if the case would be expensive to rediscover after a release regression, it probably belongs in the golden set. If the case is only useful as a quick smoke check or a prompt-development probe, keep it in an exploratory pool instead. That separation matters because canonical goldens must be reviewed, versioned, and maintained with more discipline than ad hoc experiments. Their cost is justified only when they protect a durable product obligation.

**Stability comes from preserving semantic structure, not exact phrasing.** The most reliable goldens separate what must remain fixed from what can legitimately vary. Fixed elements include the scenario, the expected outcome, and any source context required to understand the answer. Variable elements include surface wording, minor formatting choices, and other details that do not change the user-visible result. If you overfit the golden to one reference phrasing, you

force the suite to punish harmless model improvements. If you under-specify it, you lose the ability to detect drift.

This trade-off is easiest to see in open-ended tasks. A support answer may admit several acceptable phrasings, but the core obligations are still precise: acknowledge the issue, state the relevant policy, and avoid promising something the system cannot deliver. Encode those obligations in the golden itself. If you need exact strings, use them sparingly and only where the contract truly depends on wording, such as legal disclosures, safety refusals, or structured output schemas. For everything else, preserve the scenario and expected behavior at the right level of abstraction.

**Provenance makes a golden reviewable and retireable.** A golden without provenance becomes an unexplained artifact. You may still run it, but you cannot easily decide whether to keep it, revise it, or discard it. Every durable benchmark item should carry an origin story: where it came from, why it matters, who owns it, and what changed if its expected behavior no longer matches the product. That provenance can live in dataset metadata, adjacent documentation, or the evaluation repository, but it must exist somewhere durable.

Provenance also supports maintenance decisions. If a golden came from a real incident, you can trace it back to the failure class it represents. If it came from a product design review, you can connect it to a requirement. If it came from a synthetic generation pipeline, you can inspect the source documents and the generation method. Those details matter because a case generated from outdated context can quietly become misleading even when the text still looks plausible. Benchmarks do not fail only when they are wrong; they also fail when they become historically detached from the system they are supposed to represent.

**Synthetic goldens scale coverage, but they do not self-**

**validate.** Synthetic generation is attractive because it fills gaps quickly. You can derive many candidate goldens from a document set, a policy corpus, or a structured domain specification. That is especially useful in retrieval-heavy workloads, where source material is already available and you want broad coverage of likely questions. But synthetic cases are only candidates until you curate them. They often inherit the generator’s biases, overrepresent obvious examples, and miss the adversarial or ambiguous cases that matter most in practice.

The right way to use synthetic goldens is to treat them as accelerated draft material. Generate them from stable sources, inspect them for representativeness, and add provenance notes about the source document and the generation method. Then validate that they do not accidentally encode the answer in a way that makes the evaluation trivial. If a generated golden is too easy, too formulaic, or too close to the source text, it may measure retrieval overlap rather than actual model behavior. Synthetic generation expands the candidate pool; human curation still decides what belongs in the canonical suite.

**Staleness is a benchmark failure mode, not a housekeeping issue.** A golden can become misleading even when no one touched it. The source corpus may change, the product policy may evolve, or the user workflow may shift. In retrieval systems, the most common source of staleness is document drift: the evidence a golden was built around no longer matches the live corpus. In model systems, staleness can come from prompt evolution or output schema changes. In both cases, the benchmark may still run cleanly while the score becomes semantically ambiguous.

That is why versioning matters. You need to know which benchmark definition produced a result, which source corpus the goldens were built from, and which product behavior they were intended to guard. If a score changed after a policy update, you should be able to tell

whether the change reflects better modeling or just a changed reference. Version boundaries also help you compare runs responsibly. A score against version 1.4 of a benchmark is not directly comparable to one against version 2.0 unless you have intentionally preserved overlapping items and documented what changed.

**A good golden lifecycle is explicit and mechanical.** Promotion, revision, and retirement should follow a predictable process. A candidate case enters review. You check whether it represents a recurring or high-risk behavior, whether the context is stable enough to remain useful, and whether the expected outcome is stated at the right level of abstraction. If the case passes, you assign ownership, version it, and place it into the appropriate benchmark tier. If the case is too brittle, too synthetic, or too narrow, you revise it or keep it out of the canonical suite.

A compact lifecycle sketch helps keep that process operationally clear:

```
candidate -> review -> promote -> version -> revalidate
      |               |               |
      |               |               +--- keep or revise
      |               +--- compare against current product
      +--- reject if brittle or unrepresentative
```

That flow is deliberately conservative. Canonical goldens are expensive to maintain, so you should promote only cases that will remain meaningful over time. Retirement is just as important as promotion. If a golden no longer corresponds to the live product behavior, or if it is duplicative of a newer case, remove it from the stable suite and archive it with a note explaining why. Dead goldens quietly dilute signal by adding obsolete constraints to the benchmark history.

**Use structured expectations when exact reference text would be brittle.** One recurring decision is whether to lock an exact expected output or to encode a structured expectation. Exact references work well when the behavior is narrowly specified and wording matters.

Structured expectations work better when multiple outputs are acceptable as long as they satisfy a set of constraints. A cancellation confirmation, for example, may be correct if it confirms the request, states when the cancellation takes effect, and avoids unsupported guarantees. You do not need one frozen sentence to enforce that contract.

This distinction becomes important as products mature. Early in development, a rigid reference can help catch glaring regressions. Later, the same reference may become a maintenance burden because the assistant learns to express the same intent more naturally. If the benchmark is supposed to test product behavior rather than echo a wording template, move the expectation from exact phrasing toward structured acceptance criteria. That keeps the golden stable while still preserving the underlying obligation.

**Preserve enough context to make the outcome interpretable.**

A golden should contain the source context necessary to judge correctness, but not so much that the answer becomes inevitable. If the case comes from a policy document, include the relevant excerpt, not the entire handbook. If it comes from a product workflow, include the state that the user and assistant would actually know at the point of interaction. If the behavior depends on a retrieved document, preserve the source fragment and its provenance. The goal is to make the evaluation explainable without turning the benchmark into a copy-paste exercise.

Context discipline is especially important for RAG workflows. A retrieval-grounded golden should preserve the document slice that makes the correct answer possible, along with any metadata needed to trace the source. If the corpus changes, you should be able to identify whether the benchmark failure reflects a real product regression or simply a moved document. This is where goldens differ from casual examples: they are not just inputs and outputs, but durable evidence records with enough lineage to survive system change.

**Benchmark design benefits from explicit taxonomy.** Goldens stay interpretable longer when you organize them by scenario class, not just by topical label. A taxonomy helps you track coverage across common paths, edge cases, adversarial cases, policy-sensitive cases, and recovery cases. That structure is useful because it tells you what the suite is protecting. If all the goldens cluster around the easy path, the benchmark may look healthy while missing the failure surfaces that actually drive incidents.

A taxonomy also supports maintenance. When a product changes, you can review the affected scenario class instead of searching through a flat list of examples. If a new policy changes cancellation behavior, you revisit the cancellation goldens. If the retriever changes its chunking strategy, you inspect the retrieval-grounded items. If a safety rule changes, you review the refusal cases. The point is not taxonomy for its own sake; it is to keep the benchmark legible as a product artifact rather than as a pile of historical prompts.

**Comparability requires intentional overlap, not accidental drift.** If you refresh a benchmark, you still need continuity. That means preserving a core set of items across versions so you can compare old and new runs meaningfully. When you add new goldens, note whether they extend coverage or replace obsolete items. When you revise a golden, record whether the change reflects a product requirement update, a source corpus update, or a correction to the benchmark itself. Without that bookkeeping, a score improvement may simply reflect easier examples.

The right mental model is that a benchmark is a living asset with controlled evolution. It should not be frozen forever, but it also should not mutate silently. Keep the suite small enough to review, rich enough to represent the product, and versioned enough to explain why a result changed. That balance is what makes goldens useful over time rather than merely useful on the day they are written.

## 2.4. Choosing Coverage for Real Product Behavior

A large evaluation suite can still miss the behavior that matters. One team may accumulate thousands of prompts and feel safer for it, yet still fail to cover the product paths that decide whether users trust the system. Another team may keep a smaller, carefully chosen suite and make better release decisions because every case maps to a real risk. Coverage is not a counting problem. It is a design problem that starts with user impact, not with dataset size.

**Coverage should follow product risk, not benchmark convenience.** The first question is not how many examples you can collect, but which behaviors deserve protection. A high-value journey, a safety-sensitive refusal, and a routine paraphrase all deserve different levels of scrutiny. The benchmark should reflect that difference explicitly. If a behavior can hurt users, violate policy, break a contract, or trigger an expensive support incident, it belongs in the coverage model even if it is hard to label. If a behavior is easy to generate but irrelevant to product risk, it should not dominate the suite just because it is cheap to add.

A practical coverage strategy starts by mapping the product surface into scenario classes. You can usually separate them into common-path requests, edge cases, adversarial inputs, domain-constrained tasks, recovery scenarios, and business-critical journeys. That taxonomy gives you a way to reason about portfolio balance. A common-path case validates that the model handles the everyday workload. An edge case checks brittle boundaries. An adversarial case probes abuse resistance. A recovery case tests what happens after confusion or correction. A business-critical journey measures whether the interaction satisfies the actual obligation the user came for.

**Use different evidence types for different kinds of behavior.**

Once you know what the scenario classes are, decide which evidence object best preserves the behavior. A single-turn case is sufficient when the task is local and stateless. A conversational case is necessary when meaning depends on prior turns, corrections, or accumulated context. A golden becomes the durable reference when the interaction should remain comparable across releases. Coverage improves when you choose the smallest evidence object that still preserves the failure mode you care about.

That choice matters because overly large evidence objects can hide signal. If you flatten a multi-turn workflow into one prompt, you may measure only the final response and miss memory decay. If you split a stateful workflow into isolated single-turn cases, you may miss a trajectory that only fails after the assistant makes a commitment and then violates it later. The correct granularity is the one that keeps the relevant state visible. The wrong granularity often produces a suite that looks broad but cannot localize real failures.

**A practical coverage plan starts with an explicit portfolio.** You can express the portfolio as a small matrix of scenario classes and suite roles. Use smoke cases to catch gross regressions quickly. Use regression cases to protect stable product obligations. Use exploratory cases to probe new failure surfaces before they are promoted. Use release-gate cases to decide whether the current version is safe enough to ship. The same goldens can appear in more than one tier, but their role changes with the decision you are trying to support.

A compact operational sketch looks like this:

```
portfolio = {  
  "smoke": ["login_help", "refund_policy"],  
  "regression": ["billing_cancel", "handoff_escalation"],  
  "exploratory": ["long_context_repair", "adversarial_prompt"],  
  "release_gate": ["policy_refusal", "critical_booking_flow"],  
}
```

That structure is useful because it turns coverage into an auditable decision. You can ask why a case is present, which risk it protects, and which suite tier consumes it. If a case does not answer one of those questions, it is probably noise. The portfolio is not just a list of examples; it is a map from product behavior to evaluation intent.

**Combine production, historical, expert, synthetic, and adversarial data.** A balanced suite rarely comes from one source. Production data shows what users actually do. Historical incidents show what has already broken. Domain experts contribute cases that are rare but important. Synthetic generation helps scale coverage where real examples are sparse. Adversarial inputs expose abuse paths and brittle policy boundaries. OpenAI’s evaluation guidance emphasizes that these sources should be mixed rather than treated as substitutes for one another, and that the resulting dataset should include typical, edge, and adversarial cases rather than only the easiest examples.

The value of this mix is diversity of failure surfaces. Production traces reveal what the product sees in the wild, but they can underrepresent rare hazards. Historical incidents are high signal, but they only capture the past. Expert-authored cases can encode strategic judgment, but they may overfit the team’s intuition. Synthetic cases scale quickly, but they may inherit generator bias. Adversarial cases force the system to confront behavior users do not always announce plainly. You need all of them if you want the suite to reflect the full shape of risk.

**Typical cases prevent benchmark drift toward only the hard path.** It is easy to fill a suite with failures and edge cases because they are memorable. That makes the benchmark look rigorous, but it can distort your view of overall product quality. Typical flows still matter because they account for most of the traffic and the majority of user experience. If the assistant behaves well only on difficult examples but struggles on the routine path, the suite will mislead you about real utility.

Typical cases also act as a baseline for change detection. When the model improves, you want to know whether it improved on the common path, not just on a narrow set of pathological prompts. If the common path regresses, the product may feel worse even if the hard cases look strong. For that reason, a healthy suite keeps routine behavior visible and does not let edge cases monopolize the benchmark narrative.

**Edge cases should be selected for interpretability, not novelty.** An edge case is useful only if you can explain why it matters and what failure it exposes. Novelty alone does not make a good benchmark item. A strange prompt that produces a strange failure may be interesting for research, but if it does not correspond to a meaningful product risk, it should not displace more representative cases. Good edge cases usually sit near a known boundary: the maximum context length, an ambiguous user correction, a partial retrieval hit, a borderline policy violation, or a workflow that changes state after the assistant commits to an answer.

A strong edge case often reveals a class of bugs that a typical prompt misses. For example, a calendar assistant may appear fine until the user changes time zones mid-conversation. A support bot may work until the user asks for a refund after a partial shipment. A retrieval system may answer correctly until the source document contains conflicting clauses. You do not need many such examples, but you do need enough to expose whether the system fails in a predictable way when the boundary is reached.

**Adversarial cases protect against the behaviors users do not volunteer.** Adversarial coverage matters whenever the system can be manipulated, confused, or steered into unsafe output. These cases are not just about malicious users. They also cover prompt injection, instruction conflict, jailbreak attempts, and misleading context. A suite that omits adversarial examples can look excellent while leaving obvi-

ous exploit paths untested.

The key is to make adversarial cases realistic enough to exercise the system's defenses and specific enough to diagnose failures. You are not trying to generate abstract hostility; you are testing whether the model follows policy under pressure. That means the case should preserve the relevant context, the malicious instruction or conflicting constraint, and the expected refusal or safe completion. The benchmark should be explicit about which behavior counts as success. In safety-sensitive systems, an adversarial test often matters more than many easy successes because one miss can be far more costly than a long list of passes.

**Domain-specific tasks need evidence slices that match stakeholder concerns.** General correctness is not enough for specialized products. Health, finance, legal, education, and enterprise workflows each have their own acceptable-risk boundaries. HealthBench is a useful reminder that realistic coverage often requires domain-specific conversations and rubrics that capture nuance rather than generic right-or-wrong labels. In those settings, a benchmark item should preserve the stakeholder role, the domain constraints, and the granularity at which correctness will actually be judged.

That does not mean every suite needs a complex rubric. It means the evidence slice must match the domain. A finance assistant may need a gold case for fee disclosure. A medical assistant may need a case that preserves safety boundaries and escalation cues. A legal drafting tool may need a case that distinguishes factual support from unsupported inference. If you evaluate those systems with generic prompts, you may miss the quality dimension that matters most to the user and to the organization.

**Coverage depth and suite size are different levers.** A larger suite is not automatically a better suite. If you add many cases from

the same scenario class, you may increase confidence in that class while leaving other risks unrepresented. Depth helps when you want to stress one behavior under slightly different conditions. Breadth helps when you want to expand into a new risk surface. Confusing the two leads to suite bloat.

A useful rule is to expand breadth first until you cover the major scenario classes, then deepen only where the product risk justifies it. For example, if your assistant handles account support, scheduling, and policy questions, make sure each category has at least a few representative cases before you add ten variants of the same password-reset prompt. Once the major surfaces are covered, additional variants are valuable only if they probe a real boundary, a known regression, or a measurable source of variance.

**Coverage decisions should be traceable to release decisions.**

If a benchmark does not inform a release decision, it is probably not carrying its weight. Every suite tier should answer a specific question. Smoke cases answer whether the system is obviously broken. Regression cases answer whether known product obligations still hold. Exploratory cases answer what new failure surfaces appear as the system changes. Release gates answer whether the current version is safe enough to ship. When a case does not support one of those decisions, keep it out of the canonical portfolio or move it to a research pool.

This traceability also helps when the product evolves. If a feature is removed, you should know which benchmark items become obsolete. If a policy changes, you should know which goldens require revision. If a new workflow is added, you should know which scenario class is missing coverage. The benchmark should not be a passive archive. It should be a controlled representation of product risk.

**Maintenance is part of coverage, not an afterthought.** Coverage decays when no one owns it. Production traffic shifts, policies

change, and source corpora drift. A suite that was balanced six months ago can become lopsided if one scenario class keeps growing while another silently disappears. That is why you need review cadence and ownership. Periodically ask whether the cases still reflect current user behavior, whether any goldens have become brittle, and whether any scenario class is underrepresented.

The best suites carry explicit maintenance triggers. A gold case should be revisited when the product behavior changes, when the source documents change, or when repeated false failures suggest that the expectation is too specific. A synthetic case should be retired if it no longer reflects a realistic path. A historical incident case should stay only if the failure mode is still plausible. Maintenance keeps the suite honest. Without it, the benchmark becomes a museum of old assumptions.

**A balanced suite makes the product legible.** Coverage becomes useful when it tells you what the product can do, what it cannot do, and where it is most likely to break. That clarity comes from intentional composition: common paths for baseline behavior, edge cases for brittleness, adversarial cases for robustness, domain-specific cases for stakeholder requirements, and goldens for durable regression control. When those parts are aligned, the suite becomes a decision tool rather than a scoreboard. When they are not, the suite may still look impressive while telling you very little about the system users actually experience.



## Chapter 3

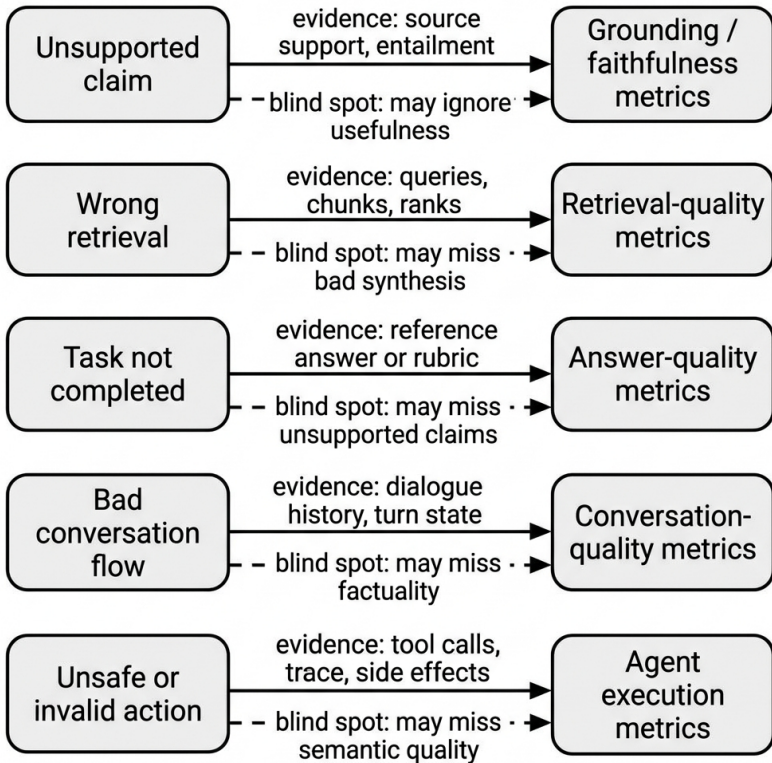
# Selecting Metrics and Understanding LLM-as-a-Judge

*A strong evaluation harness does not begin with whichever metric is easiest to import; it begins with the operational question, the failure mode, and the decision that the metric must support. This chapter teaches readers to treat metrics as instruments with scope, bias, and failure characteristics of their own, then use that understanding to build judge-based evaluation that is interpretable, debuggable, and aligned with real engineering risk.*

### **3.1. Metric Families as Failure-Mode Instruments**

A single attractive score can hide the defect that matters most. A team may watch answer similarity rise while the system quietly begins omitting citations, fabricating facts, or retrieving the wrong context. That failure pattern is common because metrics are often treated as universal grades instead of instruments with a narrow measurement band. The right question is not which metric is best in the abstract; it is which failure mode you need to expose, what evidence exists for that failure, and what kind of metric can observe it without being misled by the rest of the system.

For a concrete starting point, consider a support assistant that answers policy questions from an internal knowledge base. If you score only final-answer relevance, a fluent but unsupported answer can look excellent. If you score only grounding, the assistant can earn high marks for citing the right passages while still failing to solve the user’s actual problem. If you score only retrieval quality, you can prove the search layer worked while the generation layer still invents details. The measurement problem is therefore structural: each metric family sees a different slice of the system, and each one becomes dangerous when used outside its intended failure surface.



That map is the core vocabulary for the rest of the chapter. A failure mode names the defect you care about. A metric family names the measurement instrument that can detect that defect. The distinction matters because one metric can be excellent for release gating in one setting and actively misleading in another. Exact-match scoring is strong when the output space is constrained; it becomes brittle when multiple answers are acceptable. Semantic similarity can capture paraphrase and partial overlap; it becomes weak when unsupported content is more dangerous than lexical mismatch. Judge-based scoring can evaluate nuanced criteria; it becomes fragile when the rubric is vague or the

judge is not calibrated. You do not choose a metric because it sounds sophisticated. You choose it because it can see the defect you expect.

**Failure mode first, metric family second.** Start from the product risk. If the cost of failure is an unsupported medical recommendation, your primary concern is not phrasing elegance but evidential grounding. If the cost of failure is a bad support reply that ignores the user's question, then response usefulness matters more than whether every sentence is backed by retrieved text. If the cost of failure is a tool-using agent taking the wrong action, then correctness of the final natural-language answer is secondary to the validity of the action trace.

That ordering also prevents a common anti-pattern: adopting a convenient metric family because the test harness already supports it. Convenience can be a reasonable constraint, but it is not a justification. The metric must be coupled to an observable defect. If you cannot name the defect, the score is decorative.

**Reference-based and referenceless evaluation.** The first split in the landscape is whether you have a reference. Reference-based metrics compare the system output against an expected answer, expected structure, or known target behavior. Referenceless metrics instead inspect the output and its supporting evidence directly, without requiring a gold answer. The split is more than a data-management detail; it changes what kinds of claims the metric can make.

Reference-based evaluation is useful when the task is bounded and the target behavior is stable. A structured extraction task, a finite classification task, or a scripted response pattern can often be scored against an expected result. In those settings, a reference gives you a crisp target and makes regression testing straightforward. The weakness is obvious: a reference only helps if it represents all acceptable solutions or at least the defect class you care about. When the response space is open-ended, a single reference often under-specifies correctness and

punishes legitimate variation.

Referenceless evaluation is better when the main question is not *Did the model match this answer?* but *Did the model obey the constraint?* or *Did the model use the evidence correctly?* Grounding metrics are a typical example. They do not need a canonical answer if the claim is directly checkable against provided context. Conversation-quality metrics are another. They may care about coherence, turn-taking, refusal behavior, or task progress rather than exact textual overlap with a gold reply. The trade-off is that referenceless metrics often require a stronger rubric or stronger judge, because the metric must infer correctness from evidence rather than compare strings.

**Answer-quality metrics.** Answer-quality metrics ask whether the response satisfies the user’s request. They are the right tool when the user’s goal is explicit and the acceptable outputs are still broad enough that exact matching is too narrow. In practice, these metrics often blend relevance, completeness, clarity, and format compliance. For a bounded task, a reference-based answer-quality metric can be enough. For a more open task, a judge-based score is common because the metric must account for paraphrase, partial success, or domain-specific acceptability.

Their blind spot is that answer quality says little about provenance. A model can answer the question and still hallucinate the supporting facts. If you use answer quality alone, you reward fluent synthesis even when the system is fabricating. That makes the metric useful for user satisfaction but unsafe as the only release gate in domains where unsupported content matters.

A practical rule follows from that blind spot: if a wrong answer can be costly even when it sounds good, pair answer-quality metrics with a grounding or correctness metric. Let answer quality tell you whether the response is useful; let another family tell you whether it is justified.

**Grounding and faithfulness metrics.** Grounding metrics ask whether the output is supported by the supplied context, retrieved passages, tool traces, or source documents. Faithfulness is the narrower claim that the answer does not introduce unsupported statements. These metrics are central whenever generation is constrained by evidence: retrieval-augmented generation, citation-heavy workflows, summarization over source documents, and assistant behavior that must stay inside an approved corpus.

The key strength of grounding metrics is that they can expose hallucination even when the output is polished. Their weakness is complementary: they can miss whether the answer is actually helpful. A perfectly grounded answer that addresses the wrong subquestion is still a failure. They can also miss subtle synthesis errors, where each sentence is individually supported but the conclusion drawn from them is wrong. That failure mode is common in multi-hop reasoning, where support exists but the inference chain is invalid.

You should treat grounding metrics as evidence-consumption checks, not as full task-success scores. They tell you whether the system used the provided evidence responsibly. They do not tell you whether the evidence was retrieved correctly, whether the user’s intent was satisfied, or whether the answer is complete.

**Retrieval-quality metrics.** Retrieval metrics measure whether the search or selection stage found the right evidence. Typical observables include recall at  $k$ , precision at  $k$ , mean reciprocal rank, and evidence relevance judgments. These metrics are valuable when the model’s generation quality is constrained by upstream retrieval. If the retriever misses the critical passage, the generator often cannot recover.

Retrieval metrics detect a different failure surface from grounding metrics. A grounded answer may still rest on a poor retrieval process if the model had to ignore a lot of irrelevant context to produce some-

thing acceptable. Conversely, a retrieval system can score well while the generator fails to use the retrieved material. That separation is useful operationally because the remedy differs: a retrieval defect points to indexing, chunking, query formulation, or reranking; a grounding defect points to prompt design, context usage, or generation policy.

Their blind spot is that retrieval quality often assumes a notion of relevance that is only loosely aligned with final user value. A chunk may be statistically relevant and still not help the answer. Another chunk may look weak in lexical similarity but contain the one fact that resolves the task. For that reason, retrieval metrics are strong diagnostic signals, but they should not be mistaken for end-to-end product success.

**Conversation-quality metrics.** Conversation-quality metrics capture turn-level behavior across a dialogue: coherence, context retention, appropriate follow-up questions, refusal behavior, tone, and state tracking. These metrics matter when the user interaction is iterative rather than one-shot. A model that gives a correct final answer can still be a poor conversational system if it forgets prior constraints, contradicts itself, or answers too early without clarification.

These metrics are frequently judge-based because the relevant property is distributed across multiple turns and depends on conversational context rather than a single output string. The important discipline is to keep the criterion narrow. If a conversation metric tries to judge coherence, politeness, factuality, and instruction following all at once, the score becomes hard to interpret and difficult to debug. Split the failures by behavior class when you can. A conversation can be polite and still be unhelpful; it can be coherent and still be unsafe; it can maintain state and still miss the user's request.

Conversation-quality metrics are especially vulnerable to over-crediting fluent style. A model that sounds engaged can mask shallow reasoning. Use them to assess dialogic behavior, not to replace factual

or grounding checks.

**Agent execution metrics.** Agent metrics evaluate whether an autonomous or tool-using system chose valid actions, invoked the right tools, respected constraints, and completed the task without damaging side effects. The observable evidence is no longer just text. You need tool calls, intermediate state, execution traces, and sometimes environment deltas. That makes this family distinct from ordinary answer scoring, because the natural-language reply can be correct while the action path is wrong, or the action path can be correct while the final reply is terse.

For agents, the highest-value metrics often inspect process correctness before output quality. Did the system call the right function with the right arguments? Did it stop when it should? Did it request clarification instead of making a destructive assumption? These are execution questions, not language questions. A language-only metric will miss them.

The blind spot runs in the opposite direction: execution metrics can verify that the agent used the right tool sequence but say little about whether the user received a useful result. That is why agent evaluation often needs a stack: one family for tool correctness, one for task completion, and one for the quality of the final communication.

**When a metric is informative, weak, or misleading.** A metric is informative when its evidence is close to the failure mode, when its rubric is aligned with the defect, and when its false positives and false negatives are acceptable for the decision at hand. It is weak when it sees only a proxy for the failure, or when it needs assumptions that are often violated. It is misleading when it produces a number that looks authoritative while systematically ignoring the defect you care about.

That last category deserves the most caution. Similarity metrics can be misleading when paraphrase hides factual error. Grounding met-

rics can be misleading when all claims are supported but the response misses the user's intention. Retrieval metrics can be misleading when good ranking does not translate into good synthesis. Judge metrics can be misleading when a plausible explanation masks unstable scoring. The score is never the truth; it is a measurement with a measurement model. If the model is wrong, the score is wrong in a way that can look convincingly precise.

**Choosing a portfolio from the evidence fields you actually have.** Metric choice depends partly on what your test case contains. If you have only an input and an output, you can score surface behavior, some answer quality dimensions, and possibly judge-based style or policy criteria. If you also have an expected output, you can add reference-based checks. If you have context or retrieved passages, you can score grounding. If you have retrieval traces, you can score search quality. If you have tool logs, you can score execution behavior.

That relationship is operationally important because it limits overreach. Do not ask a metric to infer evidence you never captured. A grounding metric cannot be reliable if the relevant context is absent. An agent metric cannot be trustworthy if tool calls are not logged. A reference-based metric cannot validate open-ended tasks if the reference is only one of many acceptable solutions. The test case model defines the observable space; the metric family must fit inside it.

**False negatives, false positives, and business cost.** Metric selection is ultimately a risk decision. A false negative means the metric misses a real failure. A false positive means the metric blocks a behavior that is acceptable or even desirable. The right balance depends on the cost of each error. In a high-stakes domain, you may prefer a conservative metric that catches more defects at the expense of extra human review. In a low-stakes productivity tool, you may prefer a lenient metric that avoids unnecessary blocking and uses diagnostics rather than gates.

That trade-off changes how you deploy families. Grounding and safety metrics often deserve release-blocking status because the harm from a miss can be large. Conversation fluency or brevity may belong in the advisory tier because a false positive there is annoying but not catastrophic. Retrieval metrics are often diagnostic, because they help localize the issue even when they are not the final decision rule. The same family can play different roles depending on the business risk, but the role must be explicit.

**A disciplined selection workflow.** A practical selection pass begins with the defect list. Write down the top production failures you fear, not the qualities you admire in a demo. Map each failure to the smallest metric family that can observe it. Then check the evidence fields available in your evaluation dataset. If a failure has no observable evidence, either add the evidence to the test case or accept that the metric will remain indirect.

From there, separate release blockers from diagnostics. Release blockers should correspond to non-negotiable failures with clear thresholds and low ambiguity. Diagnostics should help you understand why a run failed and where to fix it. Keep the families separate. A metric that is useful for triage is not automatically stable enough for gating, and a gating metric that is too coarse will hide the local defect you need to repair.

That discipline is what turns evaluation from scoreboard theater into engineering practice. You are not searching for one perfect scalar. You are assembling a set of instruments whose combined coverage matches the system's real failure surface.

## 3.2. How LLM-as-a-Judge Metrics Really Work

When a team first adopts judge-based evaluation, the appeal is obvious: you can score qualities that are hard to pin down with string matching, and you can do it without hand-labeling every edge case. The risk is just as obvious once you look closely. A judge metric is not a mystical evaluator; it is a prompted program whose behavior depends on the evidence it receives, the rubric it is given, the scoring path it follows, and the model that interprets all of those instructions. If you understand those pieces, you can debug the metric. If you do not, a polished score can mask the same problems that plagued the system under test.

A compact judge prompt makes the structure visible. The exact syntax varies by library, but the logic is stable: define the task, provide the available evidence, specify the rubric, constrain the score range, and decide whether the judge should explain itself. In a typical OpenAI-style grader flow, you make the criteria explicit and bind the grader to the sample fields instead of leaving it to infer the task from free-form text.

```
from openai import OpenAI

client = OpenAI()

prompt = """
Rate the answer on factual grounding.
Use only the provided context.
Return a score from 0 to 1.
"""

resp = client.responses.create(
    model="gpt-4.1",
    input=[
        {"role": "system", "content": prompt},
        {"role": "user", "content": "Q: ..."},
        {"role": "user", "content": "Context: ..."},
        {"role": "user", "content": "Answer: ..."},
    ],
)
```

That fragment shows the core design pattern. The judge is not asked to improvise a theory of quality from scratch. It is given bounded inputs and a constrained output format. The closer you keep the prompt to the measurement question, the easier it becomes to reason about the score.

**Judge metrics are prompted measurements.** A judge metric receives some mixture of task input, candidate output, expected output, supporting context, retrieval evidence, tool traces, or a rubric. It then produces a bounded verdict: a label, a score, a pairwise preference, or a pass/fail decision. The key point is that every one of those outputs is a measurement artifact, not a direct property of the system. The judge does not *observe* correctness in the way a unit test observes a return value. It infers correctness from a prompt and whatever evidence you decided to provide.

That makes judge metrics unusually flexible. You can ask for semantic relevance, policy compliance, reasoning quality, tone, or citation discipline without inventing a deterministic checker for each one. The same flexibility also makes them brittle. Small changes in wording can shift what the judge treats as salient. A missing reference answer can change whether the score reflects absolute quality or relative similarity. A weak rubric can leave the model free to reward style over substance. A large score range can imply precision that the prompt cannot support.

A useful mental model is to treat the judge as an evaluation operator with an explicit input contract. If the contract is underspecified, the score inherits that ambiguity. If the contract is overpacked, the score becomes a compressed summary of several latent judgments that may not agree with each other.

**The anatomy of a judge metric.** A well-formed judge metric usually contains six parts.

First, *task framing* tells the model what class of behavior it is judging.

A relevance judge, for example, should not have to guess whether it is scoring usefulness, factuality, or brevity.

Second, *rubric instructions* define the criterion. This is where you constrain what counts as success, what counts as failure, and which evidence the judge may use. A good rubric excludes irrelevant information as aggressively as it includes relevant information.

Third, *available evidence* specifies the fields the judge may inspect. That may include the original prompt, the candidate answer, reference output, retrieved passages, conversation history, or tool trace. The evidence set is part of the metric design, not an implementation detail.

Fourth, *scoring scale* defines the output shape. A binary pass/fail verdict is easier to interpret than a seven-point scale, but it loses nuance. A 0–1 score is convenient for aggregation, but it can disguise low-confidence distinctions. A 1–5 scale may feel more human-like, yet it can invite spurious precision if the rubric does not truly support that granularity.

Fifth, *threshold logic* determines how the score becomes a decision. A metric can be advisory at one threshold and release-blocking at another. If you do not define the threshold behavior, the same raw score may be read differently by different consumers.

Sixth, *reason-generation instructions* control whether the judge emits an explanation and what that explanation should contain. Reason strings are useful for debugging, but they are also a source of false confidence when they sound more grounded than they are.

If any of those parts is vague, the metric becomes noisy. If several parts are vague, the metric becomes hard to audit because you cannot tell whether a failure came from the model, the rubric, the evidence, or the threshold.

**What happens at evaluation time.** At runtime, the judge receives

the prompt and evidence bundle, then maps that input through its internal representation of the rubric. That process usually looks simple from the outside: the model emits a score and maybe a brief explanation. Internally, though, the score emerges from several hidden decisions. The judge must decide which evidence to privilege, how to interpret ambiguous criteria, how to balance competing rubric clauses, and whether to treat a borderline case as passing or failing.

This is where the strongest failure modes appear. A prompt that looks precise to a human may still leave the judge with multiple valid readings. If the rubric says *be concise and complete*, the judge has to decide whether concision can justify omission. If the rubric says *use only the provided context*, the judge must decide whether common knowledge is disallowed or merely lower priority. If the rubric asks for *correctness and helpfulness*, the judge has to resolve the fact that those dimensions can diverge.

The evaluation model may also compress uncertainty into a single categorical output. A score of 4 out of 5 can hide very different conditions: a response could be almost perfect with one small omission, or broadly mediocre but stylistically polished. The number is useful, but only if you know what the number can and cannot distinguish.

**Absolute scoring versus pairwise comparison.** Judge metrics usually come in two major forms. Absolute scoring asks whether one output meets a rubric on its own. Pairwise comparison asks which of two outputs is better under the rubric. Absolute scoring is easier to integrate into automated pipelines because every sample can be evaluated independently. Pairwise comparison often produces more stable preferences in open-ended tasks because the judge only needs to distinguish relative quality, not invent a full calibration scale.

The trade-off is operational. Absolute scoring gives you a direct threshold, but it can suffer from score compression, especially when the judge

hesitates to use the extremes of the scale. Pairwise comparison gives you cleaner contrast, but it requires a comparison policy if you want a single output score. It also becomes sensitive to answer order unless you control for position effects.

In practice, pairwise protocols are often more persuasive for subjective tasks such as helpfulness or style, while absolute scores are better when the rubric defines a hard requirement such as grounding, policy compliance, or schema validity. The right choice depends on whether you need a ranking signal or a compliance signal.

**Reason generation is useful, but not authoritative.** A judge explanation is not a proof. It is a post hoc textual artifact produced under the same latent biases that produced the score. That makes reasons extremely useful for debugging and extremely dangerous as evidence of correctness. Engineers often read a well-formed explanation and infer that the judge must have reasoned carefully. That inference is too strong.

The explanation can help you detect rubric misalignment. If the judge repeatedly cites irrelevant details, the prompt may be underspecified. If the judge praises fluency when the rubric was supposed to assess factuality, the metric is drifting. If the judge's reasons differ materially across repeated runs on the same sample, the metric is unstable. Those are real diagnostics.

What the explanation cannot do is guarantee that the score is valid. A convincing reason can still be anchored on a wrong premise. It can also be influenced by superficial fluency in the candidate answer or by phrasing in the rubric itself. For that reason, treat reasons as inspection traces, not as evidence that the metric is correct.

**G-Eval adds structure to the judge path.** G-Eval-style metrics formalize the judge process by introducing intermediate steps before the final score. Instead of asking the model to jump straight from evi-

dence to rating, you first induce explicit evaluation steps, then ask the model to score against those steps. That decomposition helps in two ways. It reduces underspecification, because the criteria are broken into narrower decision points. It also improves auditability, because you can inspect whether the score follows the planned rubric.

This structure matters most when the target criterion is composite. If you want to score answer quality, for example, you may need separate checks for relevance, completeness, and factual support. A single prompt that bundles all three can produce a score that hides which dimension failed. A stepwise rubric surfaces that structure and makes disagreement easier to diagnose.

Some implementations also normalize ratings using token probabilities over score labels rather than relying only on the decoded label. The motivation is straightforward: if the model assigns similar probability mass to adjacent labels, the metric should reflect that uncertainty instead of pretending the winner was certain. That approach does not eliminate ambiguity, but it makes the scalar less brittle than a bare `argmax` over labels.

**Why prompt wording changes the score.** Judge metrics are prompt-sensitive because the prompt is part of the measurement instrument. If you change the rubric wording, you change the model’s latent task. If you change the order of the evidence fields, you may change what the model pays attention to first. If you include the expected answer, you tilt the metric toward reference matching. If you omit it, you push the judge toward referenceless assessment. Each choice is legitimate, but each one measures a different thing.

That sensitivity is not a bug in the abstract; it is a sign that the metric is doing real work. The problem arises when the team treats the prompt as incidental. If two phrasing variants produce different scores on the same samples, the metric is not yet a stable instrument. You need to

know which phrasing captures the intended defect more faithfully and which phrasing merely changes the model's mood.

A practical way to think about this is to treat rubric wording as a calibration parameter. Tight wording usually reduces ambiguity but can increase brittleness if the failure mode is broader than the prompt admits. Loose wording can cover more scenarios but often reduces reproducibility. Good judge design makes that trade-off explicit.

**Variance sources you have to control.** Several independent sources of variance affect judge metrics.

- Model variance appears when the same input yields different outputs across runs. Even with low temperature, the model may still drift because of internal stochasticity or hidden context effects.
- Prompt variance appears when minor wording edits produce different scores. This is especially common when the rubric is underspecified or contains multiple criteria in one sentence.
- Order sensitivity appears when the position of candidates or evidence changes the verdict. In pairwise scoring, answer-order bias can be substantial enough to alter rankings. In multi-field scoring, earlier text can anchor later judgments.
- Familiarity and style bias appear when the judge prefers text that looks more familiar, more fluent, or more like typical model output. That can distort scores toward polished but shallow answers.
- Self-enhancement bias appears when the judge prefers outputs that resemble its own stylistic patterns or reasoning style. This is one reason a strong-sounding rationale should not be treated as a guarantee of objectivity.
- Score skew appears when the distribution clusters tightly near the top or middle, reducing the metric's ability to separate good

from bad examples. If every sample gets a 4 or 5, the metric is not discriminating enough to support decisions.

You do not remove all of these effects. You manage them by testing the metric under perturbation: rerun it, swap answer order, vary the evidence presentation, and compare disagreement patterns. The point is to measure the metric's own reliability, not just its average score.

**Calibration turns scores into usable signals.** A judge metric becomes useful when you calibrate it against reviewed examples. Calibration means checking whether the score distribution corresponds to the decision you care about. A raw score of 0.82 is not meaningful by itself. It matters only in relation to a threshold, a failure category, and a review sample that tells you what the score really means in practice.

For binary gates, calibration often reveals an operating point where false negatives remain acceptable and false positives do not overwhelm the team. For graded advisory scores, calibration helps you determine whether the scale supports meaningful ranking or whether it is mostly decorative. If adjacent score buckets map to the same human judgment, the metric is over-resolving. If many different failure modes collapse into one bucket, the metric is under-resolving.

Calibration also exposes whether the reasons are trustworthy. If the judge's explanation consistently matches human review, it can help analysts triage failures quickly. If the explanation looks plausible but does not track the actual decision, it becomes a liability.

**Cost and interpretability are part of the design, not afterthoughts.** Judge metrics are often more expensive than deterministic checks, and the cost goes beyond inference tokens. You pay for prompt design, rubric iteration, calibration runs, disagreement review, and ongoing maintenance when the underlying model changes. A metric that is cheap to compute but expensive to interpret can still

be the wrong choice if it slows every review meeting.

Interpretability depends on how much structure you preserve. Binary metrics are easy to explain but can hide nuance. Fine-grained scores preserve nuance but often need a reason trace to be actionable. Pairwise comparison is intuitive for humans but awkward for automation unless you convert the preference into a stable reporting format. The best judge metrics make their own uncertainty visible. When a metric cannot tell you why a sample failed, it is less useful as a debugging instrument even if it is statistically stable.

The practical discipline is to align complexity with decision value. If the release decision only needs a hard pass/fail check, do not pay for a richly explained 1–5 scale. If analysts need to understand which failure mode dominates, do not collapse distinct criteria into one opaque number. Structure should follow the decision, not the other way around.

**A repeatable evaluation workflow.** A judge metric is easiest to trust when you validate it in a fixed sequence. Inspect the prompt and the evidence fields first. Confirm that the rubric names one evaluative question, not several. Run the metric on a reviewed slice of samples. Repeat it on the same slice to estimate stability. Swap answer order if the protocol is pairwise. Read the reason strings and check whether they cite the actual failure mode. Then choose a threshold or comparison rule based on the reviewed sample, not on intuition.

That workflow is intentionally mundane. Judge systems fail most often when they are treated as one-shot oracles. They become tractable when you treat them as inspectable programs with their own operating characteristics. At that point, the score stops being a mysterious verdict and becomes what it should have been from the start: a constrained measurement with known failure modes, known biases, and a calibration history that tells you how much confidence to place in it.

### 3.3. Designing Robust Custom Metrics with G-Eval

Predefined metrics get you only as far as the generic failure modes they were built to observe. Once your system is expected to obey internal policy, preserve citation discipline, respect workflow conventions, or refuse a restricted action, you need criteria that reflect your own operating rules rather than someone else's benchmark. G-Eval-style metrics are the practical answer: you turn a business requirement into an explicit rubric, bind it to the evidence you can inspect, and ask a judge model to apply that rubric consistently. The value comes from structure, not from the label "LLM judge."

A minimal custom metric definition makes the design choices concrete. In a G-Eval-style implementation, you specify the input fields, the criterion, the rubric, and the pass threshold. When the failure mode is known, you usually get better control by writing explicit steps instead of leaving the judge to infer the rubric from a broad natural-language description.

```
from deepeval.metrics import GEval
from deepeval.metrics.geval import EvaluationParams

metric = GEval(
    name="unsupported_claims",
    criteria="Check that the answer does not "
            "make claims unsupported by context.",
    evaluation_params=[
        EvaluationParams.CONTEXT,
        EvaluationParams.ACTUAL_OUTPUT,
    ],
    evaluation_steps=[
        "Identify each factual claim in the answer.",
        "Check whether each claim is supported by context.",
        "Fail if any claim is unsupported.",
    ],
    threshold=0.8,
)
```

That example is intentionally small. It shows the core discipline: you tell the judge what evidence it may use, what defect to detect, and how to decide. The metric becomes debuggable because the evaluation question is narrow enough to inspect.

**Start from the defect, not the wording.** A custom metric should encode a defect class. “Professional tone” is not a defect class unless you can specify visible, testable properties that correspond to tone in your context. “Does not claim unsupported account actions” is a defect class. “Contains a required compliance disclaimer when recommending a restricted action” is another. The first is vague; the second and third are observable.

That difference matters because judge models can only score what they can see. If you ask for a subjective quality that is not anchored in the available evidence, the model will substitute its own interpretation. That can be useful for exploratory analysis, but it is too unstable for disciplined evaluation. A good custom metric therefore starts with an exact failure statement: what went wrong, what evidence should reveal it, and what should be ignored.

This is where G-Eval-style design is especially strong. It pushes you to write the criterion as an operational rule rather than a brand statement about quality. The more closely the rule maps to the defect, the easier it is to debug disagreement cases later.

**Scope each metric to one evaluative question.** A single metric should answer one question. If it tries to answer several, you lose interpretability and inflate disagreement inside the score. A rubric that asks whether an answer is factual, complete, concise, and polite sounds efficient, but it actually bundles at least four latent judgments that may not move together. The judge can pass one dimension and fail another, and the resulting score will obscure which one drove the outcome.

This is the main reason to separate dimensions into independent met-

rics. Correctness, completeness, tone, policy compliance, and citation quality usually deserve distinct scores. That separation makes anchoring less likely and gives you diagnostic leverage. If correctness falls while tone stays stable, you know where to intervene. If everything is merged into one score, you only know that the aggregate moved.

The same principle applies to composite rubrics. If a criterion secretly contains a conjunction such as “must be grounded and concise and safe,” split it unless the business decision truly requires all three at once. G-Eval works best when each metric has a crisp semantic boundary.

**Make the evidence contract explicit.** A robust metric names the evidence the judge may use. That evidence contract is not a minor implementation detail; it defines the measurement space. If the metric is supposed to inspect grounding, provide the retrieved context or source passages. If it is supposed to inspect compliance with a policy, provide the policy snippet or the approved rule set. If it is supposed to judge conversation behavior, provide the relevant dialogue turns, not just the last reply.

The evidence contract also defines what *not* to use. If world knowledge is unavailable or unreliable in the deployment context, say so. If the judge should ignore style except where style affects policy compliance, say that directly. The fewer implicit assumptions the prompt makes, the easier it is to interpret score drift later.

A common mistake is to ask a judge to infer missing facts from its own latent knowledge while pretending the metric is about grounded evaluation. That produces a hybrid signal that is hard to trust. Either supply the needed evidence, or restrict the rubric so the judge only evaluates what is present.

**Prefer explicit steps when the failure mode is known.** Natural-language criteria are useful for exploration, but explicit evaluation

steps usually improve determinism when the defect class is already understood. G-Eval’s stepwise style is valuable because it decomposes the judgment into smaller subdecisions. Instead of asking the model to directly score “answer quality,” you ask it to check relevance, then completeness, then support, and only then emit a final rating.

That decomposition has two benefits. First, it reduces ambiguity. A judge that must follow ordered checks is less likely to conflate adjacent criteria. Second, it improves debuggability. If a sample fails, the recorded steps tell you which subcriterion triggered the failure. That makes threshold tuning and rubric revision much faster.

You do not need explicit steps for every metric. If the quality dimension is truly simple, a short criteria statement may be enough. Use steps when the defect is multi-stage, when the judge is likely to conflate dimensions, or when you want a review trail that an engineer can inspect without reverse-engineering the prompt.

**Binary, graded, and gate-based metrics serve different decisions.** A custom metric should match the decision it supports. Binary metrics are ideal when the requirement is non-negotiable: policy compliance, valid schema, no unsupported claim, required disclaimer present. Graded metrics are useful when you want to rank outputs or observe improvement across training iterations. Gate-based metrics sit in between: the sample must pass the hard requirement before softer preferences are even considered.

This distinction is operational, not cosmetic. A graded score can be tempting because it feels informative, but it becomes misleading if the underlying criterion is really a release gate. If a compliance rule is mandatory, do not hide it inside a 1–5 score with a threshold near the top. Use a hard pass/fail metric for the rule and reserve the graded score for secondary qualities such as clarity or brevity.

The reverse is also true. If the business cares about ranking candidates,

a binary metric throws away too much information. In that case, a graded rubric is worth the calibration work because it captures meaningful differences among acceptable outputs.

**Avoid overlapping metrics and hidden double counting.** Overlapping metrics produce false confidence. If two metrics inspect almost the same defect, they will often move together and create the appearance of corroboration when they are really measuring the same thing twice. A good metric portfolio uses distinct observables. One metric can check factual support, another can check citation presence, and a third can check whether the answer resolves the user’s task. Those are related, but they are not identical.

Double counting is especially common when teams phrase multiple metrics around the same business language. “Accuracy,” “relevance,” and “quality” often hide substantial overlap. If the judge sees the same evidence and applies similar latent criteria, those scores will not help you isolate failures. They will only inflate the apparent confidence of the aggregate.

A useful test is to ask whether a metric would still be informative if the others were removed. If the answer is no, the metric is probably redundant. Redundancy is not always bad, but it should be deliberate. You may want one diagnostic metric and one release gate that cover adjacent failure surfaces. You do not want several names for the same judgment.

**Use references when they add real information.** When you have an expected output, reference-based judge criteria often improve stability. A reference can anchor the judge on what a good answer looks like, which is especially useful for bounded tasks, structured outputs, or domain-specific templates. When the expected output is available and representative, it materially reduces ambiguity.

That said, references are not universally helpful. If the task admits

many correct answers, a single reference can become an unnecessary constraint. In that case, you should use the reference as a guide, not as a literal target, or omit it entirely and rely on evidence-based rubrics. The key question is whether the reference adds discriminative power or simply narrows acceptable variation.

Prompt plus reference generally works better than prompt alone when the criterion involves fine distinctions that humans already resolved. Official grader guidance and systems such as Prometheus point in the same direction: structured rubrics perform best when the judge is given a clear target and enough evidence to compare against it.

**Design for reviewer agreement, not just model agreement.**

A metric is useful only if humans can understand and audit it. That means the rubric must be written in terms reviewers can apply consistently. If reviewers disagree on what the metric means, the judge will inherit that ambiguity and exaggerate it at machine scale.

You should therefore write criteria that are observable, not aspirational. “Helpful” is too broad unless you define what helpfulness means in the context of the product. “Contains the required escalation path when the user requests account closure” is inspectable. “Sounds confident” is not. The more your rubric aligns with visible evidence, the easier it is to get stable human agreement and the easier it is to diagnose judge failure.

This is also where reviewer disagreement becomes a design signal. If multiple humans cannot apply the metric consistently, the problem is usually the rubric, not the judge. Fix the criterion before you tune the model.

**Thresholds require distributional thinking.** Do not choose a threshold before you inspect score distributions. A threshold is not just a number; it is a policy decision about acceptable risk. If most scores cluster near the top, a threshold that looks reasonable on paper may

become useless in practice because it does not separate good from bad samples. If the metric is too harsh, it may block acceptable behavior and create review fatigue.

Score distributions tell you whether the rubric is too loose, too strict, or too compressed. They also reveal whether the judge is using the scale in a meaningful way. If nearly all outputs land on one side of the scale, you likely need to revise the rubric or split the criterion into smaller dimensions. Threshold stability matters because a threshold that jitters across small prompt changes is not robust enough for release gating.

For that reason, threshold selection belongs after you have reviewed examples, not before. You want a threshold that corresponds to a visible operational boundary, not one that merely satisfies an arbitrary target percentage.

**Audit the explanation channel separately from the score channel.** If your metric returns reasons, evaluate those reasons as a separate artifact. A good explanation should identify the specific evidence that drove the score and the specific criterion that failed or passed. It should not wander into generic praise or model-like commentary. If the reason string is verbose but uninformative, it adds cost without adding signal.

Reason quality matters because it affects triage time. A metric that can tell you *why* a sample failed is easier to use during regression review than one that only emits a scalar. But do not let a plausible explanation substitute for metric validation. The score can still be wrong even when the reason sounds disciplined. Reason auditing should check whether the explanation actually tracks the rubric, not whether it merely sounds polished.

A practical pattern is to sample several failing and passing cases, read the reasons, and ask whether a human could use them to repair the defect. If not, the metric may be too opaque for operational use.

**Versioning matters because the metric is executable code.**

Custom metrics are not just descriptions; they are executable policies. If the underlying judge model changes, the prompt template changes, or the field mapping changes, the metric changes. That is why version notes matter. A rubric that behaves well under one judge model can shift under another, even when the text of the prompt stays the same.

Treat the metric definition as a versioned artifact. Record the prompt, the evaluation parameters, the pass threshold, and the judge model used during calibration. If you later change any of those pieces, you should expect the score distribution to move. That is not a failure; it is the natural result of changing the measurement instrument.

This is especially important when a library exposes multiple ways to define the same metric. Natural-language criteria and explicit steps are not interchangeable in practice. One may be easier to write, the other easier to debug. Keep the implementation choice visible so you can explain score changes later.

**A robust authoring loop is iterative, not one-shot.** The cleanest way to build a custom metric is to iterate through a short loop. Write the defect statement. Identify the evidence fields. Draft the rubric or evaluation steps. Run the metric on a reviewed slice. Inspect failures, passes, and reasons. Then refine the prompt until the score aligns with the decision you need.

That loop is where G-Eval-style metrics become practical. You are not attempting to encode perfect human judgment. You are building a bounded evaluative instrument with a known failure surface, a clear evidence contract, and a threshold that supports the business decision. The resulting metric is most valuable when it is narrow enough to explain and strict enough to trust.

### 3.4. Metric Composition, Weighting, and Conflict Resolution

A complex LLM system rarely fails in one dimension only. It can be factually correct but unsafe, grounded but incomplete, fluent but wrong on a critical edge case, or excellent on average while breaking one high-risk scenario that matters to your users. That is why composition matters: you are not trying to compute one elegant score that erases all nuance. You are trying to build a decision system that preserves the distinctions engineering teams need in order to act.

A practical composition rule makes the structure visible. Hard constraints gate the run first, while softer preferences contribute only after the sample clears the non-negotiable checks. In code, that usually looks like a blocking metric plus one or more advisory metrics rather than one blended number.

```
blocking = [
    "schema_valid",
    "policy_compliance",
    "no_unsupported_claims",
]

advisory = {
    "helpfulness": 0.5,
    "conciseness": 0.2,
    "tone": 0.3,
}

if all_pass(blocking):
    score = weighted_sum(advisory)
else:
    score = 0.0
```

That pattern is simple on purpose. It forces you to separate release blockers from diagnostic signals, and it prevents a noisy preference score from masking a non-negotiable defect. The exact implementation can vary, but the governance logic should not.

**Start with the decision, not the arithmetic.** Weights only make sense when they support a real decision. If you do not know whether the composite score will gate a release, rank candidates, triage regressions, or drive human review, then the weights are arbitrary. A composite metric is justified only when the consumer needs one summary signal for a specific choice.

That decision-first framing changes how you compose metrics. If the choice is release versus no release, use hard gates for safety, schema validity, policy compliance, or other non-negotiables. If the choice is among several acceptable variants, then weighted scoring may help you rank options. If the choice is diagnostic, keep the metrics separate and inspect disagreement rather than compressing it away.

The wrong habit is to average because averaging feels fair. Averaging can be mathematically tidy and operationally useless. If one metric checks grounding and another checks style, a simple average creates a number that is easy to report and hard to trust. The score no longer says which defect dominated, and the team loses the very signal it needs to fix the system.

**Use gate-then-score as the default composition pattern.** The most robust pattern is layered composition: evaluate hard constraints first, then score softer qualities only if the sample passes. This is not just a convenience for safety-critical systems. It is a general way to preserve meaning. A sample that violates a blocking rule should not be rewarded for a nice tone or a concise answer.

Gate-then-score works because it respects the difference between failure classes. Some defects are disqualifying. Others are preferences. If you blur the two, the composite score becomes politically pleasant and technically misleading. A support bot that gives a polished but unsupported answer should not receive a strong aggregate score merely because it sounds useful. A tool-using agent that chooses the wrong

action should not be redeemed by a well-written final sentence.

This layered structure also simplifies debugging. When a run fails, you know whether the failure came from a hard gate or a weighted preference. That separation is far more actionable than a single opaque scalar.

**Complementary metrics should stay visible.** Metrics interact in three basic ways: they can be complementary, correlated, or conflicting. Complementary metrics illuminate different parts of the system. Correlated metrics may appear to provide independent support while actually measuring nearly the same thing. Conflicting metrics can reveal a real trade-off or a mistake in metric design.

When metrics are complementary, preserve the separation. A grounding metric and an answer-quality metric should usually remain distinct because each observes a different failure surface. One tells you whether the answer is supported; the other tells you whether it addresses the user’s need. If you combine them too early, you remove the ability to tell whether the system is failing on evidence use or on task completion.

A useful rule is to keep atomic metrics aligned to distinct defects, then add a composite only when a decision truly needs a roll-up. That preserves interpretability while still giving management a summary signal when appropriate.

**Correlated metrics can create false confidence.** Two metrics that move together are not necessarily two independent confirmations. They may simply be two views of the same underlying defect. That is common when teams create multiple rubric variants that all measure the same latent notion of “quality.” If the same model judgment drives both scores, the apparent redundancy can look like stronger evidence than it really is.

Correlation becomes dangerous when it hides double counting. Sup-

pose one metric rewards fluency and another rewards answer quality, but both are influenced by surface polish. Then a verbose, elegant but wrong answer can score well on both. The composite inflates confidence without increasing coverage. You have not increased measurement power; you have stacked two partially redundant proxies.

The remedy is conceptual before it is statistical. Ask what each metric sees that the others do not. If the answer is not clear, either simplify the portfolio or rewrite the rubric boundaries. Correlation analysis can help, but only if you first know what the metrics are supposed to distinguish.

**Conflict is not always a problem.** A disagreement between metrics may indicate a bug, but it may also encode a genuine trade-off. A concise answer may score lower on completeness. A grounded answer may be less helpful than a more synthetic one. A safe refusal may look less useful than a direct answer. If you eliminate all conflict by force, you risk flattening meaningful distinctions.

That is why conflict resolution should be rule-based and explicit. Define precedence. Define tie handling. Define escalation thresholds. Define whether uncertainty or disagreement sends the sample to human review. The governance rule should not depend on whichever metric happens to be louder in the aggregate.

For example, a legal assistant may treat citation discipline as a hard gate while allowing tone and brevity to trade off within a weighted score. A customer-support copilot may do the opposite: task completion can be weighted heavily, but any policy violation escalates to review. The right conflict rule depends on business risk, not on metric convenience.

**Balanced-position evaluation matters for pairwise judges.** If you use pairwise judges, answer order matters. The judge may prefer the first or second candidate because of position bias, verbosity bias,

or framing effects. A robust composition rule therefore evaluates both answer orders and aggregates them before finalizing the preference.

The simplest balanced-position protocol is to score the pair twice, swapping the order on the second pass, then combine the two outcomes. If the judge prefers A over B in one order and B over A in the other, that disagreement itself is informative. You can treat it as uncertainty, a tie, or a trigger for human review depending on the stakes.

This matters for composition because an uncorrected pairwise bias can leak into downstream weighting. If one candidate consistently occupies the preferred slot, the aggregate metric will reward position rather than quality. Balanced ordering removes that confound and keeps the preference signal anchored to the content.

**Weighting should reflect policy, not taste.** Weights are policy statements. They say which qualities matter more when several acceptable outputs compete. If you cannot explain why one dimension deserves more weight than another, the composite is not grounded in operational reality.

A good weighting scheme is usually asymmetric. Hard constraints get full precedence. Business-critical properties get large weight. Cosmetic qualities get smaller weight. But avoid hiding risk in soft weights when the underlying property is non-negotiable. Safety, legality, and schema validity do not belong in the same linear sum as tone or brevity.

If you must use weights, keep them coarse and interpretable. The goal is not to optimize a machine-learning loss function. The goal is to capture the organization's preference order in a way that can survive review. In practice, a small set of weights is easier to justify and maintain than a finely tuned vector whose rationale nobody can explain a month later.

**Do not average away uncertainty.** Aggregating noisy judge scores

can create a false sense of precision. Averages compress disagreement, but the disagreement may be the most important part of the signal. If one metric is uncertain and another is stable, a blind average treats them as equally trustworthy. That is rarely what you want.

Instead, preserve uncertainty explicitly. If a metric has a narrow margin near the threshold, escalate. If pairwise judges disagree across answer order, mark the sample as unstable. If two dimensions conflict in a way that signals a genuine trade-off, keep the disagreement visible in the report. A composite score that hides uncertainty is often less useful than a dashboard that shows separate sub-scores and the rule used to combine them.

This is especially true for judge-based metrics, where score variance can come from prompt sensitivity, model stochasticity, and hidden bias. Averaging repeated runs can smooth some noise, but it can also bury systematic error. Use repeated measurements to estimate stability, not to pretend the instability never existed.

**Separate diagnostic metrics from release blockers.** A metric that helps you understand a failure is not automatically suitable as a release gate. Diagnostic metrics can be broad, subjective, or scenario-specific because their job is to explain behavior. Release blockers must be stable, narrow, and aligned with policy because their job is to protect the system boundary.

That distinction changes every decision downstream. Diagnostic metrics can tolerate more ambiguity if they improve triage. Blocking metrics need tighter thresholds and stronger calibration. Diagnostic metrics can remain correlated if they help analysts. Blocking metrics should avoid redundancy because redundant gates are hard to reason about and easy to over-enforce.

A practical governance rule is to label each metric explicitly as *block*, *rank*, or *diagnose*. “Block” means the sample cannot pass. “Rank”

means the metric contributes to selection among acceptable outputs. “Diagnose” means the metric explains behavior but does not directly determine the decision. Once those roles are fixed, the composition logic becomes much easier to defend.

**Inspect disagreement cases before you revise weights.** When metrics disagree, resist the urge to smooth the conflict with a new weight vector. First inspect the examples. The disagreement might reveal a true product tension, a threshold problem, or a rubric overlap. If a grounded but terse answer loses to a longer but less supported one, you may need a gate for support and a separate scorer for helpfulness. If a safe refusal loses to a direct answer, you may have assigned too much weight to completion. If two metrics disagree on the same samples repeatedly, one of them may be measuring the wrong thing.

Disagreement analysis is also how you tell expected conflict from pathological conflict. Expected conflict is part of the design. Pathological conflict points to a broken rubric, a biased judge, or a composite that bundles incompatible requirements. The right response depends on which of those you find, not on the raw disagreement rate alone.

**Threshold layering is usually better than a monolithic cutoff.** One composite threshold is attractive because it is easy to explain, but it often hides too much structure. Threshold layering gives you finer control. For example, a sample may need to pass schema validity, then safety, then grounding, and only then contribute to a weighted quality score. Different failure modes get different handling because they carry different operational risk.

Layered thresholds are also easier to tune. If the quality score is too harsh, you can adjust the advisory layer without weakening the gate. If the safety gate is too permissive, you can tighten it without affecting the ranker. A monolithic score forces you to move everything at once, which is the opposite of disciplined governance.

The most practical layering scheme is simple: mandatory checks first, then major quality dimensions, then advisory refinements. That ordering matches the way teams actually make release decisions.

**Conflict resolution should be rule-based and documented.**

When a sample triggers competing signals, the resolution rule must already exist. Do not let reviewers improvise a policy from case to case. A documented rule might say that any safety failure overrides all other metrics, that a tie in pairwise ranking escalates to human review, or that a narrow disagreement between two advisory metrics is acceptable unless the sample is near a release boundary.

Explicit rules protect you from post hoc rationalization. They also make regressions easier to interpret because you can tell whether the system changed or the policy changed. If the rule is implicit, the same evidence can lead to different decisions depending on who reads the dashboard.

A good rule has four parts: precedence, tie handling, escalation, and exception criteria. Precedence determines which metric wins when they conflict. Tie handling defines what to do when scores are too close to call. Escalation specifies when to involve a human. Exception criteria describe scenario types where the default policy should not apply. That structure keeps the composition logic operational rather than ad hoc.

**The right composite is often a small one.** There is no prize for squeezing every observed dimension into a single scalar. A compact portfolio is usually easier to maintain than a sprawling one. If you can make a decision with one gate and two or three interpretable advisory metrics, that is often better than a high-dimensional scorecard whose weights nobody trusts.

Small composites are also easier to recalibrate. When the product changes, you can inspect each metric's distribution and adjust the policy without refitting an opaque summary. That keeps the system hon-

est. It also forces the team to confront which quality dimensions actually drive the decision, rather than hiding behind a glamorous aggregate number.

The end state is not metric minimalism. It is metric discipline. You keep the metrics that observe distinct failures, you weight them only where policy requires a trade-off, and you preserve disagreement whenever it is revealing. The result is a decision system that can support engineering action without pretending that a single number captures the whole truth.

## Chapter 4

# Engineering Trustworthy RAG Evaluation

*RAG systems often look impressive right up until a production incident reveals that nobody can tell whether the failure came from retrieval, context packing, or generation. This chapter shows how to build evaluation evidence that preserves those boundaries, how to read core RAG metrics without fooling yourself, and how to design benchmark suites that expose the actual subsystem that needs repair.*

### **4.1. Evaluating RAG as a Decomposed Retrieval-to-Answer Pipeline**

A RAG answer can look polished while the system underneath is failing in three different places: the retriever may have missed the best evi-

dence, the packing layer may have dropped or diluted that evidence, or the generator may have ignored it and filled the gap with plausible text. If you only log the final answer, those failures become indistinguishable. A decomposed evaluation model avoids that collapse by treating retrieval, context assembly, and synthesis as separate transformations with separate evidence. That is the operating model for trustworthy RAG evaluation: each stage leaves an artifact, and each artifact supports a different kind of judgment.

A practical DeepEval-style test case makes those boundaries explicit. The minimum useful fields are the user input, the raw retrieval output, the packed context that the model actually saw, and the generated answer. If you can also provide an expected output or a structured reference, you gain an anchor for some metrics, but you should treat that anchor as a diagnostic aid rather than a universal truth source. The important point is that these fields are not redundant copies of the same information; they capture different control points in the pipeline.

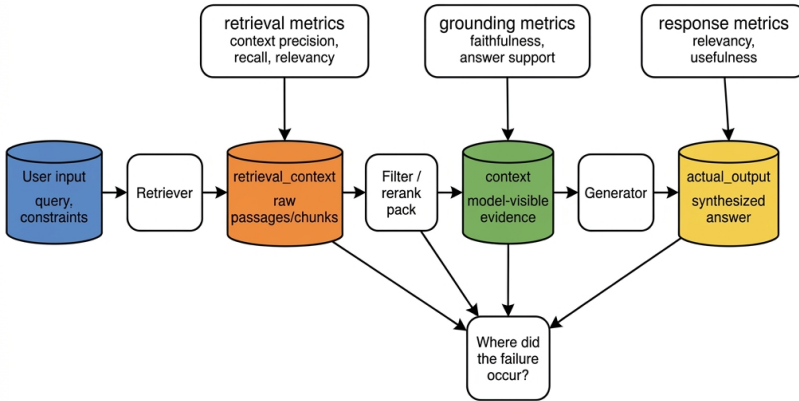
```
from deepeval.test_case import LLMTestCase

case = LLMTestCase(
    input="What was the main cause of the outage?",
    retrieval_context=[
        "Postmortem: DNS records expired at 02:14 UTC.",
        "A cache stampede followed the DNS failure."
    ],
    context=[
        "Postmortem: DNS records expired at 02:14 UTC."
    ],
    actual_output="The outage was caused by expired DNS records.",
    expected_output="Expired DNS records caused the outage."
)
```

That shape is useful because it separates what the retriever surfaced from what the generator received. `retrieval_context` is evidence of retrieval behavior: the candidate passages, chunks, or documents returned by search. `context` is evidence of packaging behavior: the subset that survived reranking, filtering, truncation, prompt assembly, or

tool-specific formatting. `actual_output` is evidence of synthesis behavior: the final claim set produced by the model. `expected_output` is a reference point, but in retrieval-augmented tasks it must be handled carefully because an answer can be partially correct, contextually grounded, or intentionally abbreviated without matching the reference text verbatim.

Those distinctions matter because each stage can fail independently. If retrieval finds the right passage but packing removes it, a faithfulness metric may flag the answer as unsupported even though the retriever behaved well. If packing is sound but the model answers from prior knowledge, the answer may look fluent and even factually correct while still violating the evidence contract. If the answer is irrelevant, the issue may not be grounding at all; it may be query interpretation, prompt framing, or refusal behavior. The evaluator's job is to preserve enough evidence to assign the failure to the correct subsystem.



The diagram captures a boundary that is easy to blur in production: `retrieval_context` is not the same as `context`. The first is the evidence pool; the second is the evidence actually consumed by the generator. If you do not persist both, you lose the ability to tell whether a failure came from retrieval quality or from downstream packing. In practice, that boundary is often where engineering mistakes hide. Teams log the raw top- $k$  results, then later evaluate faithfulness against the final prompt, but they forget that the prompt may have been truncated, re-ordered, or compressed. Once that happens, a model can be blamed for unsupported claims that never had a fair chance to be grounded.

This is why the evaluation flow should mirror the pipeline itself. Start by capturing the query exactly as issued. Preserve the raw retrieval results before any filtering. Preserve the packed context after all transformations that influence what the model can see. Only then evaluate retrieval-oriented metrics against the raw retrieved set, grounding-oriented metrics against the packed context, and response-oriented metrics against the final answer. That ordering is not bureaucratic overhead; it is what keeps the score interpretable.

Once the evidence is separated, failure localization becomes much sharper. Suppose the answer is correct, but the supporting passage appears low in the retriever output and only survives because the packer keeps a wide context window. Retrieval precision is weak, even though the end-to-end answer looks good. That tells you the current pipeline may work under generous token budgets, yet remain brittle under load, cost pressure, or slightly different queries. Now suppose the retriever returns strong passages, the packed context contains them, and the answer still introduces an unsupported claim. That is a synthesis fault, not a retrieval fault. The generator saw the evidence and failed to stay within it.

A different failure mode appears when the retriever finds useful evidence, but the packer discards it to make room for distractors or because of an overly aggressive token limit. In that case, the model may generate a faithful answer relative to the context it received, but the answer is still incomplete or wrong relative to the task. This is one reason reference-free grounding metrics alone are not enough for diagnosis. A faithful answer can still be an artifact of impoverished context. You need both the retrieval layer and the packed-context layer to know where the evidence disappeared.

The opposite case is equally dangerous: a model can return a highly relevant answer for the wrong reasons. It may match the question well, use the right terminology, and satisfy a human skim test, yet its claims

may be unsupported by the retrieved material. This usually happens when the model relies on parametric knowledge, memorized patterns, or generic domain priors instead of the provided context. End-to-end evaluation may miss that distinction entirely unless you preserve the context boundary and inspect support claim by claim.

The role of `expected_output` deserves special care. In open-ended RAG tasks, a reference answer is not the same thing as ground truth in the traditional supervised-learning sense. It may be a concise paraphrase, a partial answer, or one valid rendering of several acceptable answers. That makes it useful for comparing omission and relevance, but not sufficient for evaluating support. If you force exact-match thinking onto retrieval-backed QA, you will reward stylistic similarity over grounded correctness. You will also undercount answers that are correct but phrased differently from the reference. For that reason, `expected_outputs` work best as structured anchors: they help define the target intent, but they do not replace evidence-based judgment.

In systems with strong retrieval guarantees, the most useful question is not simply whether the final answer is right. It is where the evidence contract broke. Did the retriever fail to surface the decisive passage? Did the packer exclude it? Did the generator ignore it? Or did the model answer conservatively because the context was incomplete but still sufficient for a partial response? Those distinctions matter operationally because they point to different fixes. Retrieval faults suggest index tuning, query rewriting, reranking, or corpus coverage work. Packing faults suggest token budgeting, prompt assembly, and truncation strategy. Synthesis faults suggest instruction tuning, judge calibration, or response-format constraints.

A clean decomposition also exposes anti-patterns that otherwise look harmless. Logging only the final answer removes the evidence needed for root-cause analysis. Conflating retrieved documents with supplied context makes packing bugs invisible. Evaluating with goldens that

can be answered from common world knowledge without any context makes the retrieval path optional, so retrieval metrics become decorative. And using a single aggregate score to gate releases encourages teams to optimize whichever subsystem is easiest to move, not the subsystem that actually failed.

The right evaluation discipline therefore treats each stage as a contract. The retriever must return candidates that contain the needed evidence. The packer must preserve that evidence in a form the generator can use. The generator must stay within the support available in the packed context unless the application explicitly allows augmentation from external knowledge. A RAG test case is the record of those contracts: query, retrieved evidence, packed evidence, and response. Once you keep those records intact, metric disagreement becomes informative instead of confusing. Low retrieval precision with strong answer faithfulness points to an over-inclusive retriever that still happens to feed the generator enough useful context. Low faithfulness with strong retrieval metrics points to generator drift. Low recall with a correct answer can indicate a conservative system that succeeded only because the question was easy or the missing evidence was not actually needed.

That decomposition is the foundation for the rest of the chapter. The response-layer metrics make sense only when the context boundary is preserved, and the retriever-facing metrics make sense only when the raw retrieval set remains visible. When you keep those boundaries intact, each score becomes an explanation of a specific subsystem, not a vague verdict on the whole application.

## 4.2. Measuring Answer Relevancy and Faithfulness in Grounded Generation

A polished answer can still fail in two very different ways: it can miss the user's actual request while staying perfectly within the retrieved evidence, or it can answer the request directly while drifting beyond what the evidence supports. That distinction is easy to miss when you only inspect the final text. For grounded generation, you need two response-layer questions answered separately: does the response address the task, and does the response stay supported by the supplied context? Answer relevancy and faithfulness give you those two axes.

A minimal DeepEval-style test case makes the separation concrete. The response-layer evaluator should see the user input, the packed context the generator actually consumed, and the generated answer. If you also have an expected output, treat it as a reference for task intent, not as a universal proxy for truth. That structure lets you ask different questions of the same response without collapsing task alignment into grounding.

```
from deepeval.test_case import LLMTTestCase

case = LLMTTestCase(
    input="Which subsystem caused the outage?",
    context=[
        "Postmortem: the DNS cache expired at 02:14 UTC.",
        "The retrieval layer found the incident report."
    ],
    actual_output="The outage was caused by expired DNS records.",
    expected_output="Expired DNS records caused the outage."
)
```

The first metric, answer relevancy, asks whether the response addresses the user's request, constraints, and information need. It is a task-alignment check, not a factuality check. A response can be relevant even if it is wrong, because it focuses on the right subject and speaks to the right intent. It can also be irrelevant even if every sen-

tence is technically supported by context, because it answers the wrong question, omits the requested scope, or dwells on a related but unasked detail.

Faithfulness asks a different question: are the claims in the response justified by the supplied context? That makes it a grounding check. DeepEval’s faithfulness metric is referenceless with respect to a gold answer, but it is not referenceless with respect to evidence. It still depends on the context field, which means it evaluates support, not worldly correctness. That is a crucial distinction. A response can be true in general and still fail faithfulness if the retrieved context does not support it. Likewise, a response can be faithful to the context and still be incomplete or unhelpful.

### **Why the two axes diverge**

The divergence is not a corner case; it is the normal failure surface of RAG. Suppose the context contains a postmortem that attributes the outage to expired DNS records, and the model answers, “The outage came from an expired DNS entry, and the team should also audit certificate rotation.” The first clause may be faithful, but the second clause is unsupported unless the context mentions certificates. The response is relevant and partly grounded, yet it crosses the support boundary. Now reverse the case: the model replies, “The incident appears related to infrastructure configuration,” which is safe and supported, but too vague to answer the user’s actual question. That answer is faithful but weakly relevant.

The operational value of splitting the metrics is that each one points to a different corrective action. Low relevancy with strong faithfulness often means the retriever supplied adequate evidence, but the generator failed to synthesize it into an answer that matched the user’s request. You may need a better prompt, a different answer template, or a stronger instruction to address the question directly. Strong relevancy

with weak faithfulness often means the model is overconfident or over-generalizing from context. You may need stricter grounding instructions, answer citation requirements, or a refusal policy when evidence is incomplete. When both are low, the system likely missed the task and the evidence simultaneously.

### **How judge-based scoring works**

Both metrics are often implemented with LLM judges, but they inspect different latent properties. For relevancy, the judge assesses whether the response covers the user's intent. Ragas operationalizes this by reverse-generating questions from the response and comparing those synthetic questions with the original query in embedding space. That approach captures whether the answer is about the right thing, not merely whether it repeats query terms. It is especially useful when the answer is paraphrased or when the user's request is broad enough that exact lexical matching would be too brittle.

Faithfulness usually follows a claim-support pattern. The evaluator extracts atomic claims from the response, then checks whether the supplied context entails or supports each claim. You can think of this as decomposing the answer into propositions and asking whether each proposition is licensed by the evidence. The granularity matters. If you evaluate faithfulness only at the paragraph level, a single unsupported detail can be hidden inside an otherwise good answer. Claim-level checking exposes exactly which statement crossed the line.

That claim-level structure also explains why faithfulness is not the same as correctness against the world. If the context is incomplete, the model may answer conservatively and remain faithful while still being wrong or incomplete. If the context is wrong, the model can be faithful to bad evidence and still produce a mistaken answer. In other words, faithfulness measures adherence to the provided evidence, not absolute truth. For RAG engineering, that is usually the right abstraction,

because the system’s contract is to ground output in context.

**A useful interpretation matrix**

A two-axis mental model helps you read the scores without overfitting to a single number.

|                                          |                                                                                                                                                                                        |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>High relevancy, high faithfulness</b> | The answer addresses the request and stays within evidence. This is the target state for grounded QA.                                                                                  |
| <b>High relevancy, low faithfulness</b>  | The answer sounds useful and on-topic, but introduces unsupported claims. This often indicates synthesis drift or overreliance on prior knowledge.                                     |
| <b>Low relevancy, high faithfulness</b>  | The answer is grounded but misses the user’s intent, answers only part of the question, or summarizes the wrong passage. The generator is behaving conservatively but not effectively. |
| <b>Low relevancy, low faithfulness</b>   | The answer neither addresses the task nor respects the evidence boundary. This usually signals a larger prompt, retrieval, or orchestration failure.                                   |

The matrix is useful because it prevents a common mistake: treating any good-looking answer as a success. Style can mask both task misalignment and unsupported elaboration. A fluent response can score well in casual review and still fail either axis. You need the pair of metrics to separate cosmetic quality from diagnostic quality.

**Thresholds and operational policy**

The right threshold depends on the application, not the metric alone. In low-risk exploratory search, you may accept modest relevancy with moderate faithfulness if the system is designed to help users discover leads rather than finalize decisions. In high-stakes support or regulated settings, you usually want a stricter faithfulness threshold, even if that increases abstentions or shorter answers. That is a product decision, not just an evaluation decision.

The trade-off is between completeness and restraint. If you optimize for completeness, the model may fill gaps with plausible but unsupported material. If you optimize for restraint, the model may answer cautiously and omit useful inference. The right balance depends on

the cost of false positives versus false negatives. When unsupported claims are expensive, faithfulness should dominate. When omission is the bigger risk, relevancy and coverage matter more, but only if the evidence can still be justified.

The same policy choice affects how you interpret low scores. A low faithfulness score is not always a failure of the generator; it can be a sign that the system correctly refused to invent details. A low relevancy score is not always a failure of understanding; it can mean the answer was intentionally narrow because the context did not support a broader claim. Good evaluation practice distinguishes these cases by reading the judge’s rationale and by inspecting the claims the answer actually made.

### **Inspecting judge reasons without over-trusting them**

Reason strings from judge-based metrics are useful, but they are not ground truth. Use them as debugging traces. If a relevancy judge says the response “talks about the right domain but does not answer the query,” check whether the answer really omitted the requested action, entity, or constraint. If a faithfulness judge says a claim is unsupported, look for the exact sentence that introduced it and verify whether the context contains evidence in a form the judge might have missed.

That last point matters because context formatting affects judge behavior. A support-checking model may fail to connect a claim to a passage if the relevant sentence is buried in a long paragraph, split across chunks, or obscured by distractors. You should not treat a single judge explanation as final proof of failure. Instead, compare the judge rationale to the context and look for clear alignment. If the rationale is vague, repeatable, or inconsistent across similar cases, that is a sign to calibrate the judge prompt or scoring model.

The same caution applies to relevancy. A judge may overvalue lexical overlap or paraphrase similarity and underweight whether the answer

actually resolves the user’s information need. That is why answer relevancy should be read as a task-fit signal, not as a semantic similarity score in disguise. The target is not mere topicality; it is whether the response performs the requested job.

### **Common failure patterns**

One frequent pattern is the grounded-but-unhelpful answer. The model stays inside the evidence but answers narrowly, generically, or partially. This often appears when the generator is over-penalized for unsupported statements and learns to minimize risk by producing cautious summaries. It can also happen when the context is relevant but poorly organized, so the model extracts the wrong emphasis from the right material.

The opposite pattern is the useful-but-unsupported answer. The response directly addresses the question and may even be exactly what a human wanted, but it introduces claims not present in the supplied context. In consumer-facing settings, this can look like good product behavior. In engineering terms, it is a boundary violation. If the system is supposed to ground its output in retrieved evidence, a good guess is still a defect.

A third pattern is partial correctness. The answer contains a supported core and one or two unsupported embellishments. That is precisely where claim-level faithfulness matters. Aggregate judgments can miss these mixed cases, especially when a long answer contains enough correct material to fool a coarse score. Fine-grained support checking catches the unsupported tail before it becomes an incident.

### **A compact debugging sequence**

When a response fails, inspect it in this order.

- Check relevancy first. Ask whether the answer addresses the

user's actual request, not just the same topic.

- Check faithfulness next. Mark each unsupported claim and compare it directly with the context the model received.
- If relevancy is low but faithfulness is high, look for prompt framing, answer format, or task interpretation problems.
- If faithfulness is low but relevancy is high, look for synthesis drift, overgeneration, or weak grounding instructions.
- Escalate to retrieval diagnosis only if the needed evidence is absent from the packed context.

That sequence keeps the blame assignment clean. It also prevents a common debugging error: jumping straight to retriever tuning when the generator simply ignored evidence that was already available. The reverse error is just as common: blaming the generator for an unsupported answer when the context never contained the decisive facts. A response-layer check lets you separate those cases before you touch the index.

### **What not to use these metrics for**

Do not use faithfulness as a substitute for correctness when the context itself may be wrong, stale, or adversarial. A faithful answer can still reflect bad evidence. Do not use answer relevancy as a substitute for user satisfaction, because a response can be on-topic and still miss the user's real intent, tone, or required depth. Do not use either metric as a single release gate unless you have already calibrated them on representative failures from your own workload.

The strongest practice is to report them together, with context attached, and to keep the score rationale visible in your evaluation trace. That gives you a response surface rather than a single scalar. Once you can see that surface, you can tell the difference between a model that is

merely fluent, one that is merely cautious, and one that is both useful and grounded.

### 4.3. Interpreting Contextual Precision, Recall, and Relevancy for Retriever Diagnosis

A retrieval layer can fail in three different ways at once: it can miss the needed evidence, rank the useful evidence too low, and fill the context window with noise that burdens the generator. If you only inspect the final answer, those defects collapse into one opaque failure. Retrieval-oriented metrics separate them again. Contextual precision tells you whether the retrieved set is mostly useful and well ordered. Contextual recall tells you whether the set actually contains the evidence needed for a correct answer. Contextual relevancy tells you whether the retrieved material is broadly aligned with the query or task. Read together, they tell you whether the retriever is missing evidence, wasting budget, or surfacing the wrong material entirely.

A practical test case should preserve the raw retrieved set, because the diagnosis depends on what the retriever returned before any reranking, filtering, or prompt packing changed it. If the retriever surfaces candidate passages, you want to evaluate that set directly, not the postprocessed prompt alone. A DeepEval-style structure keeps that separation visible: the retrieved candidates form the raw retrieval evidence, while the packed context belongs to the generator-facing boundary. The metrics in this section focus on the first boundary.

```
from deepeval.test_case import LLMTestCase

case = LLMTestCase(
    input="What caused the outage?",
    retrieval_context=[
        "DNS records expired at 02:14 UTC.",
        "A cache stampede followed the outage."
    ],
    context=
```

```
        "DNS records expired at 02:14 UTC."
    ],
    actual_output="The outage was caused by expired DNS records."
)
```

That distinction is useful because retrieval quality is not one number. A retriever can find the right passage and still be weak if it buries that passage beneath distractors. It can return a focused set and still be weak if the evidence you need is absent. It can retrieve useful material and still be operationally poor if it floods the prompt with low-value text that steals attention from the answer generator. Contextual precision, recall, and relevancy expose those different failure surfaces.

### **Contextual precision as a ranking and noise signal**

Contextual precision measures how much of the retrieved material is actually useful, with extra weight on rank position when the metric implementation is order-sensitive. In practice, you care most about the top of the list, because top- $k$  chunks consume the limited token budget that the generator sees. A highly precise retrieval set places the useful passages early and minimizes irrelevant material. A low-precision set may still contain the right evidence somewhere, but it forces the generator to sift through noise or it displaces better passages from the prompt.

The ranking sensitivity matters because retrieval is often used under a hard context budget. A passage that appears at rank one can shape the answer, while a passage at rank fifteen may never be seen. That is why contextual precision is usually more diagnostic than a simple binary hit count. If the set contains one relevant passage and nine irrelevant ones, the system technically found something useful, but it also signaled that the ranking function cannot reliably separate evidence from distraction. When the answer quality later drops, the retrieval trace often shows that the relevant passage was available but too low to matter.

Precision also helps you distinguish two different retrieval faults. If precision is low but recall is acceptable, the retriever can usually find the right evidence, yet it struggles to rank or filter it. That pattern points to reranking, hybrid retrieval tuning, query rewriting, or chunking defects. If both precision and recall are low, the issue is broader: the retriever is neither finding the right evidence nor keeping the distractors out.

### **Contextual recall as a coverage signal**

Contextual recall asks whether the retrieved set covers the facts required to answer the question or support the reference claims. It is omission-sensitive rather than ordering-sensitive. A high-recall set may include some irrelevant material, but it still gives the generator access to the evidence needed for a grounded answer. A low-recall set means some necessary information never made it into the retrieved candidates at all.

That makes recall the right lens for absence. If an answer fails because the necessary passage never appeared, no amount of prompt engineering downstream can recover it reliably. You need a broader search, a different indexing strategy, or better corpus coverage. Recall is therefore the metric that tells you whether the evidence supply is complete enough to support the task.

Ragas supports multiple recall modes, and that flexibility is worth preserving in your mental model. When you have a reference answer, LLM-based recall can decompose that answer into claims and ask whether the retrieved context supports them. When you have reference contexts, non-LLM or ID-based variants can measure whether the expected documents or passages were retrieved. The implementation choice changes the evidence source, but the diagnostic meaning stays the same: did the retriever surface the pieces needed for correctness? The answer matters because a system can look competent under one

variant and fragile under another, especially when chunking or document identity is unstable.

Recall also reveals an important asymmetry. A retriever may have high recall even if the top-ranked results are not especially sharp. If the needed passage appears somewhere in the retrieved set, the metric can still look healthy while the generator struggles with noise. That is why recall must be read together with precision, not as a substitute for it.

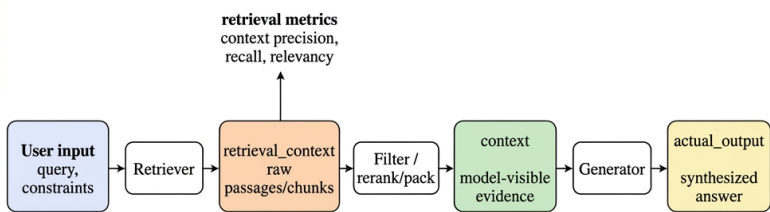
### **Contextual relevancy as a broader alignment signal**

Contextual relevancy evaluates whether the retrieved material is generally useful for the query or task. It is a more permissive signal than labeled precision or recall, especially in referenceless settings. That makes it helpful for broad monitoring, but weaker for root-cause diagnosis. If you only know that retrieved material is “relevant,” you still do not know whether the retriever ranked it well or whether the set omitted a key fact.

This difference matters in production because contextual relevancy can remain high even when precision and recall split apart. A topically aligned retrieved set may still be too noisy, too shallow, or too incomplete for the generator to answer correctly. Conversely, a modestly relevant set may still be sufficient if it contains the decisive facts early and cleanly. Relevancy is therefore best read as a coarse alignment check: does the retrieval step generally point in the right direction? The finer question of whether it did so efficiently and completely belongs to precision and recall.

In referenceless evaluation, contextual relevancy becomes especially attractive because it does not require explicit labels. That convenience comes at a cost. Without reference passages or answer claims, the metric cannot tell you what was missed, only whether the retrieved set seems broadly aligned. Use it for continuous monitoring when label availability is limited. Use precision and recall when you need to ex-

plain the failure.



The diagram captures the boundary that matters here. Retrieval metrics attach to the raw `retrieval_context`, not to the postprocessed context that the generator sees. If you blur those two artifacts, you can no longer tell whether a precision drop came from retrieval ranking or from packing. That confusion is common in production logs, where teams record only the final prompt and then try to infer retriever quality from it. The result is a false diagnosis whenever reranking, truncation, or filtering changes the evidence set.

## Reading the three metrics together

The diagnostic power comes from disagreement among the metrics. High recall with low precision means the retriever can find useful evidence, but it cannot keep noise under control. That usually points to ranking defects, insufficient reranking, or chunking that produces too many quasi-relevant fragments. High precision with low recall means the retriever is selective but misses necessary evidence. That pattern suggests narrow query coverage, weak synonym handling, corpus gaps, or a top- $k$  setting that is too small for the task.

If relevancy is high while precision and recall disagree, the retriever is probably capturing the general topic but failing at the operational details. That can happen when the corpus is semantically aligned but the chunk boundaries do not preserve the exact support needed for a precise answer. It can also happen when query rewriting broadens the search enough to find useful material, but not enough to surface all decisive facts.

Low relevancy with moderate precision or recall indicates a broader mismatch. The retriever may be returning technically related passages, but they do not match the user's actual intent. In those cases, query understanding or corpus segmentation may be more at fault than ranking. A question about an incident root cause is not well served by passages that merely mention the incident; the retrieval target must match the informational task, not just the topic.

## How chunking distorts the signal

Chunk size and overlap strongly influence all three metrics, which is why rigid chunk-level interpretation can mislead you. Small chunks improve locality and can increase apparent precision because each chunk is narrowly focused. They also risk recall loss if the evidence needed for a claim is split across boundaries. Large chunks improve the chance that a complete argument or fact set stays intact, but they often reduce

precision by pulling in adjacent noise. Overlap can mitigate boundary loss, yet it also creates duplicate near-duplicates that inflate apparent retrieval volume without improving coverage.

That trade-off is not a theoretical nuisance; it changes how the metrics should be read. If precision drops after you increase overlap, the retriever may not have become worse. It may simply be surfacing more redundant but still semantically related chunks. If recall drops after shrinking chunks, the retriever may still be finding the right document, but the evidence is now fragmented across passages that no longer count as a clean hit under your evaluation scheme. The metric is reporting the interaction between retrieval and representation, not just the quality of the search algorithm.

This is why retrieval evaluation should respect the relevant unit of evidence. In some corpora, a paragraph is the right unit. In others, a document section, a table row, or an entire note is the meaningful atomic unit. If you choose the wrong unit, you can punish a good retriever for the way the corpus was segmented. The metric is only as honest as the evidence definition behind it.

### **Top- $k$ , reranking, and the token budget**

The top- $k$  choice links ranking quality to context pollution. A larger  $k$  can improve recall because it gives the retriever more chances to surface the right passage. But every extra item increases the chance of distractors and consumes context budget that the generator could have used for more useful evidence. In other words, recall and precision move in opposite directions once you push beyond the most relevant results.

Reranking changes the same trade-off by altering which passages survive into the prompt. A good reranker can lift precision without sacrificing too much recall, but only if the candidate pool already contains the needed evidence. If the candidate pool is incomplete, reranking

cannot recover what never appeared. That is why reranking should be evaluated as a ranking improvement, not as a substitute for corpus coverage.

The practical decision rule is simple. Increase candidate breadth when recall is the bottleneck. Tighten ranking when precision is the bottleneck. Use the token budget to preserve the passages that matter most to the generator, not to maximize the raw number of retrieved items. If a small set of passages answers the question cleanly, a large noisy set can make the final model less reliable, not more.

### **Anti-patterns that distort retriever diagnosis**

A common mistake is to treat low recall as proof that the retriever is broken. Sometimes the corpus simply does not contain the needed evidence. In that case, the correct diagnosis is a data coverage problem, not a retrieval failure. Another mistake is to assume that reranked context is equivalent to raw retrieval output. Reranking changes what the generator sees, so you must preserve both the original candidate set and the final packed context if you want to localize the defect accurately.

Another anti-pattern is building benchmarks whose answers can be produced from general world knowledge alone. If the question is easy to answer without retrieval, then a high answer score tells you little about retrieval quality. The system may look strong because the generator already knew the answer. In that setting, contextual precision and recall can be artificially low-value metrics because the task does not force the evidence path to matter. Good retrieval benchmarks make the answer contingent on the retrieved material.

### **A compact interpretation matrix**

You can read the metrics as a small decision table.

|                         |                                                                                                                                         |
|-------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>Precision down</b>   | Noise or ranking weakness. The retriever returns too much irrelevant material, or it fails to put the best passages near the top.       |
| <b>Recall down</b>      | Coverage weakness. The retriever misses necessary evidence, or the evidence is split in a way the evaluation unit cannot capture.       |
| <b>Relevancy down</b>   | Broad task mismatch. The retriever is returning material that is related in topic but not aligned with the query intent or answer task. |
| <b>Metrics disagree</b> | Likely chunking, packing, or query-shaping issue. The retrieval step may not be the sole bottleneck.                                    |

That matrix is useful because it turns a score report into an action plan. A precision problem suggests ranking and filtering work. A recall problem suggests broadening search or repairing corpus coverage. A relevancy problem suggests the query is not being matched to the right evidence class. When the metrics disagree, do not jump directly to retriever tuning; inspect chunking and context assembly first.

**Choosing the right metric for the right question**

Use contextual precision when you want to know whether the retriever is efficiently ranking useful evidence near the top. Use contextual recall when you want to know whether the necessary evidence is present at all. Use contextual relevancy when you want a broader, lower-cost view of whether the retrieved set is on task. Do not expect any one of them to localize every failure. They work as a bundle because retrieval quality is multi-dimensional.

That bundle becomes especially valuable when you compare runs over time. A change that improves recall may hurt precision, and the answer quality can still worsen if the generator becomes overloaded with distractions. A change that improves precision may hurt recall, especially if the retriever becomes too conservative. The right interpretation depends on the workload, the corpus, and the cost of missing evidence versus surfacing noise. In practice, retriever diagnosis is not about finding a single best score. It is about understanding which evidence failure your system can tolerate and which one it cannot.

## 4.4. Building RAG Benchmark Suites That Expose Root Causes

A RAG benchmark only earns its keep when it tells you *why* a system failed. A flat score that says an answer was bad is operationally weak: you still do not know whether the retriever missed the evidence, the packer dropped it, the generator ignored it, or the prompt encouraged unsupported synthesis. A useful suite therefore has to encode the failure surface itself. It must stratify cases by the subsystem under stress, preserve the evidence structure needed for diagnosis, and make metric disagreement informative rather than confusing.

That design goal changes how you build goldens. A RAG golden is not just a question and a preferred answer. It is a small diagnostic object that includes the question, the evidence expected to matter, the distractors that should not matter, and the answer form that should be accepted. In practical terms, you want to know whether a case tests retrieval miss, ranking error, context pollution, chunk-boundary fragmentation, multi-hop synthesis, or long-context compression. If a benchmark cannot distinguish those failure classes, it will blur engineering decisions instead of guiding them.

**Start from failure hypotheses, not from domain labels** The strongest benchmark suites begin with explicit hypotheses about what can go wrong. If you want to test chunking, create cases where the decisive fact straddles a boundary. If you want to test ranking, create cases where one relevant passage competes with many near-miss distractors. If you want to test synthesis, create cases where the answer requires combining two or more passages that are each individually insufficient. If you want to test abstention, create cases where the corpus does not support a confident answer and the correct behavior is to decline or qualify.

This approach is more diagnostic than grouping cases by topic alone. A suite with *finance*, *support*, and *legal* buckets can still be opaque if each bucket mixes lookup, multi-hop reasoning, and missing-evidence cases without annotation. By contrast, a suite stratified by failure surface lets you ask finer questions: did recall improve only on direct lookup cases, or did it also improve on multi-hop synthesis? Did a new reranker reduce noise but hurt coverage on distractor-heavy cases? Did a prompt change reduce hallucination on answerable questions while increasing abstention on partial-evidence tasks? Those are the questions that drive engineering.

**Encode the evidentiary structure in each case** A good RAG test case describes not only the answer target but also the evidence geometry. You need to know whether the answer depends on one document, several documents, a specific passage, or a claim that must be assembled across fragments. You also need to know whether the corpus contains nearby distractors that look plausible but should not count as support. That annotation lets you interpret metric disagreement correctly.

For example, a direct lookup case should usually have one clearly relevant source, a small set of controlled distractors, and a concise answer. A multi-hop case should require two or more supporting passages, with each passage carrying part of the needed inference. A partial-evidence case should supply enough context to answer cautiously but not enough to justify high-confidence completion. An abstention case should deliberately omit the decisive fact so that a correct system refuses to overclaim. These distinctions matter because the same answer text can mean different things depending on what evidence was available.

If you already maintain goldens in a general dataset format, specialize them for RAG by adding evidence labels or reference document identifiers. Chapter-level abstractions from the earlier discussion still apply,

but here the important point is that the benchmark must preserve the gap between `retrieval_context`, `context`, and `actual_output`. Without that gap, you cannot tell whether a failure came from candidate generation, packing, or response synthesis.

```
from deepeval.test_case import LLMTestCase

case = LLMTestCase(
    input="What caused the outage?",
    retrieval_context=[
        "DNS records expired at 02:14 UTC.",
        "The cache stampede followed the DNS failure."
    ],
    context=[
        "DNS records expired at 02:14 UTC."
    ],
    actual_output="The outage was caused by expired DNS records."
)
```

That example is intentionally simple. In a real suite, you would attach labels for the failure surface being tested, such as `retrieval_miss`, `ranking_error`, or `context_loss`. You may also attach canonical document IDs if your corpus has stable identifiers. Those IDs make ID-based recall possible and reduce ambiguity when chunk boundaries change across indexing runs.

**Use metric disagreement as a design tool** A benchmark suite becomes more valuable when you know which metric combinations should move together and which should not. Low contextual recall and low faithfulness often indicate missing evidence. High contextual recall and low faithfulness often indicate synthesis drift or unsupported elaboration. Strong answer relevancy with weak faithfulness often indicates a fluent answer that is attractive to humans but not grounded in context. Low precision after an index change often indicates that the retriever began surfacing too much noise even if the final answer still looks acceptable on a quick skim.

You should design cases that make these disagreements visible. If every

case can be solved from one obvious passage, the suite will not expose ranking failures. If every case has a single clean answer and no distractors, it will not expose precision loss. If every case can be answered from general world knowledge, it will not expose retrieval dependence at all. The point of the benchmark is not to maximize average score; it is to create controlled pressure points where the wrong subsystem produces a distinct metric pattern.

This is where long-form and multi-hop tasks deserve special treatment. Short-answer overlap metrics can miss the real failure mode when a generated response is partially correct but incomplete, or when it paraphrases the evidence without preserving the full chain of support. For those cases, claim-level or key-point recall is often a better diagnostic lens than exact-text matching. A long answer can contain the right theme while omitting a necessary clause, and a coarse similarity metric may still reward it. A root-cause suite should expose that omission.

**Stratify by evidence difficulty and distractor structure** Not all retrieval tasks stress the same weakness. A suite that exposes root causes should vary the evidence landscape along a few controlled dimensions.

First, vary the number of supporting passages. Direct lookup cases test whether one passage is found and retained. Multi-hop cases test whether several passages are all present and whether the generator can combine them. If performance falls sharply as the number of required passages rises, the problem may be query decomposition, candidate breadth, or context assembly rather than a single bad chunk.

Second, vary distractor density. Some cases should place highly similar but incorrect passages near the correct one. These cases reveal whether ranking can keep noise below the useful evidence. They also reveal whether the generator is vulnerable to nearby distractors even

when the correct passage is present. This distinction matters because a system with good recall can still fail if precision is poor enough to overload the context window.

Third, vary chunk boundaries. A fact that spans two chunks tests whether the retrieval unit matches the semantic unit of the answer. When chunking is too aggressive, a system may surface half the evidence but fail the benchmark because the other half landed in a different chunk. That is not just a data nuisance; it is a design choice that changes the metric interpretation. The benchmark should make that choice visible.

Fourth, vary answer length and synthesis demand. Some questions require a one-sentence extraction; others require comparing multiple passages, qualifying uncertainty, or synthesizing a summary. Long-context and long-form cases often need richer scoring than simple overlap, because the right answer may be a concise synthesis of several separate claims. If you treat those cases like single-fact lookup, you will underdiagnose failures.

**Build cases that test abstention as well as completion** A robust suite should not reward completion at any cost. If the corpus does not support a confident answer, the correct behavior may be to abstain, qualify, or request clarification. That means your goldens need some cases where the best possible response is not a fully specified answer. These are among the most valuable cases in a production suite because they reveal whether the system knows the boundary of its evidence.

Abstention cases are especially important when your application is high-stakes. A system that confidently answers from thin evidence can score well on superficial relevancy while still violating the grounding contract. A benchmark that includes unsupported questions forces the model to choose between invention and restraint. If the suite never

asks that question, you cannot tell whether the system learned to refuse appropriately or merely learned to sound convincing.

### **Use synthetic data to broaden coverage, not to replace reality**

Synthetic test generation is useful because it gives you breadth, controllability, and edge cases that are hard to find in production logs. It is especially helpful for multi-hop reasoning, near-miss distractors, and rare ambiguity patterns. Knowledge-graph-based generation is particularly effective when you want relation-sensitive cases, because it can produce questions whose answer depends on explicit links between entities, events, or documents.

Synthetic data has a ceiling, though. It can exaggerate regularity and underrepresent the messiness of real user behavior. Real corpora contain messy phrasing, incomplete records, stale entries, contradictory notes, and documents with uneven structure. A benchmark built only from synthetic cases often overestimates how neatly a retriever will behave in production. The practical rule is to use synthetic cases to fill gaps in coverage, then anchor the suite with real examples from production, support tickets, incident reports, or analyst notes. The synthetic cases widen the failure net; the real cases keep it honest.

**Validate the judge before trusting the suite** A benchmark suite is only as trustworthy as the evaluator reading it. If you use judge-based metrics, treat the judge prompt and model as versioned components. That caution is not academic. A weak or drifting judge can make a good system look bad or hide a real regression behind overly generous scores. Papers on automated RAG evaluation repeatedly emphasize the need to compare judge judgments with human assessment and to keep the judge construction process controlled.

That means you should validate a subset of benchmark cases with human review, especially the ambiguous ones. Cases involving partial

support, paraphrased evidence, or long-form synthesis are the most likely to expose judge error. If the judge consistently disagrees with human labels on those cases, you need to adjust the prompt, the support criteria, or the scoring model. Treat the evaluator as part of the benchmark infrastructure, not as an oracle. Once you change the judge, rerun a calibration slice before you trust the new scores.

**Operationalize the suite as a workflow** A root-cause suite is most useful when it drives a repeatable workflow. First, define the failure hypotheses you care about: retrieval misses, ranking noise, chunk-boundary fragmentation, synthesis drift, or abstention failure. Second, curate or generate cases that isolate each hypothesis. Third, annotate the evidence needed for a correct answer and the distractors that should not count. Fourth, run retrieval-oriented and response-oriented metrics together. Fifth, inspect disagreements to decide which subsystem owns the failure.

That flow gives you a concrete diagnostic sequence. If contextual recall is low and faithfulness is low, the retriever likely failed to surface the needed evidence. If recall is high but faithfulness is low, the generator likely ignored or overextended the supplied context. If precision is low but the answer is still good, the system may be compensating for noisy retrieval, which suggests a fragility risk rather than a visible incident. If relevancy is low while the evidence is available, the issue may be query interpretation or task framing rather than retrieval itself.

The important part is that the benchmark must preserve enough structure for that diagnosis to be possible. If you only store the final answer and a binary pass-fail label, you lose the evidence needed for root cause analysis. If you store the retrieval set, the packed context, and the expected evidentiary structure, the same suite can tell you where the engineering effort belongs. That is the difference between a report card and an instrument panel.

**Prefer stable slices over one monolithic score** A single aggregate score is attractive because it is easy to trend, but it hides the most useful information. You get more leverage from a small set of stable slices: direct lookup, multi-hop composition, distractor-heavy retrieval, partial-evidence abstention, and long-form synthesis. Track them separately. When a change improves one slice and harms another, the aggregate may look flat even though the system changed materially.

That slice-based view also helps with release gating. A retriever change that improves direct lookup but worsens multi-hop cases is not a neutral change; it altered the system's operating envelope. A generator update that improves answer relevancy but lowers faithfulness may make demos look better while increasing hallucination risk. A benchmark suite that isolates those shifts gives you a far better basis for deciding whether to ship.

A good RAG benchmark does not just say that the answer was wrong. It points to the layer that needs work, the evidence boundary that failed, and the failure mode that repeated. When your suite can do that consistently, it stops being a scorekeeper and becomes an engineering tool.



## Chapter 5

# Evaluating Multi-Turn and Agentic Systems

*Single-turn scoring can certify an answer while completely missing the failure that matters: the assistant forgot a user constraint three turns later, called the wrong tool, or wandered through an expensive plan before producing a plausible result. This chapter shows how to design an evaluation harness that treats conversations and agents as temporal systems, preserving enough evidence to judge trajectories, intermediate choices, and task outcomes instead of mistaking polished final text for reliable behavior.*

### 5.1. Conversation-Level Quality Beyond Single-Turn Accuracy

A conversational system can look competent one message at a time and still fail the user at the level that matters most: the full exchange.

A support assistant that answers each turn fluently can still forget a shipping constraint after a clarification, contradict an earlier commitment, or recover from a misunderstanding by accident rather than by design. Per-turn grading catches local defects, but it misses the temporal structure that determines whether the dialogue actually serves the user's goal. That gap forces you to evaluate conversation quality as an evolving stateful process, not as a pile of independent responses.

A practical way to ground that distinction is to score each turn for local quality while separately judging the conversation trajectory. A local score tells you whether the response is relevant, safe, and well-formed in isolation. A trajectory score tells you whether the system preserved constraints, repaired errors, and converged toward the user's objective over time. The same conversation can produce strong turn-level outputs and a weak trajectory if the assistant introduces a silent assumption on turn two and only discovers the mismatch on turn five. That is the core failure mode for conversational systems: damage often appears late, after the system has already committed to a path.

```
from deepeval import evaluate
from deepeval.metrics import AnswerRelevancyMetric
from deepeval.test_case import LLMTestCase

turn1 = LLMTestCase(
    input="I need a laptop under $1200.",
    actual_output="Here are some options.",
)

turn2 = LLMTestCase(
    input="It must weigh less than 3 pounds.",
    actual_output="I recommend the X13; it is 2.8 pounds.",
    expected_output="A light laptop under budget.",
)

evaluate(
    test_cases=[turn1, turn2],
    metrics=[AnswerRelevancyMetric()],
)
```

That style of evaluation is useful, but it is incomplete for a dialogue.

You still need to ask whether the assistant remembered the budget after the weight constraint arrived, whether it revised earlier assumptions, and whether it kept the same objective in view when the user corrected an error. In practice, conversation-level evaluation asks you to attach meaning to the sequence itself: what the user committed to, what the system committed to, and which commitments survived later turns.

### **Temporal failure modes that do not show up in isolated turns**

The most important conversational defects are temporal. They accumulate across turns, and each one creates a different kind of user-visible damage.

**Memory decay** appears when the system loses a prior constraint, preference, or reference point. The failure is not simply that the system forgot a fact; it is that it behaved as if the fact never existed. In a travel planner, that means offering nonrefundable options after the user asked for flexibility. In a medical triage assistant, it means ignoring an allergy mentioned three turns earlier. A turn-level grader may score the latest answer as fluent and specific, but the trajectory has already gone wrong.

**State corruption after interruption** happens when a conversation is paused, diverted, or resumed and the assistant fails to reestablish the active state. The user may return after a clarification detour and expect the original objective to remain intact. If the assistant restarts from the wrong assumption, it can continue confidently in the wrong branch. This is especially visible in long support flows, forms, and workflow assistants where progress depends on preserving partially collected information.

**Inconsistency under restatement** appears when the user repeats a requirement and the assistant responds as if the restatement were a new request rather than a correction. Good dialogue systems should treat repetition as evidence, not noise. If the user says, “No, I said

Thursday, not Friday,” the correct behavior is to repair the state, not to defend the earlier answer or re-open a settled question.

**Brittle clarification behavior** shows up when the assistant asks unnecessary follow-up questions or asks them at the wrong time. A system can seem cautious and polite while still imposing avoidable friction. The conversation score should distinguish between justified clarification, where uncertainty genuinely blocks action, and over-clarification, where the system offloads responsibility instead of making progress.

**Recovery failure** is the inability to repair after a misunderstanding. A robust assistant can acknowledge an error, update its internal framing, and move forward without repeating the same mistake. A brittle one loops, re-asks the same question, or returns to the wrong assumption after each correction. The difference is not cosmetic; it determines whether the user can complete the task efficiently.

**Objective drift** emerges when the assistant slowly shifts away from the user’s original goal. This often happens when the system over-optimizes for the latest turn. The user begins with one task, adds a constraint, then asks for an example, and the assistant quietly starts optimizing for the example rather than the original task. Drift is easy to miss if you inspect only the final answer. It becomes obvious only when you trace the conversation as a sequence of commitments.

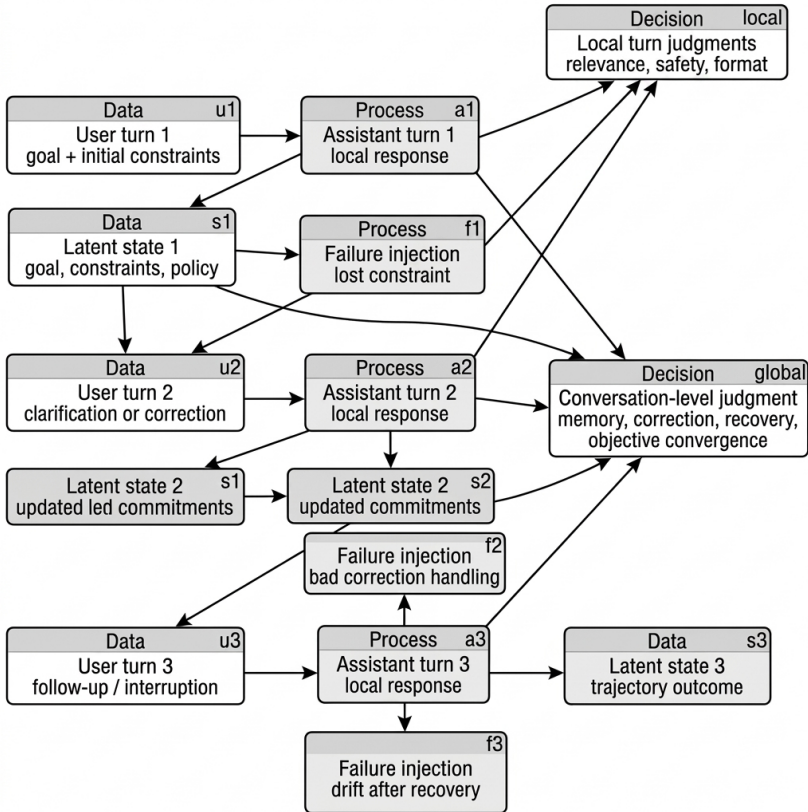
These defects are not rare edge cases. They are the normal failure surfaces of stateful dialogue. The reason is structural: a conversation is a temporal object. Each turn modifies the latent state of the interaction, and the quality of the next turn depends on whether that state was updated correctly.

### **Why local turn quality and global trajectory quality differ**

Local turn quality measures what is visible now. Global trajectory qual-

ity measures whether the whole exchange stayed coherent. Those are related, but they are not substitutes. A high-quality reply can still be a bad conversational move if it encodes the wrong assumption for later turns. A terse clarification can be bad locally but excellent globally if it prevents the assistant from committing to an unsafe interpretation.

This distinction matters because users do not experience messages independently. They experience the continuity of the interaction: whether the assistant remembers, corrects itself, keeps context, and converges on the right outcome. A conversation-level evaluator therefore has to track more than response text. It needs evidence of latent state transitions: which constraints were introduced, which were revised, which were preserved, and which were silently dropped.



That visual separation is useful because it prevents a common evaluation mistake: collapsing the dialogue into the last answer. If the final response looks polished, it is tempting to assign a passing grade. But a conversation can end with a good answer after spending three turns in the wrong state. In operational settings, those detours matter. They increase user effort, raise the chance of compounding mistakes, and make the system harder to trust.

### What you must preserve to evaluate a conversation

Conversation-level evaluation only works if the harness preserves

enough evidence to reconstruct the dialogue as a stateful process. At minimum, you need the ordered user messages, the assistant messages, and the annotations that identify which utterances changed the working state. That state can include named slots, task goals, preferences, refusal boundaries, or any other artifact that the application uses to maintain continuity.

You do not need to expose hidden reasoning to do this well. You need externally observable evidence: the turns, the revised constraints, the explicit corrections, and the end condition. If the system supports structured state, preserve that state or a faithful trace of it. If it does not, preserve the messages and the evaluation labels that indicate what the assistant should have inferred from them. This is where a conversation-level evaluator differs from a simple text scorer. The scorer does not just inspect the latest string; it reasons over how the string fits into the evolving interaction.

Framework support for conversational abstractions is version-sensitive in practice. Some releases expose first-class conversation or chat test cases, while others require you to model a dialogue as a list of message objects or as a trace with turns. You should treat that support as an implementation detail, not as a change in the underlying evaluation objective. The objective is stable: judge whether the sequence of messages maintains coherent task state.

### **Conversation types that require trajectory-level judgment**

Different dialogue types surface different risks, and that should shape the evaluation contract.

Information-seeking conversations emphasize context retention and answer consistency. The assistant may need to keep track of multiple facts introduced over time, especially when the user refines a request. The key trajectory question is whether the final answer remains anchored to earlier constraints.

Constrained task completion flows emphasize commitment management. Scheduling, booking, and configuration tasks often require the system to collect partial information across turns. A single missed constraint can invalidate the entire session. Here, the evaluator should inspect whether every required constraint survived to the terminal state.

Correction-heavy support flows emphasize repair. The assistant must detect when it is wrong, update its state, and avoid reintroducing the same error. The important metric is not simply whether the final response is correct, but whether the system recovers promptly and does not cycle through the same mistake.

Ambiguity-resolution conversations emphasize clarification quality. The assistant should ask questions only when uncertainty blocks safe progress, and it should ask the right question at the right time. A conversation score should reward questions that reduce uncertainty and penalize questions that merely delay resolution.

These patterns have one thing in common: correctness depends on more than output content. It depends on how the system uses the sequence of turns to stabilize meaning.

### **How to encode conversation-level expectations**

A conversation-level test case should describe the interaction as an evolving scenario with explicit end conditions. The end condition is not always a textual answer. It may be a correct state transition, a valid plan, a completed form, or a repaired understanding after correction. The expectation should specify what must remain true throughout the dialogue, not only what must be true at the end.

That means you should encode at least three kinds of conversation-level expectations.

First, define **persistent constraints**. These are facts or preferences that must survive until the conversation ends. If the user says the prod-

uct must be vegan, the evaluator should check that the assistant never returns to a non-vegan assumption later.

Second, define **repair conditions**. These state how the system should react when the user corrects it. The expected behavior may be to acknowledge the correction, update the active state, and proceed without reopening the old mistake.

Third, define **convergence conditions**. These capture whether the exchange is moving toward completion. A good trajectory reduces uncertainty, narrows the solution space, and eventually reaches a state that satisfies the user's intent.

This is not the same as writing a long prompt with vague expectations. The expectations should be machine-checkable or human-checkable against concrete evidence. If the system is supposed to preserve a constraint, name that constraint. If it is supposed to recover from a correction, define what recovered state looks like. If it is supposed to converge, identify the observable end condition.

### **Common anti-patterns in conversational evaluation**

Several mistakes recur whenever teams retrofit single-turn testing onto multi-turn systems.

The first is **final-answer fixation**. This reduces the dialogue to the last assistant message and ignores all earlier errors. It is attractive because it is easy to implement, but it hides the failures users feel most strongly: confusion, repetition, and wasted effort.

The second is **politeness over-weighting**. A fluent and courteous assistant can still be wrong. Politeness is a surface property; conversation quality depends on whether the system respects constraints, handles corrections, and advances the task.

The third is **vague scenario design**. If the expected end state is

underspecified, no metric can distinguish good behavior from lucky behavior. The evaluation then becomes an argument about interpretation rather than evidence. A useful scenario names the critical constraints and the repair conditions that matter.

The fourth is **over-penalizing valid alternate paths**. Multi-turn systems often have more than one acceptable route to completion. If the evaluator insists on a single script, it will reject robust dialogue strategies that ask a different clarifying question or sequence the steps differently. Conversation-level evaluation should distinguish between essential constraints and incidental phrasing.

The fifth is **treating user satisfaction as identical to task success**. A user may feel satisfied by a pleasant answer that never actually solves the problem, or dissatisfied by a terse but correct correction. Satisfaction can be a useful signal, but it is not the same thing as trajectory quality.

### **Why this framing matters operationally**

Once you adopt a trajectory view, the release question changes. You are no longer asking whether the model can produce a good response in isolation. You are asking whether it can hold state, absorb corrections, and stay aligned with the user's goal under realistic dialogue pressure. That is the standard that matters for assistants that negotiate requirements, guide workflows, or maintain long-lived context.

The practical consequence is that conversation evaluation becomes diagnostic rather than decorative. A good trajectory score tells you the assistant can sustain a coherent interaction. A bad one tells you where the interaction broke: memory loss, correction failure, clarification misuse, or drift. Those signals are far more actionable than a single pass/fail label on the last message, because they tell you whether the defect lies in context retention, recovery logic, or goal tracking.

## 5.2. Designing Multi-Turn Metrics and Scenario Suites

A multi-turn benchmark is useful only when it makes failure expensive to hide. A system that appears strong in isolated turns can still fail in a six-message interaction because it loses a constraint, asks for information it already has, or drifts into an answer that is locally plausible but globally wrong. The benchmark design task is therefore not to inflate single-turn test cases into longer transcripts. It is to construct dialogue trajectories that force the system to prove continuity, correction, and convergence under realistic conversational pressure.

The practical mistake teams make is to average per-turn scores and call the result a conversation score. That produces a number, but not a measurement model. Averages obscure whether the assistant preserved a crucial constraint, whether it recovered after being corrected, and whether the final success required unnecessary detours. A conversation benchmark should instead represent a sequence of dependent states, where each new turn changes the evaluation surface for the next one. That makes trajectory quality observable rather than inferred.

### **Start from the dialogue spine, not from the metric.**

The most reliable way to design a scenario suite is to define the conversation spine before choosing any score. The spine is the minimal sequence of events that the dialogue must pass through to expose the quality property you care about: initial goal, one or more constraint-bearing turns, one friction event, one repair opportunity, and a terminal condition. Once you have that skeleton, you can decide which dimensions deserve strict scoring and which deserve softer judgment.

For example, a scheduling assistant scenario might begin with a meeting request, add a time-zone constraint, introduce a last-minute change, and end with a confirmation. A support scenario might begin

with a complaint, add account details, include an incorrect suggestion from the assistant, and then test whether the system recovers cleanly after correction. These are not simply longer prompts. They are structured sequences in which later turns depend on the integrity of earlier state.

```
scenario = {
  "goal": "Book a refund-eligible flight",
  "turns": [
    {"user": "Find a flight to Boston."},
    {"user": "It must be refundable."},
    {"user": "Actually, I need it next Friday."},
    {"user": "Don't exceed $600."}
  ],
  "expected_state": {
    "destination": "Boston",
    "refund_policy": "refundable",
    "date": "next Friday",
    "budget": 600
  },
  "success": "Assistant preserves all constraints and proposes a
    valid option."
}
```

That kind of structure is valuable because it exposes trajectory dependencies directly. If the assistant forgets the refundable constraint after the date changes, a final-answer-only metric may still score the last suggestion as relevant. A scenario spine makes that hidden error visible. It also gives you a place to define what the assistant is allowed to ask, when it is allowed to defer, and what it must never lose.

### Map conversational risk to observable dimensions.

Once the spine is defined, you need metric dimensions that reflect the actual risk in the dialogue. Good multi-turn metrics are not generic quality buckets. They are tied to the way the interaction can go wrong.

**Constraint retention** measures whether earlier requirements survive later turns. This is the backbone metric for planning, booking, configuration, and any workflow where the user introduces hard con-

straints gradually. Retention is not just memory in the abstract; it is evidence that the system maintained a coherent working state.

**Correction responsiveness** measures whether the assistant updates its state when the user changes or rejects a prior assumption. A system that politely acknowledges a correction but continues to answer from the old state fails this metric. The evaluator should inspect both the acknowledgment and the downstream effect.

**Clarification efficiency** measures whether the assistant asks questions that reduce uncertainty without unnecessary delay. The best clarification is the one that rules out the right set of possibilities with the fewest turns. That metric is inherently contextual: in high-risk domains, cautious clarification is often correct; in low-risk flows, it may be friction.

**Recovery quality** measures how cleanly the assistant returns to the task after an interruption, an error, or an explicit correction. This is different from raw task completion. Recovery asks whether the assistant re-establishes the right frame before continuing.

**Goal convergence** measures whether the dialogue moves toward a stable end state. A conversation can be fluent and still non-convergent if it loops, broadens scope, or keeps revisiting settled questions. Convergence captures whether the interaction is ending because the goal is solved, not because the assistant ran out of steam.

**Trajectory efficiency** measures the cost of reaching the final state. A system can eventually succeed while wasting turns, repeating questions, or forcing the user to restate requirements. Efficiency matters because extra turns are not free: they add latency, increase abandonment risk, and amplify the chance of new errors.

These dimensions should not all be scored the same way. Constraint retention may deserve a hard penalty because missing a hard require-

ment invalidates the task. Clarification efficiency may deserve a softer, risk-aware score because a cautious assistant can legitimately trade one extra question for lower error probability. That distinction is the point of a scenario suite: it lets you encode different thresholds for different kinds of conversational risk.

### **Use scenario families to expose distinct failure modes.**

A good suite mixes canonical success paths with adversarial and recovery-heavy patterns. If you only test ideal dialogues, you will mostly measure competence under cooperative conditions. Real users do not behave that way.

**Happy-path scenarios** verify ordinary task completion. They provide the baseline and keep the suite grounded in the intended use case. These cases should be cleanly solvable with the information provided and should establish the minimum acceptable interaction pattern.

**Correction-heavy scenarios** force the assistant to revise itself. The user changes a detail, rejects an assumption, or clarifies an earlier statement. These cases expose whether the system can update state without reintroducing the old error. They are especially useful for assistants that summarize, configure, or plan.

**Interrupted scenarios** test whether the conversation can resume after a detour. The user may switch topics and then come back, or the assistant may need to pause for missing information. These scenarios reveal whether the system can preserve latent context across gaps.

**Ambiguous scenarios** test the quality of uncertainty handling. The user request lacks a necessary detail, contains a conflict, or admits multiple interpretations. The key question is not whether the assistant can answer immediately, but whether it asks for the right missing information.

**Frustrated-user scenarios** test resilience under emotional pres-

sure. The user may be impatient, repetitive, or terse. These cases matter because a system that performs well only with cooperative phrasing is not robust enough for production dialogue.

**Abrupt-return scenarios** test whether the assistant can recover a dropped thread. The user returns to a topic after a long detour, and the assistant must re-anchor to the original objective. This is one of the clearest ways to detect objective drift.

A balanced suite uses all of these families because each one attacks a different layer of conversational quality. Together they distinguish a model that merely responds well from one that sustains a usable interaction.

### **Score progression, not just final output.**

The central design choice in a multi-turn metric is whether you score the ending state alone or the path to that state. In dialogue systems, the path matters. A final answer may be correct for the wrong reasons, or it may be correct only because the user repaired the model repeatedly. A trajectory-aware metric captures that distinction.

A practical scoring scheme usually combines three signals. First, a **terminal success** check verifies that the final state satisfies the user objective. Second, a **state continuity** check verifies that the assistant retained the required constraints throughout the exchange. Third, a **repair quality** check evaluates how the assistant handled errors, corrections, or interruptions along the way.

A simple weighted sum can work if the scenario is stable and the task is well-defined, but you should resist collapsing everything into a single opaque score too early. One reason is interpretability: if the score drops, you need to know whether the problem was memory, clarification, or recovery. Another reason is governance: a support flow with a safety constraint should fail for different reasons than a casual recom-

mendation flow.

Binary trajectory correctness is often too coarse. If you only record pass or fail, you lose the distinction between a near miss and a total breakdown. Counting extraneous turns, missed constraints, or incorrect recovery steps gives you a sharper signal. That idea matters operationally because two failed conversations may require very different fixes. One may need better state tracking; the other may need a revised clarification policy.

**Choose thresholds from user risk, not from metric convenience.**

Metric thresholds should reflect the cost of being wrong. A booking assistant that loses a date constraint can create a bad reservation. A casual brainstorming assistant can tolerate more ambiguity. The benchmark should encode that difference.

In high-risk domains, you want hard constraints and low tolerance for recovery failure. If the assistant must retain allergy information, policy restrictions, or financial limits, then a single omission may be a blocking failure. In lower-risk domains, a longer but more careful clarification path may be acceptable. The threshold is therefore not just a statistical cutoff; it is an expression of domain risk.

That also means you should avoid a universal “conversation score” threshold across all suites. A support chatbot, a scheduling assistant, and a creative writing helper have different acceptable trade-offs. Even within one application, different scenario families may deserve different thresholds. A correction-heavy case may require near-perfect repair behavior, while a low-stakes information-seeking case may tolerate a mild detour.

**Build suites from real traces and curated edge cases.**

A durable scenario suite should not be made only from synthetic happy-

path examples. Production traces reveal where users actually confuse the system, where corrections happen, and where conversations stall. Those traces are useful because they reflect real friction, not imagined friction. Curated edge cases then fill in the gaps: contradictory statements, partial revisions, topic jumps, abandoned threads, and repeated clarifications that are rare in logs but expensive in practice.

That mix matters for coverage. Production traces tell you what happens often. Curated adversarial cases tell you what happens dangerously. If you rely only on one of those sources, you will bias the suite. Real traces without adversarial design tend to under-sample hard failures. Hand-authored cases without production grounding tend to overfit to artificial patterns.

When you convert traces into scenarios, normalize them into a consistent conversation template. Preserve the turn order, the critical constraints, and the observable end condition. If the original exchange contains irrelevant chatter, drop it unless it affects the state. The goal is not to reproduce transcript length. The goal is to reproduce the conversational dependency that caused the outcome.

**Treat disagreement between the metric and the human reviewer as data.**

A useful suite does not merely produce a score. It produces disagreements that teach you where the scoring model is brittle. If the metric rewards a conversation that a reviewer finds unsafe, your metric is under-specifying a risk. If the metric rejects a dialogue that a reviewer considers acceptable, your scenario may be over-constrained.

That is why you should run a small calibration loop before you freeze the suite. Sample a set of conversations, score them automatically, and then inspect the mismatches manually. Look for patterns. Are false positives concentrated in polite but wrong answers? Are false negatives concentrated in valid alternate paths that the metric did not anticipate?

Are correction cases being penalized because the evaluator expects a single scripted wording rather than a semantically equivalent repair?

This calibration step is not optional. It is how you keep the benchmark from encoding accidental preferences. For multi-turn systems, accidental preferences are especially dangerous because they can reward the wrong conversational style while missing trajectory failure.

### **Operationalize the suite as a repeatable workflow.**

A multi-turn benchmark becomes useful when you can run it consistently and evolve it without breaking comparability. The operational workflow should be simple enough to repeat and strict enough to preserve meaning.

Start by defining a conversation risk. Then author a scenario template that makes that risk observable. Instantiate several turn sequences from the template, including both clean and adversarial variants. Declare what must remain true across the dialogue: constraints, repaired state, or convergence. Choose metrics that reflect those requirements. Run the suite. Inspect disagreements between automatic scoring and human judgment. Refine the scenario or the metric when the disagreement reveals a blind spot.

```
from deepeval.test_case import LLMTestCase
from deepeval.metrics import GEval

case = LLMTestCase(
    input=[
        "I need a budget phone.",
        "It must have long battery life.",
        "Actually, make that a phone with a headphone jack."
    ],
    actual_output="Here is a model that fits all three needs.",
    expected_output="A phone that preserves all constraints."
)

metric = GEval(
    name="constraint_retention",
    criteria="Checks whether prior constraints persist across turns."
)
```

That style of evaluation keeps the scenario readable and the intent explicit. The important point is not the specific API shape. It is that the test case expresses a trajectory and not just a single prompt-response pair. If your framework version represents conversations as separate messages rather than a grouped input list, the same design principle still applies: the evaluator must see the sequence and the persistence conditions.

**Avoid benchmark designs that cannot produce stable judgments.**

Several anti-patterns recur in multi-turn suites.

**Averaging single-turn scores** creates a number that looks precise but hides trajectory failures. If one turn is excellent and another is catastrophic, the average can be misleadingly moderate. That obscures the very defect you are trying to detect.

**Overfitting to one assistant persona** makes the suite fragile. If every scenario assumes the model will ask questions in a particular tone or format, you will penalize valid behavior that uses different wording but satisfies the same conversational requirement.

**Using unobservable end states** makes scoring impossible. If the terminal condition depends on internal intent or on facts the evaluator cannot inspect, the suite cannot be judged reliably. The end condition should be inferable from the transcript, the state, or the task artifact.

**Conflating satisfaction with success** mixes UX sentiment with task completion. User satisfaction is informative, but it does not replace a well-defined outcome. A concise correction can feel abrupt and still be the right move. A pleasant but evasive answer can feel good and still fail the task.

**Ignoring valid alternate paths** punishes robust systems. In multi-turn dialogues, multiple trajectories may solve the same problem. Your metric should identify the required invariants and leave room for semantically equivalent progress.

A mature scenario suite makes these pitfalls harder to hide. It forces the assistant to demonstrate continuity, repair, and convergence in ways that a single-turn benchmark cannot. That is what gives conversation evaluation its diagnostic value: you can see not only whether the assistant arrived at the right place, but whether it got there by preserving the conversation's structure rather than by stumbling into success.

### 5.3. Tool Use, Plans, and Intermediate Execution State

An agent becomes hard to evaluate the moment its quality depends on *how* it reached an answer rather than only on the answer itself. A customer service agent can return the right refund policy after querying the wrong ticket, or a workflow assistant can complete a task after unnecessary retries and hidden branch changes. A final-response grader sees success. An operations team sees cost, latency, safety risk, or brittle behavior. Process-aware evaluation exists to close that gap.

The simplest way to make the gap concrete is to capture the execution trace as a first-class artifact and score the trace, not just the terminal text. A trace gives you the user input, the exposed plan if the system produces one, the tool calls, the tool outputs, retries, exceptions, approvals, and the state transitions that connect them. That evidence lets you ask whether the agent selected the right tool, passed the right arguments, recovered correctly from failure, and preserved state across steps. Without that evidence, the evaluator can only guess.

```
from openai import OpenAI
```

```
client = OpenAI()

resp = client.responses.create(
    model="gpt-4.1",
    input="Find the latest docs for the billing API.",
    tools=[{"type": "web_search"}],
)
print(resp.output_text)
```

That kind of invocation looks deceptively simple, but the evaluation problem starts after the call returns. You need to know whether the agent decided to search at the right time, whether it used the right query, whether it consumed the result correctly, and whether any hidden retry or fallback path changed the meaning of the run. The model output alone does not answer those questions. The surrounding process does.

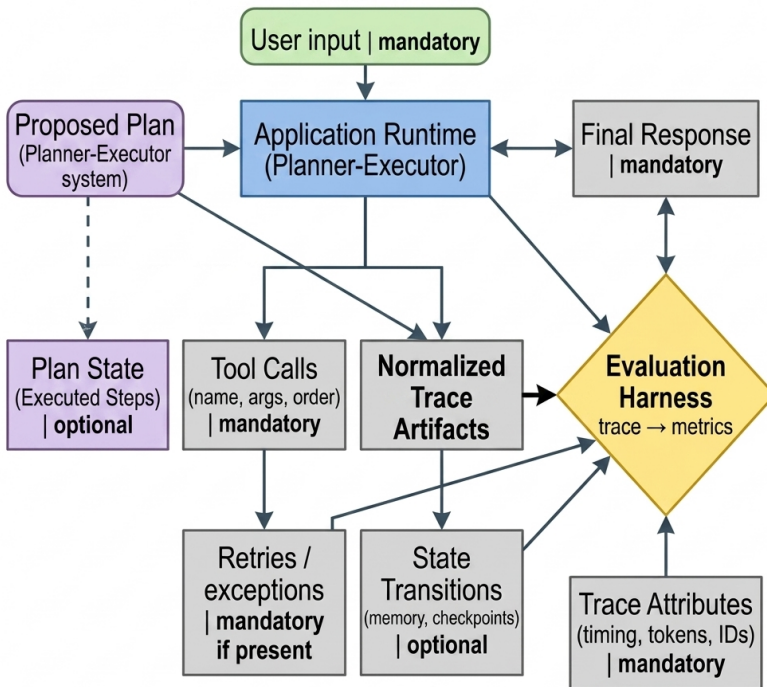
**Treat traces as evidence, not as debug logs.** A useful trace is not a verbose printout of everything the runtime happened to emit. It is a normalized record of the decision path. You want to capture enough structure that a later evaluator can reconstruct what happened without re-running the agent. That usually means preserving the root user request, each subtask or planning step, every tool invocation with arguments, the corresponding tool result, any exception or retry, and the final response.

A good evidence model distinguishes three layers.

- *Application runtime* is the live agent execution: the model, its planner, its tool router, and the external systems it talks to. This layer produces behavior, but it is not yet evaluation-ready.
- *Captured artifacts* are the normalized records of the runtime: messages, spans, tool events, attributes, state snapshots, and exceptions. This layer turns behavior into analyzable data.

- *Evaluation inputs* are the subset of artifacts that a metric actually consumes: expected tool families, allowed argument forms, intermediate state assertions, latency windows, or branch-specific recovery expectations.

That separation matters because observability and evaluation solve different problems. Observability can tolerate noisy logs and incomplete semantic labeling. Evaluation cannot. If you dump raw traces into a scorer without defining the contract, you get a brittle harness that overfits to log format rather than agent behavior.



The diagram highlights a practical point: not every artifact has to be present for every system, but the harness must know which artifacts are mandatory for the class of agent being evaluated. A simple retrieval agent may not expose a plan, while a planner-executor system often should. A human-in-the-loop workflow may require approval checkpoints. A durable workflow may require state snapshots at interruption boundaries. The evaluator should not infer those fields from the final response; it should expect them explicitly.

**Separate observable process from latent reasoning.** You should not build evaluation around unverifiable chain-of-thought access. That is both operationally fragile and conceptually the wrong target. What you can evaluate reliably is the process that the agent manifests externally: the plan representation if the system emits one, the sequence of tool calls, the arguments chosen, the returned tool outputs, the branch points taken, and the way the state changes after each event.

That distinction keeps the harness grounded in evidence. If the agent claims to have checked a customer record, you should see the call that did it. If it claims to have compared two candidate actions, you should see the order in which those actions were considered or executed. If it restarts after failure, you should see which steps repeated and whether the repeated steps were safe to repeat.

OpenTelemetry provides a vendor-neutral way to represent that evidence. A trace contains a root span for the overall interaction and child spans for sub-operations. Attributes can hold tool names, argument hashes, query IDs, model IDs, timing, or cost metadata. Events can record exceptions, approvals, or checkpoints. That structure is valuable because it is stable enough for analysis across systems, yet flexible enough to carry the specific fields your agent produces.

**Know which artifacts matter for which agent pattern.** The evidence requirements differ by agent type, and the harness should reflect that.

- *Retrieval agents* need query text, retrieval parameters, selected documents, and the context passed to generation. If the agent does not retrieve the right material, the final answer may still look plausible while being grounded in the wrong evidence.
- *API-orchestrating assistants* need the tool name, argument payload, response payload, and any validation or retry logic. Here, correctness depends on argument shape as much as on tool selection. A call to the right API with the wrong date, ID, or filter is still a failure.
- *Workflow automators* need step order, checkpoints, state transitions, and retry boundaries. These systems can appear successful even when they take a needlessly expensive path or duplicate work after an interrupt.
- *Planner-executor systems* need the plan representation, the plan-to-action mapping, and the difference between intended and executed steps. If you only inspect the final answer, you miss whether the agent executed the plan it actually proposed.

Tool-use evaluation becomes much sharper when you decompose it into the same dimensions that govern competence. Toolformer-style decomposition is helpful here: you can ask when the agent decided to call a tool, which tool it selected, what arguments it used, and whether it integrated the result correctly. Those are separate questions, and they often fail independently. A system may know *that* it needs a tool but fail on the argument format. Or it may call the right tool but ignore the returned result and continue reasoning from stale assumptions.

**Make state explicit when execution can pause or resume.**

State persistence becomes essential as soon as the agent can pause, resume, or branch. A durable-execution model preserves progress at checkpoints so that a run can continue after failure or interruption. That is not just an infrastructure convenience; it affects what you can evaluate.

A resumed run may re-execute part of a node from the beginning rather than continuing from the exact instruction where it stopped. The implication is straightforward: the evaluator must tolerate duplicated pre-interrupt execution and must require idempotent side effects for steps that can run twice. If a node sends an email before an interrupt and then runs again on resume, the harness should treat duplicate sending as a failure in the workflow, even if the final answer appears correct.

That is why intermediate state matters. You need to know what the agent had committed to before the pause, what state it had persisted, and what it reconstructed on resume. If the state is not explicit, the evaluator cannot distinguish a clean recovery from an accidental re-derivation that happens to end well.

**Evaluate intermediate steps by traversing the run tree.**

Intermediate-step evaluation is more informative than terminal inspection because it lets you grade the path, not just the destination. A run tree provides the structure needed for that: root interaction, child substeps, nested tool calls, retries, and state updates. You can then attach checks to any node in the tree.

That makes several useful metrics possible.

- *Tool selection correctness* asks whether the agent called the right tool family for the task. A date lookup should not become a web search if the local calendar already contains the answer.

- *Argument correctness* asks whether the selected tool received valid and sufficient parameters. This is often where agents fail quietly. The API call succeeds, but it succeeds on the wrong entity or with incomplete filters.
- *Output use correctness* asks whether the agent incorporated the returned result into the next step. A model may retrieve the right record and then ignore it, falling back to a prior assumption.
- *Branch soundness* asks whether the agent followed a justified path after uncertainty or failure. A branch that recovers from an error can still be poor if it takes too many detours or discards valid state.
- *Replay consistency* asks whether the same trace, when resumed or replayed under the same conditions, produces the same high-level behavior. This is essential when you rely on durable execution and checkpointing.

These checks are stronger than final-answer scoring because they localize failure. If the answer is wrong, you can tell whether the fault lies in retrieval, planning, argument formation, or state management. That is the difference between a useful evaluation harness and a binary pass/fail script.

**Normalize observability data before you score it.** A common mistake is to reuse raw logs directly as metrics input. Raw logs are helpful for debugging, but they are not yet evaluation artifacts. They often contain unstable field names, duplicate events, backend-specific trace formats, and timing artifacts caused by retries or queueing.

You should normalize them into a schema that preserves the semantics you care about. At minimum, each event should identify the parent span or run node, the operation type, the tool or model involved, the

arguments or state deltas, the timestamps, and the success or failure outcome. If the agent uses message-based state, preserve the message order and any redaction applied before scoring.

This is also where version sensitivity matters. Frameworks that support conversation or agent traces may change field names, event nesting, or the default shape of captured state across releases. Your evaluation contract should not depend on incidental representation details. It should depend on stable semantics: what was called, with what inputs, in what order, and with what result.

**Use the right evidence granularity for the class of judgment.**

Not every evaluation needs the same resolution. If you are testing whether the agent invoked a required safety check, a coarse span-level event may be enough. If you are testing whether it passed the right date to a booking API, you need argument-level inspection. If you are testing recovery after interruption, you need checkpoint-level state and repeated-step visibility.

That granularity choice is an engineering trade-off. Finer-grained traces give you better diagnostics, but they increase storage, processing cost, and operational complexity. Coarser traces are cheaper, but they obscure near-misses and make it harder to explain failures. The right choice depends on the release risk. A customer-facing workflow with external side effects justifies much richer evidence capture than a low-risk internal assistant.

A useful rule is to capture the finest evidence you need for the riskiest failure mode you are willing to tolerate. If the consequence of a wrong action is high, preserve the artifacts needed to reconstruct that action unambiguously.

**Anchor the evidence model in concrete failure patterns.**

Several failure patterns recur often enough that the harness should expect them explicitly.

- *Missing tool arguments* produce successful-looking calls that fail semantically. The API returns something, but not the right something.
- *Non-deterministic trace serialization* makes repeated runs hard to compare. If field order, identifiers, or event nesting change between runs, the evaluator cannot tell whether behavior changed or only representation changed.
- *Hidden retries* distort cost and latency. If the system internally retries a tool call and only the successful branch survives in the log, the trace understates the true operational burden.
- *Conflating proposed and executed plans* hides divergence. The agent may say it will do one thing and then do another. The evaluator must inspect execution, not intent alone.
- *Preserving only successful terminal branches* creates survivorship bias. You learn what worked, but not what failed before the agent recovered.

These are not peripheral logging bugs. They are evaluation bugs, because they change what you can infer about the agent.

**Use traces to support better judgments, not just stricter ones.**

The value of process evidence is not only that it lets you reject bad runs. It also lets you distinguish among good runs that are operationally different. One agent may complete a task in three clean steps; another may complete the same task after a detour that is acceptable in a low-risk workflow but too expensive in a latency-sensitive one. A

trace-aware harness can represent that difference without collapsing both runs into the same final score.

That is why evidence modeling comes before metrics. Once you have a reliable trace, you can build judgments over it: whether the tool path was valid, whether the state survived interruptions, whether retries were safe, and whether the plan-to-execution gap stayed within bounds. Without the trace, those judgments are guesses.

A trustworthy agent harness therefore treats the execution record as part of the test case itself. The user request defines the goal; the trace defines the path; the evaluator checks whether the path was acceptable for the risk level and operational constraints of the task. That is the level at which agent evaluation becomes useful for release decisions rather than merely descriptive for postmortems.

### **5.4. Task Completion, Tool Correctness, and Step Efficiency**

Once you can observe an agent's trace, the evaluation problem shifts from evidence collection to judgment. The useful question is no longer whether the run ended with a plausible answer, but whether the agent completed the task through an acceptable path. That path can be correct, inefficient, brittle, or unsafe in ways a terminal answer will never reveal. A release decision depends on those differences. A customer-facing workflow can tolerate a slightly longer path if the tool selection is robust and the state is preserved. A latency-sensitive or side-effecting workflow cannot.

A practical way to structure that judgment is to separate four dimensions: task completion, tool correctness, plan quality, and step efficiency. These dimensions are related, but they fail independently. An agent may complete the task while using the wrong tool family, or it

may use the right tool but pass malformed arguments, or it may follow a reasonable plan while taking too many unnecessary steps. The harness should expose those distinctions rather than flatten them into one opaque score.

```
from deepeval.test_case import LLMTestCase
from deepeval.metrics import GEval

case = LLMTestCase(
    input="Update the order to ship to Boston.",
    actual_output="I updated the shipping address.",
    expected_output="Order updated with Boston address."
)

metric = GEval(
    name="task_completion",
    criteria="Checks whether the task was completed.",
)
```

That example is intentionally minimal. In a real agent trace, you would score not only the final response but the route taken to reach it: whether the order update tool was called, whether the address argument was valid, whether the system confirmed the right record, and whether it retried or branched in a way that changed the operational cost of the run. Final-answer inspection can tell you that the user saw a satisfactory sentence. It cannot tell you whether the system updated the wrong order first and recovered later.

### **Task completion is necessary, but not sufficient.**

Task completion is the most intuitive dimension because it corresponds to the user's goal. Did the agent achieve the outcome the conversation or workflow demanded? In a booking task, that means the reservation was created with the right constraints. In a retrieval task, that means the agent returned the required evidence. In a support workflow, that means the issue was resolved or escalated correctly.

Completion alone, however, is too weak a standard for agentic systems. An agent can complete the task after making unsafe assumptions, vio-

lating a preferred process, or consuming far more resources than the workflow can tolerate. If you evaluate only terminal success, you reward accidental competence and hide brittle behavior. That matters most when the agent has side effects. A run that ends correctly after the wrong intermediate action is not harmless just because it recovered.

This is where the benchmark design from earlier sections becomes operational. Your scenario should define not only what counts as completion, but what completion is allowed to cost in terms of steps, tool families, and state transitions. If the task is low-risk, the harness can allow a broader set of acceptable paths. If the task is safety-sensitive or externally visible, completion must satisfy stricter process conditions.

### **Decompose tool correctness into separate failure modes.**

Tool correctness is not a single property. It has at least four separable parts.

- *Tool selection correctness* asks whether the agent chose the right tool family for the job. A calendar lookup should not become a web search if a local scheduling API already owns the source of truth. A record update should not become a free-form natural-language response. Tool selection errors often look minor in logs and major in production because they force the agent onto the wrong operational surface.
- *Argument correctness* asks whether the selected tool received a valid and sufficient payload. This is where many agents fail quietly. The tool call may succeed syntactically while targeting the wrong record, leaving out a required filter, or using a stale identifier. Argument correctness usually matters more than the tool name because the same tool can be used correctly or incorrectly depending on the payload.
- *Output-use correctness* asks whether the agent incorporated the

result of the tool call into the next step. A model can call the right tool and then ignore the returned value, reverting to a prior assumption or hallucinated summary. That failure is especially dangerous in multi-step workflows because the tool output often exists to constrain later reasoning.

- *Policy compliance* asks whether the agent respected forbidden-tool rules, access constraints, or workflow boundaries. An agent that reaches the right answer by using a disallowed source or an unapproved action still fails the policy dimension even if the final result looks correct. This is often the decisive dimension in regulated or internally controlled environments.

Tool correctness becomes much easier to score when you treat the trace as a sequence of decision points. Toolformer-style decomposition is useful here because it gives you a clean scoring rubric: when to call, which tool to call, what arguments to pass, and how to use the result. Those are distinct judgments, and each one can fail independently. A system may know it needs a tool but choose the wrong one. It may choose the right tool but pass an invalid identifier. It may receive the right output but fail to propagate it into the next state.

### **Plan quality measures whether the path is coherent.**

If the agent exposes a plan, you can evaluate the soundness of that plan against the executed trace. Plan soundness asks whether the decomposition is sensible for the task, whether the order of steps matches the dependencies, and whether the agent avoids gratuitous detours that do not add information.

A good plan for a multi-step workflow should be causally ordered. Gather the missing constraint before making a decision. Validate the record before writing to it. Retrieve evidence before summarizing it. Plans that invert these dependencies often create self-inflicted recovery

work. The final answer can still be right, but the path becomes brittle and expensive.

You should also compare the intended plan with the executed plan. In practice, these diverge. The agent may say it will verify a condition and then skip it. Or it may pivot into an unrelated branch after encountering ambiguity. That divergence matters because the plan reveals what the system believed it was doing, while the trace reveals what it actually did. A robust evaluator checks both.

Plan quality is not the same as strict process conformance. Some tasks admit multiple valid plans. A rigid evaluator that insists on a single canonical path will incorrectly reject good alternative strategies. That is why the judgment model should distinguish essential invariants from incidental sequencing. If the tool family, critical constraints, and safety checks are satisfied, alternative decompositions may be acceptable.

**Measure step efficiency as an operational cost, not a cosmetic preference.**

Efficiency is often the first dimension teams underweight and the last one they wish they had measured. An agent can complete the task, call the right tools, and still be operationally unacceptable because it takes too many turns, repeats unnecessary actions, or generates avoidable latency. In customer-facing systems, those costs directly affect abandonment. In backend workflows, they translate into compute cost, rate-limit pressure, and brittle failure surfaces.

A useful efficiency metric considers several signals.

- *Step count* captures how many major actions the agent took. Fewer steps are not always better, but excess steps usually indicate either poor planning or unnecessary caution.
- *Redundant calls* count repeated invocations that add no new in-

formation. A repeated lookup can be harmless if it confirms a volatile source, but it is suspicious if the underlying data is stable.

- *Retry burden* measures how often the agent had to correct itself after malformed arguments or failed calls. Some retries are expected; repeated retries often point to weak argument formation or poor error interpretation.
- *Latency and token cost* turn trace length into resource cost. These dimensions matter when the system operates under budgets, quotas, or strict response-time requirements.

Efficiency is not a universal good. In high-uncertainty tasks, a slightly longer path can be the right trade-off because it reduces the chance of a wrong action. The evaluator should not reward brevity mechanically. It should reward only the steps that materially improve confidence, safety, or correctness.

### **Use partial credit for progress, not for accidental drift.**

Long-horizon tasks rarely fail all at once. The agent may complete the first subgoal, stall on a later one, or recover only part of the desired state. Progress-oriented evaluation is useful here because it captures meaningful advancement without pretending that partial completion is full success. This is the logic behind partial credit: you want to recognize subgoal attainment when it genuinely reduces the remaining problem.

That said, partial credit needs tight boundaries. It should reward genuine progress toward the stated objective, not incidental motion. If an agent collects a useful identifier and then misuses it, the collected identifier counts as progress only if it remains available and correctly integrated. If a run takes many steps but never narrows the search space or stabilizes the task state, it should not receive progress credit just because it was busy.

This distinction matters in workflows with long planning horizons. A system that identifies the correct customer record, confirms the relevant policy, and then fails during update is better than one that never found the record at all. AgentBoard-style progress framing is helpful precisely because it lets you represent that difference. The evaluator can then separate subgoal achievement from overall completion and avoid the all-or-nothing blindness of final-answer grading.

### **Balance robustness against process rigidity.**

The central trade-off in agent evaluation is between robust success and strict process conformance. If you require an exact tool sequence, you may reject valid alternative plans. If you accept any successful ending, you may reward unsafe or inefficient behavior. The right balance depends on the task's risk profile.

For a tightly specified workflow, exact or near-exact process conformance is often justified. In such a workflow, the order of tool calls carries meaning: validate before write, authenticate before access, fetch before transform. Trajectory matching makes sense because the valid path is well defined and deviations are likely bugs.

For a more open-ended assistant, exact path matching is too brittle. The agent may ask different clarifying questions, use alternative but valid tools, or reach the same endpoint through a different decomposition. In that case, the evaluator should use a softer judgment model that checks invariants instead of script identity. Did the agent use an acceptable tool family? Did it preserve the hard constraints? Did it avoid forbidden actions? Did it eventually converge?

The strongest harnesses combine both ideas. They enforce strict checks on non-negotiable constraints and permissive checks on legitimate variation. That gives you room to preserve robustness without turning the benchmark into a single scripted conversation or a single canonical plan.

**Anchor release decisions in the failure mode, not the aggregate score.**

Once you score the dimensions separately, you can map them to release decisions. That mapping should depend on the operational failure mode you care about.

If task completion is high but tool correctness is weak, you should not ship into a side-effecting workflow. The run may look fine from the user's perspective, but the trace reveals hidden risk.

If tool correctness is high but step efficiency is poor, you may still ship in a low-throughput environment, but you should expect higher cost and slower response. The decision becomes one of budget and latency tolerance, not correctness alone.

If plan quality is weak but completion is reliable, you need to ask whether the system succeeded by luck or whether the plan model is simply more flexible than the metric recognizes. This is exactly where trace inspection prevents false confidence.

If progress is strong but completion is incomplete, the run may still be useful as a signal that the system is learning the task structure, but it is not yet ready for unsupervised release. In that case, the harness can classify the miss as advisory rather than blocking. That classification keeps the evaluation honest without making it uninformatively binary.

A practical release workflow is simple: define the objective, specify the required tool families and forbidden actions, set thresholds for completion and efficiency, run the agent, inspect trace-level failures, and classify the miss by severity. If the miss is due to an unambiguous tool or policy violation, treat it as blocking. If the miss is a longer but acceptable path, treat it as advisory and decide whether the cost is tolerable for the target environment.

**Ground the benchmark in realistic task types.**

Executable benchmarks are more trustworthy when they reflect actual operational tasks rather than textual similarity. WebArena-style environments are useful because they force you to judge whether the agent actually completed the action in a live system, not whether it produced a plausible explanation of completion. That is especially important for agentic work, where a convincing answer can hide a failed action.

Visual and multimodal tasks raise the bar further. In a setting where the agent must reason from interface state, screenshots, or changing UI elements, intermediate observations may be the only way to explain a failure. A terminal answer alone can miss whether the agent misread a control, skipped a required interaction, or retried the wrong branch. VisualWebArena-style tasks make that visible because the intermediate state is part of the evidence.

Those benchmark styles reinforce the same principle: judge what the agent actually did, not just what it claimed. If the environment exposes the right evidence, the evaluator can distinguish between a correct completion, a lucky recovery, and a brittle run that only succeeded after too many wrong turns. That distinction is what turns traces into decisions.



## Chapter 6

# Scaling the Harness with Simulation, Synthetic Data, and Workflow Automation

*A mature DeepEval program eventually runs into a paradox: the better the team gets at finding meaningful failure modes, the faster manually curated benchmarks become too small, too slow, and too expensive to maintain. This chapter shows how to resolve that tension without sacrificing trust, by using synthetic and simulated cases to extend coverage, embedding evals into development and delivery loops, and treating release decisions as governed comparisons over time rather than reactions to isolated scores.*

## 6.1. Generating Synthetic Goldens Without Corrupting Benchmark Validity

When the failure modes are known but the hand-authored inventory is thin, synthetic goldens become tempting: you can scale coverage faster than a team can curate new examples, and you can target corners of the input space that real traffic exposes only rarely. The risk is that the same mechanism that expands coverage can also reshape the benchmark into something easier, narrower, or more aligned with the generator than with the product. That failure is subtle. Scores improve, but the suite stops representing the work the system must actually do.

A useful mental model is to separate *coverage proposals* from *trusted goldens*. A synthetic case is evidence that a behavior might matter. A promoted golden is evidence that the behavior is sufficiently real, stable, and relevant to deserve benchmark authority. Keeping that distinction explicit prevents the common collapse where every generated example is treated as equal truth. The promotion boundary is where benchmark validity lives.

**Start with a small, explicit promotion pipeline.** A disciplined workflow gives you room to generate aggressively without letting the benchmark drift. A minimal pipeline looks like this:

```
# Generate candidate expansions from a seed set.
deval synth generate \
  --seed-set data/goldens_seed.jsonl \
  --pattern edge-case \
  --max-items 200 \
  --out data/candidates.jsonl

# Remove duplicates against existing trusted cases.
deval synth dedupe \
  --base data/goldens_seed.jsonl \
  --candidates data/candidates.jsonl \
  --out data/candidates_dedup.jsonl
```

```
# Run rubric checks and calibration against the seed set.
deval synth review \
  --candidates data/candidates_dedup.jsonl \
  --calibrate-with data/goldens_seed.jsonl \
  --sample-rate 0.25 \
  --out data/review_queue.jsonl
```

That flow makes the governance boundaries visible. Generation produces candidates. Deduplication blocks trivial inflation. Review checks whether the candidate set is actually contributing new signal. Calibration against the trusted seed set gives you a reference for judging whether the synthetic items behave like plausible variations of the task or merely like artifacts of the generator.

### **Choose a generation pattern that matches the coverage gap.**

Different synthetic methods increase different kinds of coverage, and each method also introduces its own distortions.

Templated perturbation is the most conservative approach. You start with a human-authored golden and vary one dimension at a time: entity names, numeric values, ordering constraints, temporal qualifiers, or retrieval context. This works well when the system is sensitive to small changes in structure and when you want controlled regression tests. Its weakness is that it often preserves the same latent assumptions as the seed. The benchmark gets larger, but not necessarily broader.

Scenario recombination is better when the product risk comes from interacting dimensions. A support workflow may depend on account tier, region, refund policy, and device state. Recombination lets you generate valid combinations that rarely appear together in curated data. The danger is that logical validity does not guarantee operational plausibility. If a scenario is rare because the product never emits that combination, the benchmark may become artificially adversarial.

Controlled paraphrase generation increases lexical diversity and can expose brittle parsing or prompt-overfitting. It is useful for systems that should be invariant to surface form. It is weak for behavioral diversity: the semantics often remain nearly identical, so the added value is mostly linguistic. If paraphrase dominates the synthetic pool, you may accidentally reward models that memorize broad task shape while missing models that handle the actual operational variety.

Edge-case expansion around known failures is the most product-focused method. You take an observed defect and systematically push it into neighboring conditions: truncated context, ambiguous referents, partial tool failures, contradictory instructions, or near-threshold values. This is high value when you are hardening a release. It is also the easiest path to over-indexing the suite toward adversarial difficulty. A benchmark composed mostly of edge cases can obscure how the system behaves on ordinary traffic.

Contrastive generation creates near-miss pairs: one case should pass, the other should fail for a specific reason. This is useful when you need fine-grained discrimination between similar behaviors, especially for rubric design. It also helps expose whether a grader is sensitive to the failure mode you care about. If the system and the grader both pass the easy positive and the subtle negative, you have evidence of weakness in the evaluation design itself.

The right choice is rarely one method in isolation. You generally want a portfolio: conservative perturbations for regression stability, scenario recombination for breadth, edge-case expansion for known risks, and contrastive pairs for rubric sharpness. The mix should reflect the production distribution, the observed failure landscape, and the maintenance budget you are willing to carry.

**Treat provenance as part of the case, not as metadata you can ignore.** Synthetic items become hard to trust when you lose track of how they were produced. Provenance should record the seed case, the generation method, the generating model or rule set, the prompt or template version, the time of creation, and any filters applied during post-processing. That record lets you audit the case later when a score changes or when a reviewer questions why the benchmark contains a particular item.

Provenance also helps you detect generator leakage. If a family of synthetic cases all carry the same phrasing conventions, the same ordering of fields, or the same distractor style, then the benchmark may begin to measure a model's familiarity with your generator rather than its skill on the task. This is especially dangerous when one model family generates cases and a similar model family grades them. Shared priors can produce false confidence. A model can look strong because the benchmark has drifted toward its own style.

**Use distribution checks before you accept novelty as value.**

Novelty alone is not a reason to promote a synthetic item. The candidate pool should be checked against the distributions that matter for the product: input length, label balance, task subtype, entity types, temporal range, locale, error modality, and difficulty tier. A synthetic set that fills one rare slice while flattening everything else may be useful for exploration but harmful for regression governance.

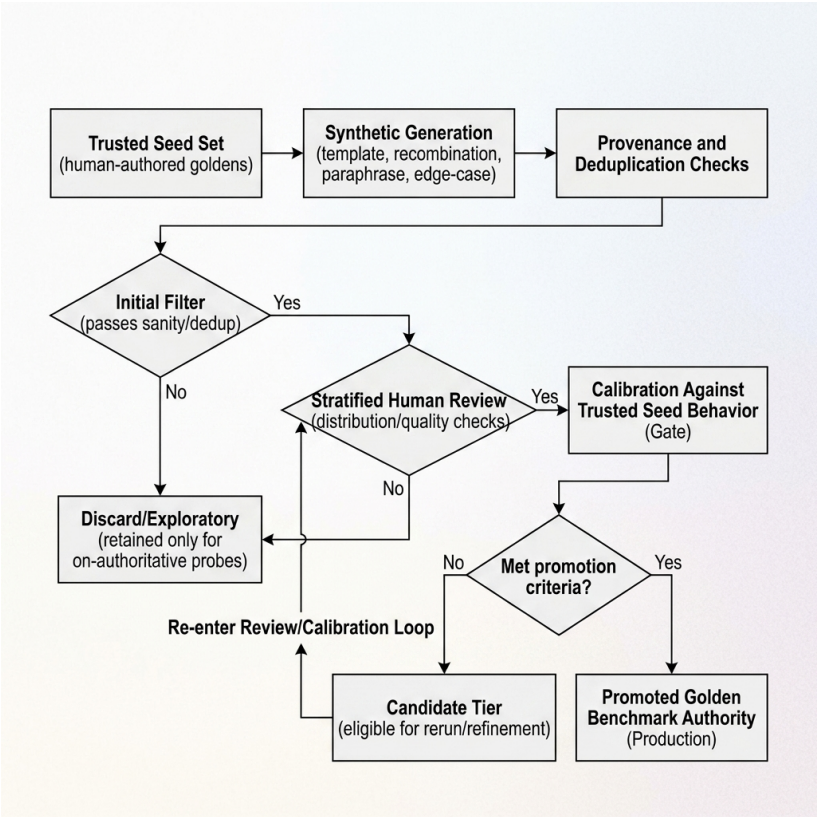
Distribution checks do not require perfect statistical matching. They require you to ask whether the synthetic batch is still anchored in the product's operating envelope. If all generated cases are longer than production traffic, or if every edge case includes an adversarial twist that real users never create, you are no longer broadening coverage. You are changing the task.

A practical heuristic is to inspect both *marginal* and *conditional* distributions. Marginal checks tell you whether the overall batch looks plausible. Conditional checks tell you whether the rare subpopulations still resemble the real contexts in which the system operates. A good synthetic suite often intentionally over-samples a failure region, but it should do so transparently, with the over-sampling labeled as an evaluation decision rather than an accidental shift in benchmark semantics.

**Sample human review, do not outsource authority.** Human review remains the gate between generated possibility and benchmark trust. You do not need to inspect every synthetic case, but you do need a review plan that is intentional enough to catch systematic failure. Stratified sampling works well: sample from each generation method, each difficulty band, and each proposed label. Reviewers should check three things: the scenario is realistic enough to matter, the expected outcome is justified, and the case is not a duplicate or near-duplicate of an existing golden.

When review is expensive, focus it on boundary cases. Review the items that the automated checks are least confident about: low-novelty paraphrases, recombined cases with unusual interactions, and candidates whose expected outputs depend on subtle policy interpretation. The goal is not perfection. The goal is to prevent obviously broken items from entering the trusted tiers and to catch hidden patterns that automated filters miss.

If you have multiple reviewers, measure agreement on a small calibration set before expanding the queue. Low reviewer agreement is not just a personnel issue; it often indicates that the rubric is underspecified or that the candidate cases are too ambiguous to serve as stable goldens. In that situation, keep the cases as exploratory probes rather than forcing promotion.



**Calibration makes synthetic data comparable to human-authored data.** A synthetic case is useful only if you can interpret its score relative to the seed set. Calibration does that work. You run the candidate batch and the trusted human set through the same scorer, then compare not just the aggregate score but the score shape across slices. If synthetic cases are consistently easier, the generator may be under-sampling ambiguity. If they are consistently harder, the generator may be injecting adversarial structure that does not reflect the production mix.

Calibration is most informative when the same rubric can score both sources. For example, if the task admits objective outputs, compare exact-match or execution-based outcomes on the seed and synthetic sets. If you rely on a model-based grader, check whether the grader's confidence or score distribution shifts materially across the two sources. A stable grader should not become more lenient on synthetic paraphrases simply because the surface forms are more regular.

This comparison gives you a promotion rule: a synthetic batch should not enter the governed benchmark until its behavior relative to the seed set is well understood. Not every promoted case needs identical difficulty, but the batch as a whole should preserve the same meaning of success and failure. That is the difference between expanding the evaluation space and rewriting it.

**Label confidence tiers to prevent category errors.** A practical benchmark often needs three tiers. *Exploratory* items are generated to discover candidate failures and to stress the evaluator. They are allowed to be messy, redundant, or highly adversarial. *Candidate* items have passed basic deduplication and review and may be used for diagnostic runs. *Promoted* items have survived calibration and sampling review and may participate in regression or release decisions.

The tier labels matter because they stop teams from over-reading immature evidence. Without them, a high score on exploratory synthetic data can masquerade as proof of robustness. With them, the same result is correctly interpreted as a sign that the candidate space is worth mining further. The tiers also make benchmark maintenance easier. When a promoted item later drifts or becomes too familiar, you can demote it without pretending the entire dataset has changed status.

**Watch for the failure modes that quietly corrupt validity.** The most common corruption pattern is *style leakage*. If the generator

prefers a particular phrasing, ordering, or constraint structure, then the benchmark starts to reward models that adapt to that style. A related failure is *difficulty inflation*, where the synthetic batch disproportionately contains tricky or contrived cases that the product rarely sees. This can make the system look worse than it is, which sounds safer than inflation but is just as damaging when it drives bad engineering decisions.

A second failure mode is *benchmark narrowing*. It happens when the easiest way to automate generation is to produce many variants of the same underlying task form. The benchmark grows, but its coverage radius shrinks because the surrounding task space remains untested. Another risk is *promotion by convenience*: a case becomes trusted not because it is validated, but because it was easy to generate and cheap to keep.

The most dangerous pattern is to let synthetic goldens dominate release gating before they have demonstrated longitudinal stability. A benchmark should not become more synthetic faster than it becomes more trustworthy. If the suite changes every week while the release policy assumes long-term comparability, the governance layer is built on moving sand.

**Keep the operational contract simple.** A reliable operating pattern is easy to remember: begin with a manually trusted seed set, generate narrow expansions tied to explicit hypotheses, deduplicate and filter the candidates, review a sampled subset, calibrate the survivors against the seed behavior, and then promote only the cases that preserve the benchmark’s intended meaning. When the product changes, revisit the hypotheses, not just the prompts.

That contract is stable even when the underlying DeepEval APIs, dataset loaders, or case serialization formats evolve. The specific

commands may change across releases, but the governance rule does not: synthetic generation is a method for discovering and shaping evidence, not a shortcut for redefining truth. When you enforce that boundary, synthetic goldens become a force multiplier instead of a source of quiet benchmark decay.

## 6.2. Simulating Multi-Turn Conversations for Targeted Failure Coverage

A system can look solid in single-turn evaluation and still fail in the exact way users experience it: not by one catastrophic reply, but by a sequence of small misses that compound. The assistant forgets a constraint, the user corrects it, the model half-recovers, and the final answer sounds plausible while the trajectory has already gone off the rails. Multi-turn simulation exists to expose that kind of failure. Its value is not in producing more dialogue for its own sake; its value is in generating trajectories that pressure persistence, repair, clarification, memory, and tool use under controlled conditions.

The practical temptation is to treat simulation as roleplay and assume realism will emerge from a prompt. That approach usually creates synthetic theater. The simulator becomes overly helpful, mirrors the assistant's wording, and resolves ambiguity too quickly. The result is a test set that rewards the exact conversational habits that production users do not exhibit. To avoid that trap, you need to treat the simulator as a bounded system with explicit controls, not as a free-form actor.

**Lead with the risk you want to surface.** The most reliable simulator design starts with a product failure hypothesis. If the product fails when a user revises requirements midway, then the simulator should generate revision-heavy trajectories. If the product loses state across tool calls, then the simulator should carry long-horizon constraints and

verify that the assistant preserves them. If the product mishandles policy-sensitive questions, then the simulator should probe the boundary without turning every conversation into a red-team exercise.

A concrete setup can look like this:

```
deval sim run \  
  --persona "busy_returning_user" \  
  --goal "reschedule_delivery" \  
  --constraints "late,annoyed,needs_refund" \  
  --turns 8 \  
  --scenario "clarification_and_repair" \  
  --out data/dialogs.jsonl
```

That command is deliberately narrow. You specify a persona, a goal, and the conversational stressors you want to test. The simulator then produces a trajectory, not just a prompt. The value of the run depends on how well those controls map to real failure modes.

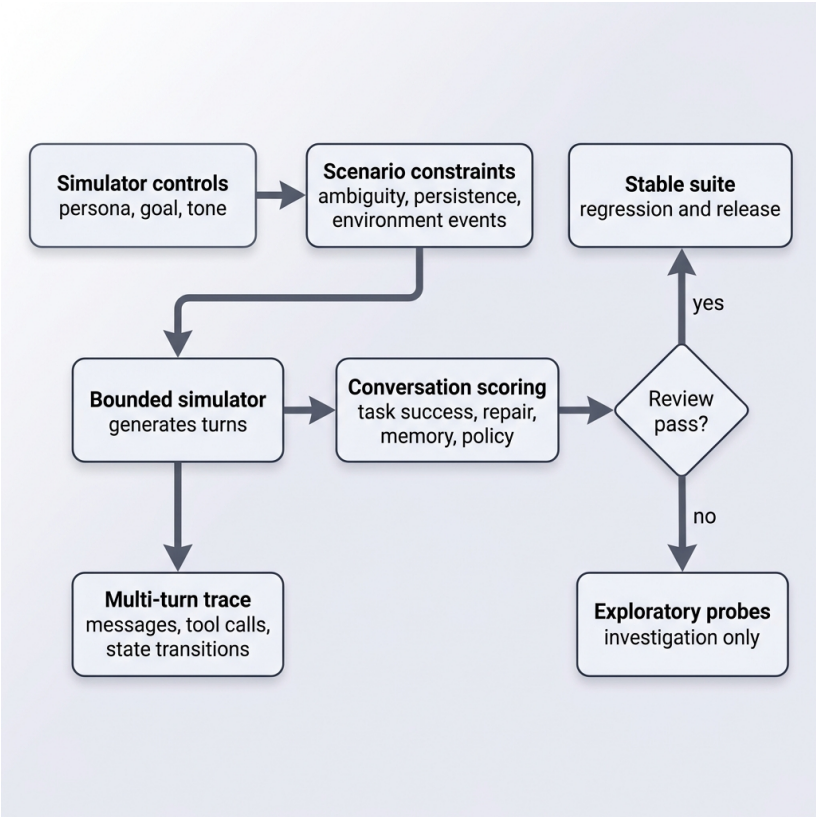
### **Model the simulator as a set of controls, not a single prompt.**

A useful simulator exposes several dimensions you can vary independently:

- **Persona** — Prior knowledge, domain familiarity, and style.
- **Goal clarity** — Whether the user knows the objective from the start.
- **Emotional tone** — Patience, frustration, urgency, or neutrality.
- **Persistence** — How often the user repeats or restates a need.
- **Ambiguity level** — How much the user leaves unsaid initially.
- **Correction behavior** — How the user reacts to misunderstandings.
- **Environment** — Missing fields, changing requirements, tool outages.

These controls matter because they determine the shape of the trajectory distribution. A cooperative user with perfect memory will make almost any assistant look competent. A simulator that randomly injects chaos will make almost any assistant look unreliable. You want neither extreme. You want the controlled variation that matches the real product risks.

The cleanest way to think about the simulator is to separate *hidden state* from *observable turns*. Hidden state includes the user's true goal, tolerance for delay, and willingness to accept partial answers. Observable turns are the actual messages exchanged. If the simulator leaks hidden state too aggressively, it starts helping the assistant rather than challenging it. If it hides too much, the trajectory loses coherence and stops resembling a real user session.



**Choose scenario families from failure hypotheses.** Multi-turn simulation is most useful when you generate dialogue families that each target a distinct trajectory risk. The families below are not exhaustive, but they cover the most common product failures.

*Happy-path completion* checks whether the assistant can carry a straightforward request to completion without unnecessary detours. Use it sparingly. It is easy to over-test the nominal path and under-test the edge cases, which produces reassuring scores and little diagnostic value.

*Clarification-heavy interactions* force the assistant to ask for missing information, preserve context, and incorporate answers later in the dialogue. These cases reveal whether the system can maintain a coherent task model across interruptions. They are especially useful in workflows where the user initially gives an underspecified request.

*Recovery after misunderstanding* starts with an assistant error: wrong assumption, wrong field, wrong tool, or wrong policy interpretation. The simulator then corrects the record and watches whether the assistant recovers cleanly. This family is important because production systems often fail not on the first mistake but on the second attempt to repair it.

*Escalating frustration* tests how the assistant behaves when the user becomes impatient. The goal is not to elicit rude language for its own sake. The goal is to see whether the assistant remains disciplined, concise, and task-oriented under conversational pressure.

*Policy-boundary probing* checks whether the assistant refuses, redirects, or qualifies responses correctly when the user asks for something that should trigger safety, legal, privacy, or business-policy constraints. Here, the simulator should be realistic enough to resemble a genuine user rather than an adversary trying to break the system.

*Long-horizon memory tests* stretch the dialogue across enough turns that earlier commitments matter. The simulator may revisit a preference, a constraint, or a partial completion several turns later. These trajectories expose systems that answer each turn well but fail to preserve state over time.

For each family, write down the failure hypothesis before you generate a batch. That discipline stops simulation from turning into a generic conversation factory. You are not trying to cover “multi-turn” as a category; you are trying to expose specific defects that single-turn tests cannot see.

**Keep realism and control in tension.** The hardest simulator decision is how much freedom to give the user model. High realism usually means richer behavior, but it also means more variance, more annotation burden, and more difficulty reproducing failures. High control gives you repeatable test cases, but it can produce stiff conversations that the assistant handles unusually well.

You can manage that tension by writing simulator instructions that privilege intent consistency over stylistic creativity. The simulated user should not improvise new goals unless the scenario explicitly calls for changing requirements. It should not echo the assistant's phrasing. It should not volunteer information simply because the assistant asked a broad question. And it should not over-explain when a real user would answer tersely.

A robust simulator prompt usually includes these constraints:

- Keep the user's goal fixed unless the scenario explicitly introduces a change.
- Reveal only the information a real user would reveal at that turn.
- If the assistant misunderstands, correct it in plain language, not with meta commentary.
- Preserve persona and tone across turns.
- Do not help the assistant complete hidden state.

That last line matters. A common failure mode is hidden cooperation: the simulator begins to act like a second engineer debugging the assistant. The conversation then becomes easier than real traffic and stops measuring the product behavior you care about.

**Generate traces, not isolated exchanges.** A multi-turn test case is a trajectory. You should store the full trace: user turns, assistant turns, tool calls, intermediate state transitions, and any simulator decisions that shaped the next user message. Trace-level representation lets you inspect where the interaction diverged from the expected path. Final-answer grading alone hides whether the assistant failed on clari-

fication, tool selection, memory retention, or policy handling.

This matters for debugging. If a three-turn dialogue fails because the assistant ignored a constraint on turn two, a final pass/fail score tells you almost nothing about the root cause. A trace scorer can isolate the issue to the missed repair step, which is the signal you need to fix the system.

When the system uses tools, keep the simulated user consistent with the tool outcome. If a tool returns no inventory, the user should not magically accept a successful reservation. If a retrieval step returns a partial answer, the user should challenge that partiality only when the scenario requires it. Otherwise, the simulator becomes a source of inconsistency rather than a test of the assistant.

**Score the conversation at the level of behavior.** A useful multi-turn evaluation usually combines several metrics rather than one aggregate judgment. The exact metric set depends on the application, but the common categories are stable.

| Metric class         | What it detects                                                          |
|----------------------|--------------------------------------------------------------------------|
| Task success         | Whether the user goal was actually met.                                  |
| Repair quality       | Whether the assistant recovers after error.                              |
| Context retention    | Whether prior facts persist across turns.                                |
| Policy compliance    | Whether forbidden actions are avoided.                                   |
| Tool discipline      | Whether the right tool is used at the right time.                        |
| Conversation economy | Whether the assistant resolves the task in a reasonable number of turns. |

Some of these metrics can be scored automatically. Others need rubric-based or model-based judgment, especially when the success criterion depends on nuanced conversational state. The key is to keep the metric interpretation aligned with the trajectory. A conversation that finishes correctly but violates policy in the middle is not a clean pass. A trajectory that ends early because the assistant refuses a valid task is not a success. The score must preserve the shape of the interaction, not just the end state.

Trace-level evaluation is especially useful here. It lets you score intermediate events such as “asked a clarifying question before acting,” “preserved the user’s stated date,” or “recovered after incorrect assumption.” Those intermediate scores often diagnose more than the final outcome.

**Ground the simulator in real workflow structure.** Simulation improves when the dialogue is anchored to actual business processes rather than generic chat. A customer-support conversation, a scheduling workflow, and a medical intake flow have different state requirements, different error patterns, and different notions of completion. If you simulate them all with the same free-form roleplay style, you erase the details that matter.

The better approach is to model the workflow as a sequence of stateful decisions. The simulator knows what information must be collected, what order is plausible, what branch conditions matter, and what a failure looks like at each step. The user messages then emerge from that structure. That makes the generated trajectory easier to audit and more likely to match real operational behavior.

For example, a delivery-rescheduling simulation may require the user to specify order ID, address confirmation, time window, and change-of-plan constraints. If the assistant proposes a new time, the simulator should react according to the hidden state: accept, reject, request another option, or correct a misread constraint. That is a better test than a generic “change my delivery” chat because it pressures the exact state transitions the product must handle.

**Sample the right conversations for manual review.** Simulation still needs human inspection. You do not need to review every generated dialogue, but you do need a calibration sample that cuts across the space where the simulator is least certain. Review conversations that involve contradictory signals, long turn sequences, rare persona

settings, or unexpected assistant tool behavior. Those are the places where synthetic conversations most often drift away from useful realism.

When reviewing, look for three classes of defect. First, the conversation may be unrealistic: the user speaks too politely, gives too much information, or resolves ambiguity too quickly. Second, the trajectory may be mislabeled: the expected outcome does not match the actual business rule. Third, the simulator may have overfit to the assistant: it changes behavior in response to the assistant's wording rather than preserving its own intent.

You should also compare simulator behavior against a small human-validated reference set. If the simulated conversations systematically differ in length, sentiment, correction style, or completion rate, the simulator may still be useful for regression probes but not yet trustworthy enough for promotion into the stable suite.

**Promote only trajectories that survive calibration.** Not every simulated conversation should become a governed test case. The cleanest operational pattern is to classify generated trajectories into exploratory probes, candidate dialogues, and promoted suite members. Exploratory probes help you discover what the simulator can express. Candidate dialogues have passed basic plausibility checks and can support debugging. Promoted dialogues have been reviewed, calibrated against human reference behavior, and judged stable enough for regression use.

That tiering protects the benchmark from overcommitment. It also makes maintenance tractable. If a simulator is later updated to reflect new product behavior, you can regenerate candidate dialogues without silently rewriting your release gates. The stable suite changes slowly; the exploration pool can change quickly.

**Watch the failure modes that make simulation look better**

**than it is.** The most common error is the *overly helpful simulator*. It asks leading questions, accepts partial answers, and quickly converges on the assistant's preferred phrasing. That behavior inflates success rates and hides repair failures. A related problem is the *stylistically narrow simulator*, which uses the same tone and sentence structure across many personas. The assistant then learns to respond to the simulator rather than to a varied user population.

Another failure mode is unrealistic turn length. If every user message is long and perfectly structured, the assistant is tested on a convenient version of the problem. Real users interrupt themselves, omit context, repeat themselves, and sometimes contradict their earlier statements. Your simulator should be capable of those patterns when the scenario calls for them.

Finally, beware of synthetic trajectories that produce only polite resolution. If every conversation ends with mutual agreement, the simulator is not probing persistence, refusal, correction, or conflict resolution. It is only generating successful dialogues. That may be useful for smoke testing, but it will not expose the trajectory failures that make multi-turn systems brittle.

**Keep the implementation contract explicit.** A practical simulation workflow is straightforward when you separate concerns. First, define the product risk. Second, encode the simulator profile and scenario constraints. Third, generate a batch of dialogue traces. Fourth, score the traces with conversation-level metrics. Fifth, sample a subset for manual review. Sixth, promote only the trajectories that remain stable under calibration.

The exact DeepEval helpers and serialization formats may vary across releases, but that workflow does not change. The simulator should produce bounded trajectories, the evaluator should inspect behavior over time, and the promotion rule should be stricter than mere generation

success. Used that way, simulation becomes a controlled way to discover conversational defects that single-turn benchmarks cannot see, while avoiding the false confidence that comes from beautiful but unrepresentative dialogues.

### 6.3. Building Local Evaluation Loops and CI/CD Gates That Engineers Will Actually Use

When evaluation stays outside the development loop, it becomes a report people glance at instead of a control surface they rely on. The failure is usually not technical sophistication; it is latency. If a change takes an hour to reach a meaningful result, engineers stop using the suite for debugging. If every check runs everywhere, they stop using it for delivery. The working system therefore has to be tiered: a fast local slice for immediate feedback, a pull-request gate for regressions, a broader scheduled run for drift, and a small release-blocking set that is stable enough to trust.

The central design choice is to make every loop answer a different question. Local evaluation asks whether you broke the thing you are touching. CI asks whether the change is safe relative to the core guarantees. Scheduled runs ask whether broader quality, cost, or safety is drifting over time. Release gates ask whether the current evidence justifies promotion. If you collapse those questions into one pipeline, you get either a slow pipeline nobody runs or a weak pipeline nobody trusts.

**Start with a fast local subset.** The local loop has one job: let you test the change you just made before you pay the cost of a full run. It should be narrow, high-signal, and cheap. In practice, that means a small subset of stable cases, ideally focused on the component you edited. If you changed a retrieval prompt, you should not wait for a

full end-to-end suite to tell you that top- $k$  recall broke. If you changed a tool router, you should not re-run every synthetic dialogue in the catalog.

A practical pattern is to mark cases by purpose and cost:

```
import pytest

@pytest.mark.smoke
@pytest.mark.retrieval
def test_retrieval_smoke():
    ...

@pytest.mark.regression
@pytest.mark.slow
def test_full_dialogue_flow():
    ...

@pytest.mark.xfail(reason="Known tool timeout")
def test_known_flake():
    ...
```

These markers let you run the right slice without editing the suite itself. Smoke tests stay small and fast. Regression tests preserve the broader guarantees. Slow tests remain available for scheduled or pre-release runs. Known-bad cases stay visible through `xfail` instead of being deleted and forgotten. That visibility matters: if you hide a failing case, you lose the signal that the issue still exists.

The local subset should also be selective by subsystem. If your harness can infer which files changed, use that information to choose the relevant cases. A prompt edit may only need prompt-sensitive tests. A change in tool schema may only need tool-integration tests. You reduce waste by matching the test slice to the changed surface area.

**Make the pull request gate small and stable.** The pull-request pipeline has a different purpose. It is not there to exhaustively validate the system; it is there to stop unsafe changes from merging. That gate should be small enough to run routinely, stable enough to avoid flake

fatigue, and high-confidence enough that engineers understand why it failed. If the gate becomes noisy, people will learn to work around it rather than through it.

A good PR gate typically contains three classes of checks: a smoke tier for the changed component, a small regression tier for core guarantees, and any specific high-risk cases tied to the modified subsystem. You can encode that with `pytest` selection or equivalent harness filtering:

```
pytest -m "smoke or regression" \  
--maxfail=1 \  
--durations=10
```

That command does not ask the engineer to wait for everything. It asks for enough signal to catch the obvious breakage and enough traceability to diagnose it quickly.

The major trade-off is between component-level gating and end-to-end gating. Component tests are fast and precise, but they can miss integration failures. End-to-end tests capture the real workflow, but they are slower and noisier. Use component tests for merge-time defense and reserve full end-to-end coverage for scheduled runs or release candidates. If you gate every merge on a wide end-to-end suite, you will eventually gate on flakiness rather than quality.

**Use control logic in CI, not just a static test list.** GitHub Actions and similar CI systems let you scope execution by branch, path, event, and expression. Use that control logic aggressively. Expensive evaluation should run only when relevant. A tool-routing change does not need the same suite as a prompt formatting fix. A documentation-only change should not consume the same compute budget as a policy update.

A representative workflow might look like this:

```
name: eval
```

```
on:
  pull_request:
    paths:
      - "src/**"
      - "prompts/**"
      - "evals/**"
  push:
    branches: [main]

jobs:
  smoke:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pytest -m smoke

  regression:
    if: github.event_name == 'pull_request'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: pytest -m regression
```

The point is not the specific syntax. The point is that you do not pay for breadth unless the change touches the area breadth is meant to protect. Path filters reduce unnecessary runs. Branch filters separate merge defense from mainline monitoring. Expression-based gating lets you decide when an expensive suite is worth the cost.

One subtle failure mode deserves attention: skipped or pending checks can block merges if branch protection expects them. If a workflow is configured as required but does not trigger, the merge can stall even though nothing is wrong with the code. Avoid that by making sure the required checks are always materialized for the relevant events, or by scoping protection rules to checks that actually run on the protected branch and pull-request path. Merge queues add another wrinkle: if you use them, wire the merge-group event explicitly or you may validate the wrong ref and create a false sense of coverage.

**Treat flake management as an engineering problem, not a nuisance.** Evaluation suites involving model calls, stochastic graders, retrieval, or external services will eventually exhibit flake. Ignoring it is the fastest way to destroy trust. The better response is to partition the failure modes.

Some checks are deterministic but expensive. For those, use runtime caps, caching, and sharding. Other checks are probabilistic but stable enough for thresholds. For those, keep a small confidence margin and avoid binary gating on a single noisy draw. Still others are genuinely flaky because the environment is unstable. For those, either mock the dependency or move the case out of the release gate entirely.

A useful operational rule is that release-blocking tests should be repeatable enough that rerunning them rarely changes the answer. If a case often flips between pass and fail without code changes, it is not a good gate. It may still be a diagnostic signal, but it should not determine merge eligibility.

When a suite fails, emit reason-rich artifacts: the exact prompt or trace, the grader output, the versioned dataset slice, and the model or tool configuration. Engineers do not want another red check; they want a path to diagnosis. If the harness makes failure legible, adoption rises quickly. If it makes failure opaque, people eventually bypass it.

**Use runtime architecture to keep breadth affordable.** The only way to make broader suites sustainable is to control cost structurally. Three mechanisms matter most: sharding, caching, and selective execution.

Sharding splits a large suite into parallel chunks so a ten-minute run behaves like a much shorter wall-clock check. That helps both PR latency and scheduled regression throughput. Caching avoids recomputing unchanged intermediate results, especially for retrieval, embedding, or

fixed prompt inputs. Selective execution filters the suite by changed subsystem, changed dataset tier, or changed risk class.

Those mechanisms interact. A cached retrieval step that feeds a sharded evaluator can reduce both wall time and GPU cost. But caching also creates consistency obligations. You need cache keys that include the model version, prompt version, dataset version, and any critical harness parameter. Otherwise, you risk replaying stale evidence under a fresh configuration.

A clean versioning rule is to treat every changing artifact as part of the test identity:

| Artifact        | Why version it                                    |
|-----------------|---------------------------------------------------|
| Dataset slice   | Keeps historical comparisons meaningful.          |
| Prompt template | Prevents silent behavioral drift.                 |
| Grader rubric   | Preserves score interpretation.                   |
| Model config    | Distinguishes product change from harness change. |
| Tool schema     | Exposes interface-level regressions.              |

If you cannot reproduce a run from its versioned inputs, you do not have an evaluation system; you have a logging system.

**Choose the right gate for the right decision.** Not every evaluation result should block a merge. A release-blocking suite should be small, stable, cheap, and strongly tied to core guarantees. A diagnostic suite can be larger, noisier, and more exploratory because it informs investigation rather than gating. A scheduled suite can be broad and expensive because it is run asynchronously. Engineers tolerate expensive evaluation when they understand its purpose and cadence.

That distinction becomes especially important when synthetic or simulated cases enter the picture. Exploratory cases are valuable for discovery, but they are often too unstable to sit on the hard merge path. If a synthetic case reflects a newly discovered failure mode, keep it in a diagnostic tier until it has been reviewed, calibrated, and shown to behave consistently. Otherwise, you risk blocking delivery on a case

whose own status is still provisional.

Similarly, do not overload the gate with every metric you can compute. A gate with ten unrelated scores is harder to reason about than a gate with three meaningful thresholds. Use the smallest set that captures the core risk: perhaps a task-success metric, a policy metric, and a cost or latency guardrail. More metrics are useful in dashboards, but more metrics in a blocking path usually reduce clarity.

**Design the local loop so engineers can explain it.** Adoption depends on legibility. If a developer cannot predict which tests will run, how long they will take, or why a failure matters, the harness feels arbitrary. You increase legibility by making suite labels meaningful, keeping naming consistent, and exposing the mapping from change type to test slice.

A practical local flow looks like this:

```
$ pytest -m smoke
3 passed, 0 failed in 6.2s

$ pytest -m "smoke or regression"
12 passed, 1 xfailed in 31.4s

$ pytest -m "regression and not slow"
9 passed, 0 failed in 18.7s
```

These outputs are not just status lines; they are a contract. The engineer sees what ran, how much was skipped, and whether the result was fast enough to use during iteration. If a local run routinely takes long enough to interrupt the editing loop, the suite should be broken into smaller slices or moved upward into a CI tier.

**Separate known failures from unstable failures.** The distinction between skip and xfail is operationally useful. A skipped case

is irrelevant in the current environment or configuration; an expected failure is a known defect that you still want to keep visible. That difference prevents the suite from becoming a graveyard of silently disabled checks.

Use `skip` when the dependency is absent, the platform is incompatible, or the configuration makes the case meaningless. Use `xfail` when the behavior is still broken but should remain on the radar. This distinction helps with roadmap prioritization: skipped cases are usually maintenance issues; xfailed cases are product issues.

You can also encode time-bounded expectations. A known failure can carry an expiration date or a ticket reference so that it does not remain indefinitely tolerated. That keeps the suite honest without making the pipeline unusable.

**Make scheduled runs do the expensive work.** Broader evaluation belongs in scheduled or pre-release runs, where wall-clock time and cost are less damaging to developer flow. That is where you run the full regression inventory, the deeper synthetic and simulated sets, and the historical comparisons that would be too expensive or too noisy for merge time. Scheduled runs are the place to measure drift, not to catch every individual typo.

Because these runs are asynchronous, they can include more trace collection, more human inspection sampling, and more slice-level analysis. They also make it easier to compare candidates against baselines over time, which is the right place to ask whether quality is improving or merely shifting.

The practical mistake is to use scheduled runs as a substitute for local and PR feedback. If the only meaningful signal arrives overnight, engineers have already moved on. Scheduled runs should deepen governance, not replace immediate control.

**Keep the governance story simple.** A healthy evaluation workflow has a readable decision path: a developer runs a small local slice, CI checks the relevant regression and smoke tiers, broader scheduled runs monitor drift, and a small release gate protects merge or deployment. Each layer answers a different question, and each layer uses a test size appropriate to that question.

That design does not eliminate cost or flakiness. It contains them. You still need versioned artifacts, explicit thresholds, clear naming, and careful event scoping. You still need to watch required checks so skipped or pending jobs do not block merges unexpectedly. You still need to revisit suite composition when the product changes. But when the control loop is short and the gates are legible, evaluation becomes part of engineering practice instead of a parallel compliance exercise.

## **6.4. Operating Historical Comparisons and Release Governance from Stable Evidence**

Once evaluation runs accumulate, the hard question stops being “What was the score?” and becomes “What changed, and does it matter enough to alter a release decision?” That question is only answerable if the evidence remains comparable across time. A high score is useful only when you can tell whether it reflects an application improvement, a benchmark change, a grader change, or a harmless shift in run context. Release governance turns on those distinctions, so the evaluation harness has to behave like a versioned measurement system, not a notebook of isolated results.

The operational failure mode is familiar. A team compares the latest run to an old dashboard, sees improvement on aggregate metrics, and approves release. Later, a slice that mattered to customers regressed, but the slice disappeared when the dataset was refreshed. The bench-

mark was not wrong; the governance model was incomplete. Historical comparison only supports release decisions when you preserve enough of the old run to know what the new run is actually being compared against.

**Lock the comparison surface before you inspect deltas.** Stable comparison begins with stable artifacts. At minimum, you should version the dataset, the metric definition, the grader rubric or prompt, the model configuration, and the retrieval or tool settings that shape the result. If you rely on traces, preserve them as well, because a final score without the underlying interaction path is often too thin to explain a release decision.

A practical way to encode that discipline is to treat every evaluation run as a bundle of versioned inputs and outputs:

```
run_id: eval-2026-05-15-rc3
dataset_version: goldens-v14
grader_version: rubric-v6
prompt_version: prompt-v9
model_version: gpt-4.1
retrieval_version: bm25-v3
trace_store: s3://eval-traces/2026-05-15/
thresholds:
  task_success: 0.92
  policy_compliance: 1.00
  latency_ms: 1200
```

This kind of manifest is not bureaucratic overhead. It is the only reason a later regression review can distinguish product movement from measurement drift. If you cannot reconstruct the run context, a score delta is at best an anecdote.

**Compare against stable baselines, not against whatever run happened last.** The most defensible release comparisons use governed baselines. A baseline is not just “the previous run”; it is a stable

reference set that the team has agreed to preserve long enough to make changes interpretable. That may be a release baseline, a protected hold-out slice, or a historical cohort representative of a known customer population. The key property is that it changes slowly and only with explicit review.

You can use several comparison modes, each with different governance value.

| Comparison mode        | What it answers                                                             |
|------------------------|-----------------------------------------------------------------------------|
| Point-in-time baseline | Did this candidate improve or regress relative to the last trusted release? |
| Rolling trend          | Is quality drifting over several runs?                                      |
| Slice cohort           | Did a specific user segment or task class move?                             |
| Policy delta           | Did a critical guarantee break, even if the average score improved?         |

Point-in-time comparisons are useful for release decisions. Rolling trends are useful for detecting slow drift. Slice cohorts are useful for diagnosis and accountability. Policy deltas are useful when some failures matter far more than the aggregate score suggests. If you only track averages, you will eventually ship a model that looks better overall while failing the cases that actually govern trust.

**Treat benchmark versions as first-class release objects.** A benchmark change is not the same as a model change. If you add synthetic cases, refresh a holdout slice, or rewrite a rubric, you have changed the measurement surface. That may be the right thing to do, but you must not compare new and old scores as if nothing changed. The governance rule is simple: if the benchmark changes materially, version it and record the reason.

That version boundary matters because metrics often respond to dataset composition more than teams expect. A fresh set may be harder, easier, broader, narrower, or more aligned with current product behavior. None of those outcomes is inherently bad. The problem arises when benchmark evolution gets mistaken for product

improvement. The score moves, but the meaning of the score moves too.

A good version record therefore includes both the change and the rationale. For example, “goldens-v14” might add new policy-boundary cases because the product expanded into a regulated workflow. That is a valid benchmark revision. It is not a reason to compare v14 directly to v13 without explaining the composition shift.

**Preserve the grader as carefully as the dataset.** Many teams version datasets but leave graders implicit. That is a mistake. A rubric change can alter scores just as strongly as a dataset change, especially when you use model-based or trace-based grading. If a grader prompt changes, or if a scoring threshold is tuned, historical comparison loses meaning unless you preserve the old grader and run it side by side.

Trace grading is especially useful for governance because it shows why a run passed or failed, not only whether it did. That extra structure helps you classify ambiguous cases. A conversation may end successfully but still violate a critical process step in the trace. A final-answer score alone can hide that problem. A trace-aware record lets you decide whether the failure belongs in a release gate or only in a diagnostic dashboard.

The same logic applies to logs-based evaluation. Historical production slices are valuable because they anchor the comparison in actual usage patterns, but only if the slice definition itself stays versioned. If the log filter changes, the resulting cohort may no longer be comparable to the one used for the previous release.

**Make threshold changes explicit and policy-driven.** A threshold is part of governance, not part of the metric. If you decide that task success must exceed 0.92 or policy compliance must be perfect, you

have encoded a release rule. Changing that rule is a governance decision, not a technical tweak. Teams often weaken thresholds after a regression to make the pipeline green again. That move hides the real question: did the product improve, or did the gate get looser?

A safer pattern is to separate three actions:

- changing the benchmark,
- changing the metric or grader,
- changing the release threshold.

Each has its own approval path. If you change the threshold, record who approved it, why it changed, and whether it is temporary or permanent. If you change the benchmark, specify whether the old version remains as a historical reference. If you change the grader, rerun a calibration set so you know whether the score movement reflects a real behavioral change or simply a new scoring interpretation.

This discipline prevents the common anti-pattern of “fixing” regressions by moving the target. When thresholds move silently, release governance stops being evidence-based and becomes negotiable in the worst possible sense.

**Use slice-level analysis to avoid aggregate blindness.** Aggregate metrics are useful, but they flatten structure. A model can improve on easy traffic and regress on the cases that matter most. If your release criteria look only at the average, you will miss this class of failure. Slice analysis gives you the leverage to see whether a regression is isolated to one workflow, one locale, one tool chain, or one policy class.

The right slice design depends on the product, but the principle is stable: define slices that correspond to real failure risk. For an assistant

that handles customer requests, the slices might be account changes, refunds, escalation, or ambiguous intents. For an agentic workflow, the slices might be tool availability, long-horizon memory, or multi-step recovery.

When a candidate regresses on one slice but improves overall, the decision depends on the severity of that slice and the confidence in the measurement. A small regression in a non-critical slice may be acceptable. A regression in a contractual, safety-critical, or high-volume slice usually is not. That is why the governance model needs policy-aware classification rather than a single global pass/fail bit.

**Require reason-rich artifacts for every blocked release.** If a run blocks a release, the failure report should answer three questions immediately: what failed, where it failed, and why the gate treats it as blocking. A bare score does not do that. You need the relevant traces, the slice membership, the baseline comparison, and the threshold that was violated.

A compact human-readable failure record might look like this:

```
Candidate: eval-2026-05-15-rc3
Baseline:  eval-2026-05-01
Slice:     refunds / policy-boundary
Metric:    policy_compliance = 0.94
Gate:      threshold = 1.00
Decision:  block
Reason:    one unsafe refund approval in trace 17
```

That format matters because engineers can act on it. They can inspect the trace, see the exact failure mode, and decide whether the problem is a bug, a benchmark artifact, or a justified policy change. If the report cannot support that level of diagnosis, the gate will eventually feel arbitrary.

**Write governance policies for ambiguous outcomes before the run fails.** Ambiguous results are the norm, not the exception. Candidate releases may improve on some slices and regress on others. Baselines may be stable overall but shaky in one high-value region. A policy needs to say what happens when evidence is mixed.

A useful policy typically defines three classes of outcome: pass, fail, and review required. Review required is important because not every release decision should reduce to an automatic gate. If the evidence is noisy, if the benchmark changed materially, or if the impacted slice carries high business risk, a human review step is the correct response.

The policy should also define who may rebaseline, what evidence is required to accept a threshold change, and when waivers expire. Otherwise, a temporary exception becomes permanent by inertia. Expiring waivers force the team to revisit the decision after the immediate pressure passes.

**Historical comparison is stronger when it includes production logs.** A stable benchmark is necessary, but it is not sufficient. Benchmarks can drift away from real usage if they are not refreshed against actual production behavior. Logs-based data sources help here because they let you evaluate candidate behavior on historical production slices. That gives you a comparison grounded in how the product was used, not just how it was imagined in a curated set.

Logs are especially valuable for debugging release decisions. If a candidate performs well on curated goldens but poorly on a production slice, the discrepancy often points to coverage gaps or benchmark staleness. If both curated and historical slices regress, the problem is probably real. The combination gives you a stronger basis for governance than either source alone.

You should still version the log slice definition. A historical production

cohort built from one time window is not comparable to a cohort filtered differently a month later unless you explicitly preserve the filter and the sampling logic. The same governance principles apply: stable definition, explicit versioning, and preserved execution context.

**Use run comparison tools, but do not outsource judgment to them.** Tools like MLflow make it easier to compare runs, inspect metrics, and keep a history of evaluation artifacts. That helps operationally, but the release decision still belongs to governance. Around MLflow 2.18.0, validation-related APIs changed enough that implementation planning needs version awareness, so you should verify the exact interface you deploy rather than assuming a stable call pattern across releases. The broader lesson is unchanged: whatever tool you use, preserve enough metadata to compare runs meaningfully, and keep the decision policy separate from the storage layer.

A tool can show you that a run improved by two points. It cannot tell you whether the improvement came from a benchmark change, a rerank tweak, or a harmless shift in the traffic mix. That interpretation belongs to the team, using the versioned evidence.

**Keep the release logic simple and auditable.** The strongest governance systems are usually the least ornate. A candidate release goes through the governed benchmark tiers. Slice-level deltas are compared to the protected baseline. Benchmark or grader changes are separated from application changes. The policy classifies the outcome as pass, fail, or review required. Any approved exception or rebaseline is recorded with a rationale and an expiration, if applicable.

That sequence is easy to explain and hard to game. It does not pretend that every metric movement is equally meaningful. It does not collapse all evidence into one dashboard number. It gives you a stable contract between engineering and release governance, which is the

only way historical comparison stays useful as the evaluation program grows. When the contract is explicit, a run history becomes a decision system rather than a score log, and the score itself retains its meaning from one release to the next.

## Chapter 7

# Calibration, Debugging, and Evolving a Trustworthy DeepEval Program

*A mature evaluation program fails not when a score drops, but when engineers stop believing what the score means. This chapter explores the hard part that begins after metrics are wired up: calibrating automated judgment against product risk, tracing failures to their true source, migrating confidently across framework changes, and preserving trust in the harness as models, prompts, datasets, and delivery workflows keep moving.*

## 7.1. Reading Scores as Evidence Instead of Truth

A benchmark can look calm while the product is misbehaving, or look alarming while the release is actually safe. That mismatch is not a bug in interpretation; it is the normal condition of measurement. You get a score because a metric, rubric, or judge model compressed behavior into a number or label under a set of assumptions. You do *not* get a verdict from nature. The practical discipline is to treat each output as evidence that supports or weakens a hypothesis about product behavior, with uncertainty attached. NIST frames AI evaluation as a measurement activity with explicit assumptions, limits, and risk tolerance, which is the right mental model for evaluation output in a trustworthy DeepEval program.

### A concrete reading pattern

When a run fails, start by asking what kind of evidence you are looking at, not whether the system is “right” or “wrong.” A scalar score is usually a summary statistic over a narrower predicate: exact match, rubric alignment, retrieval support, safety compliance, or process fidelity. A pass/fail bit is typically a thresholded decision layered on top of that statistic. A judge rationale is a textual explanation generated by another model, which can help you triage, but it is not the same thing as an execution trace or a causal diagnosis. If you want a disciplined reading workflow, use the record in layers:

```
from deepeval import evaluate
from deepeval.metrics import AnswerRelevancyMetric
from deepeval.test_case import LLMTestCase

metric = AnswerRelevancyMetric(threshold=0.80)

case = LLMTestCase(
    input="Summarize the refund policy.",
    actual_output="Refunds are available within 30 days.",
    expected_output="Refunds are available within 30 days."
)
```

```
evaluate([case], [metric])
```

That call returns a score, a pass/fail decision, and a reason string. The right first question is not “did it pass?” but “what evidence does the score supply about the hypothesis that this output satisfies the rubric under the current conditions?” If the answer is yes for a narrow, reference-based metric, that supports one conclusion. If the answer is no for a broader, rubric-driven metric, that may indicate a mismatch between the prompt, the context, and the evaluation criterion. Those are different failure modes, and they deserve different responses. DeepEval also supports standalone metric execution for debugging, but the fuller evaluation path is the one that gives you the reporting and optimization needed to interpret results consistently.

### **Signal, decision, and usefulness**

Separate the measurement signal from the operational decision. The signal is the metric output. The decision is what you do with it: block a release, require human review, file a regression, or merely log a warning. A score can be valid as a signal and still be operationally unhelpful. For example, a metric may correlate with human judgment on average but still be too noisy near the release boundary to drive an automatic gate. Conversely, a metric may have obvious blind spots but still be useful as an early warning system if you know which slice of behavior it covers.

That distinction matters because teams often over-ask one metric to do several jobs at once. A release gate needs low false-negative rate for high-risk failures. A developer-facing diagnostic needs speed and local actionability. A research dashboard may prefer trend sensitivity over hard decisions. One score cannot usually optimize all of those objectives simultaneously, so you should assign each metric a job and read it accordingly. NIST’s AI RMF emphasizes documenting risk tolerance where fixed standards do not exist; that applies directly to choosing

how much uncertainty you can tolerate in a gate versus an advisory metric.

### **What a low score actually means**

A low score means only that the output violated the metric’s criterion under the evaluator’s assumptions. It does not automatically mean the product is bad. It may mean the metric is too strict, the expected answer is underspecified, the retrieved context was incomplete, or the judge model is sensitive to wording that users do not care about. It may also reveal a genuine product defect. You need the surrounding context to decide which.

Read the same low score differently depending on metric family:

- A reference-based metric tells you the output diverged from a target answer or target facts. That is strongest when the gold answer is stable and the task is tightly specified.
- A rubric-driven metric tells you the output failed to satisfy a qualitative criterion such as helpfulness, faithfulness, or tone. That is useful for broader behaviors, but it is also more exposed to judge variation and rubric ambiguity.
- A retrieval-aware metric tells you whether the model used supporting evidence well, whether the retriever surfaced relevant context, or whether the generated answer stayed grounded.
- A process-oriented metric tells you something about the path, not just the final answer: did the tool call occur, did the agent follow the expected action sequence, did the conversation preserve state correctly.

The operational consequence is simple: do not collapse all low scores into one class called “failed.” A low retrieval score and a low safety

score point to different parts of the stack and different remediation paths. DeepEval’s tracing and component-level evaluation features are useful here because they let you inspect retrievers, tool calls, and generated outputs separately instead of inferring everything from the terminal response. Component-level evaluation is currently single-turn only, which is an important boundary when you interpret what the harness can and cannot localize directly.

### **False positives and false negatives**

Trustworthy evaluation is mostly the work of managing two kinds of mistakes. A false positive says the harness failed the system when the product behavior was actually acceptable. A false negative says the harness passed behavior that should have been blocked. Both matter, but they hurt differently.

False positives cost developer time, block releases unnecessarily, and eventually teach the team to ignore the suite. False negatives are more serious when the product has safety, compliance, or high-severity correctness requirements, because they let harmful behavior slip through. The relative cost depends on the domain. For a low-risk internal analytics assistant, a few false positives may be tolerable if they keep the suite agile. For a medical, legal, or customer-facing support system, false negatives are often much more expensive than release friction.

You can read evaluator behavior through that lens. A strict judge that over-flags stylistic imperfections may have good sensitivity but poor precision. A lenient judge may preserve throughput while missing narrow but important failures. The right threshold is not the one that feels cleanest; it is the one that matches the product’s risk profile. That is why threshold calibration should be based on labeled samples and explicit error costs, not inherited intuition. If you cannot explain which mistake is more expensive, you do not yet know how to use the metric. NIST’s TEVV guidance treats evaluation methods themselves as

objects of measurement, which reinforces the need to characterize the strengths and limitations of the metric rather than assuming its outputs are self-interpreting.

### **Thresholds as risk policy**

A threshold is not a mathematical constant; it is a policy choice. You set it by asking what kind of error you are willing to pay for at this checkpoint. That choice should differ across environments.

In CI smoke tests, you may want a loose advisory threshold that catches only obvious regressions and keeps developer latency low. In staging, you may tighten the gate and inspect more boundary cases. In a release certification run, you may use a conservative threshold and require manual review for cases near the boundary. The same metric can serve all three roles if you attach different actions to different score bands.

A practical rule is to define at least three regions:

- a clear pass region, where the score is far enough from the boundary that the decision is stable;
- a review band, where the score is ambiguous and human inspection is required;
- a fail region, where the score is sufficiently poor that the automated decision is justified.

That structure is better than a binary cliff because most judge-based metrics are not perfectly stable around a single point. You should also expect some overlap between adjacent cases: outputs with nearly identical scores may differ in risk relevance. When the cost of false negatives is high, widen the review band and route uncertain cases to human judgment instead of forcing a brittle automatic pass/fail. This is the same logic as selective use of a diagnostic instrument: when confidence is low, escalate rather than average the uncertainty away.

## How to calibrate a threshold

Threshold calibration is a product-risk exercise, not a dashboard styling exercise. Start with a labeled calibration set that reflects the failure modes you care about. Compare metric outputs to human-reviewed outcomes, then measure where the metric agrees, where it disagrees, and where disagreement is acceptable. If the metric is meant to gate a release, inspect the region around the tentative boundary first, because that is where the operational decision will actually be made.

A useful calibration loop is:

1. Freeze a representative sample of cases, including edge cases and known failures.
2. Run the metric and record both scores and reasons.
3. Have humans label the same cases under an explicit rubric.
4. Examine score distributions for passing and failing human labels.
5. Choose a threshold, a review band, or separate thresholds by risk tier.
6. Re-test on a holdout slice before promoting the choice to a gate.

Do not choose thresholds by looking only at aggregate accuracy. Two thresholds can produce the same overall score while behaving very differently on the cases that matter most. A threshold that minimizes average error may still be unacceptable if it misses rare but costly failures. If your product has distinct risk tiers, give them distinct policies. A finance workflow, a customer support chatbot, and an internal drafting tool should not share the same tolerance for judge uncertainty. The RMF's emphasis on risk management supports this segmented view:

the metric policy should follow the consequence of failure, not the convenience of one-size-fits-all automation.

### **Reading judge rationales without over-trusting them**

Judge rationales are useful because they expose the evaluator's local reasoning, which often reveals why a score moved. They are not reliable causal explanations. A rationale may mention a missing fact, but the true issue may have been retrieval drift. It may praise surface fluency while ignoring a subtle factual error. It may sound authoritative even when the underlying judgment is unstable across repeated runs.

Use rationales as triage hints. If the rationale repeatedly points to the same symptom, that is a strong lead. If the rationale contradicts the output, suspect evaluator confusion, prompt ambiguity, or hidden context mismatch. If the rationale is highly specific but changes across runs for the same case, do not treat it as ground truth; treat it as evidence of judge variance.

This matters especially with LLM-as-a-judge systems. They are not just scoring instruments; they are language models that can be influenced by prompt framing, rubric wording, context ordering, and examples. Recent judge-evaluation research emphasizes that these systems can be calibrated and selectively trusted, but not assumed to be perfectly stable or perfectly aligned with human judgment on every case. In practice, the rationale is most valuable when it helps you decide where to look next, not when it is used as a substitute for looking.

### **Run-to-run variation is information**

If the same test case produces slightly different scores across runs, that is not automatically a problem. It becomes a problem only when the variation is large enough to change the operational decision or when it points to an unstable evaluator. Judge-based systems can vary because of model stochasticity, context ordering, hidden system prompts, tem-

perature settings, or changes in retrieved evidence. The right response is not to demand perfect determinism from every metric, but to quantify how much variation you can tolerate.

A stable program records score distributions, not just point estimates. If a case sits near the threshold and oscillates across runs, it belongs in the review band, not the automatic gate. If a large fraction of cases cross the threshold under repeated execution, the threshold is too tight, the metric is too noisy, or the system is genuinely unstable. Those are different conclusions, and you need repeated measurement to separate them.

Run-to-run variation also helps you distinguish product change from harness noise. If a new model version shifts the median score only slightly but sharply widens the variance, the product may have become less predictable even if the average stayed similar. If the mean is stable but the reason strings differ wildly, the evaluator may be unstable even when the final score looks acceptable. Treat repeated evaluation as a measurement of both central tendency and dispersion. A trustworthy program documents both. NIST's newer evaluation guidance stresses choosing performance statistics and uncertainty descriptions that match the evaluation goal, which is exactly the right way to think about score variance in a release pipeline.

### **Boundary cases tell you more than averages**

Aggregate movement is useful, but boundary cases are where interpretation becomes operational. If you only inspect the mean score, you can miss the fact that one slice of traffic improved while another regressed sharply. You can also miss a threshold problem: a small mean improvement may hide a large increase in false negatives near the gate.

Sample cases near the decision boundary on every meaningful run. Inspect both passes and failures that cluster just above and just below the threshold. Those examples show you how fragile the decision really is.

When a score is close to the cutoff, the rationale, the retrieved evidence, and the product context all matter more than the raw number. This is also where disagreement between metrics is most valuable. If one metric passes and another fails, do not average them blindly; use the disagreement to infer which assumption each metric encodes.

A useful reading rule is:

- If multiple metrics fail coherently, suspect a real product problem.
- If one metric fails and the others pass, inspect the failing metric's rubric and scope.
- If a metric fails only near the boundary, treat it as unstable until calibrated.

That simple triage rule prevents the common mistake of letting a single composite score erase the internal structure of the evidence. DeepEval supports multiple metrics over the same test set, which is valuable precisely because different metrics can disagree in informative ways.

## **What to do with disagreement**

Disagreement is not noise by default. It can reveal scope mismatch. A faithfulness metric may fail because the answer is well written but unsupported. A helpfulness metric may pass because the answer is fluent and complete, even though the answer is unsafe in a narrow context. A retrieval metric may pass because the evidence is relevant, while the generation metric fails because the model ignored that evidence.

When metrics disagree, inspect their assumptions before you inspect the model. Ask whether they are measuring the same object at all. A refusal detector and a correctness metric can legitimately disagree. A process metric and an end-answer metric can legitimately disagree. If

you expected agreement, you may have chosen the wrong metric combination for the decision you are making.

This is also where human review becomes a calibration tool rather than an exception handler. A small rotating set of labeled cases can show you whether the disagreement is informative or merely unstable. If judges consistently disagree with human labels on one slice, revise the rubric or the threshold. If the judges disagree with each other but humans are consistent, the evaluator setup may be too loose. If humans disagree as well, the product specification may be the problem. The point is to map the disagreement to the right layer, not to force a premature numeric consensus.

### **Reading scores under changing releases**

Scores become especially easy to misread when a framework or metric implementation changes. A trend line that crosses a new version boundary can reflect true model drift, evaluator drift, or a changed default. DeepEval’s changelog has documented interface renames and capability maturation, which is a reminder that historical comparability only holds when you know what version produced the measurement. If score semantics, trace schemas, or parameter names change, annotate the run and re-baseline deliberately rather than comparing two eras as if nothing happened.

That version awareness matters even when the numerical output looks similar. A metric may retain the same name while its prompt, judge model, or default threshold shifts underneath it. When that happens, you have not preserved a measurement series; you have created a new instrument. The right habit is to record the metric version, judge configuration, prompt text, and dataset revision alongside the score. Otherwise, the most interesting question after a surprising result becomes impossible to answer: did the product change, or did the instrument change?

## **Anti-patterns that destroy interpretability**

Several habits make evaluation output look more trustworthy than it is:

- Treating a single composite score as product truth. A composite collapses distinct failure modes and hides the diagnosis.
- Reading judge reasons as if they were traces. A rationale is an interpretation, not a replay of execution.
- Changing thresholds every time a result is uncomfortable. That turns the metric into a justification machine.
- Ignoring boundary cases because the average looks fine. Averages can hide release-blocking failures.
- Removing hard examples from the suite after a regression. That destroys the evidence that would have kept the issue visible.

These are not just process mistakes; they are epistemic mistakes. They change the meaning of the numbers you collect. If you tune the suite until it always agrees with current expectations, the suite stops being a measurement system and becomes a ceremonial artifact. A trustworthy program stays informative precisely because it preserves inconvenient evidence long enough for you to interpret it correctly.

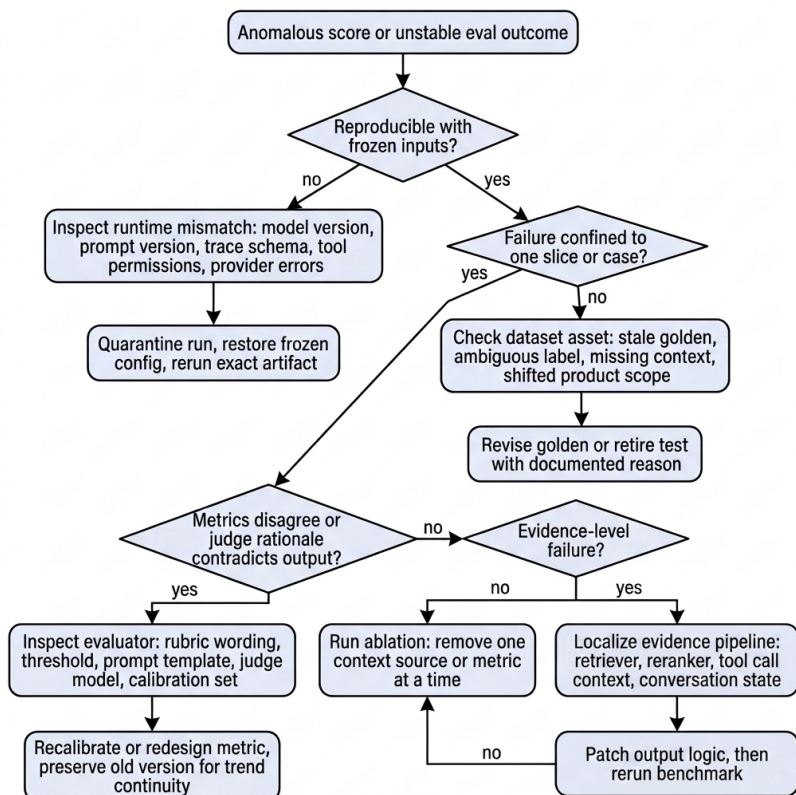
## **A disciplined reading workflow**

The most reliable habit is also the simplest. For any noteworthy result, move through the evidence in the same order every time: inspect the aggregate movement, sample cases near the threshold, compare related metrics, read the judge rationale as a hint rather than a verdict, and check the result against a small human-labeled calibration set. That workflow keeps you from overreacting to a single score and from underreacting to a meaningful regression.

If you keep that discipline, the harness earns trust for the right reason. It does not pretend to be omniscient. It produces structured, repeatable evidence, and it gives you enough information to decide when to block, when to review, and when to move forward with confidence.

## **7.2. Debugging Failure Signals Across Prompts, Data, Metrics, and System Boundaries**

Once you accept that a score is evidence rather than truth, the debugging problem becomes sharper: you are no longer asking whether the suite is “right,” but where the evidence stopped being reliable. A bad result can originate in the application, the benchmark asset, the evaluator, or the runtime around them. If you do not separate those layers, you will patch the wrong thing with high confidence. A release candidate may truly regress, or the failure may come from a stale golden, a malformed trace, a rubric that encodes the wrong expectation, or a provider-side mismatch that never touched the application logic. The incident workflow below keeps those hypotheses separate long enough for you to act on the correct one.



## Classify the symptom before you touch anything

The first mistake engineers make is to jump from a surprising score to a fix. Start instead by classifying the anomaly. The class of failure tells you which layer is most likely at fault and which diagnostics will waste your time.

A deterministic regression usually reproduces on demand with the same input, prompt, and model version. That points you toward the application path or a stable dataset mistake. An intermittent failure that appears and disappears across identical reruns is more often a

runtime, judge, or evidence-instability problem. A sudden score drift across many cases suggests a change in upstream behavior, evaluator semantics, or retrieval quality. Disagreement between related metrics tells you that the failure may be real but localized: one instrument is measuring a different property, or one of them is unstable. A judge rationale that contradicts the visible output often indicates rubric ambiguity, prompt injection into the evaluator, or an evaluator that has inferred the wrong context. Failures concentrated in a single dataset slice are a strong signal that the benchmark asset itself has drifted, or that the product's behavior changed in a way the benchmark never modeled.

You want the classification step to be explicit and repeatable. If you skip it, you will tend to interpret every anomaly as a product bug. That bias is dangerous because a broken benchmark can survive for weeks while teams repeatedly “fix” the model.

### **Freeze the exact failing artifact**

Reproduction starts with freezing the artifact that failed: the exact input, expected output, model version, prompt template, tool schema, retrieval payload, and trace captured during the run. Do not debug from a paraphrase of the case. Do not retry with a slightly different prompt. Do not use the latest model if the failure happened on an older revision. You need the frozen artifact, because evaluation systems are sensitive to small changes that are easy to miss in notebook-style debugging.

A minimal reproducibility bundle should include:

- the test case text or dataset row;
- the model identifier and provider;
- the prompt or system message used during evaluation;
- the tool permissions and available functions;

- the retrieved context or external evidence;
- the metric configuration, including threshold and rubric;
- the timestamped trace or span record.

DeepEval’s tracing model is useful here because it separates traces into spans, and component-level evaluation lets you inspect retrievers, tools, and generations independently. That makes the frozen artifact more than a log dump; it becomes a structured observation you can replay and probe. Component-level evaluation is currently single-turn only, so if the anomaly appears in a multi-turn path, you often need to isolate the relevant turn before you can localize the fault cleanly.

### **Distinguish output-level, evidence-level, and metric-level failures**

Once you can reproduce the case, ask where the mismatch occurs.

An output-level failure means the application itself produced the wrong thing. The model may have answered incorrectly, hallucinated, refused when it should not have, or taken the wrong action. If the visible behavior is clearly bad under the product rubric, the application is at least partly responsible.

An evidence-level failure means the support around the output is wrong. The retriever may have returned stale passages, the reranker may have suppressed the relevant document, the tool call may have omitted required arguments, or the conversation state may have leaked from a prior turn. In that case the final answer may look wrong because the evidence was corrupted upstream. Debugging the terminal output alone will not help.

A metric-level failure means the application output is acceptable, but the evaluator misread it. The golden may be too narrow, the rubric may be ambiguous, the threshold may be miscalibrated, or the judge model

may overweight stylistic cues over semantic correctness. You should treat this as a harness defect, not a model defect.

The fastest way to separate those classes is to ask whether the product still looks wrong when you inspect the raw supporting evidence. If the answer remains obviously wrong, the application or evidence layer is involved. If the output looks acceptable to a human but the metric still fails, inspect the evaluator before you touch the application.

### **Use nearby passing cases as your control group**

A single failure case rarely tells you enough. Compare it with nearby passing cases that differ in only one plausible factor. That gives you a control group without requiring a full experiment design.

If the failing case and a passing neighbor share the same prompt but diverge in retrieved documents, look at retrieval quality first. If they share the same retrieval payload but differ in generated responses, the issue is likely in the prompt or model path. If they produce similar outputs but only one fails, the evaluator is probably sensitive to wording that the product does not care about.

A disciplined comparison should answer three questions:

1. What changed between the passing and failing cases?
2. Which layer first observed that change?
3. Does the change matter to the product or only to the metric?

This is where differential diagnosis is stronger than a single root-cause guess. Two outputs may both be imperfect, but if only one crosses the metric boundary, the threshold may be overfitting the benchmark phrasing. Conversely, two outputs may look equally fluent while only one is grounded in evidence. In that case the metric may be correctly exposing a product issue that a casual spot check would miss.

## **Run ablations before you rewrite the benchmark**

Ablations tell you whether the failure is stable across the measurement stack. Remove one retrieval source, one tool call, or one metric at a time and see whether the anomaly persists. If the score recovers when you remove a single metric, the metric may be unstable. If the output changes only when you remove a context source, the issue likely lies in retrieval or grounding. If the result depends on one specific tool permission, the boundary condition is probably in the execution environment.

Ablation is especially useful when several metrics disagree. Coherent failure across multiple metrics is a stronger sign of a real problem than disagreement among them. If a faithfulness metric fails while a helpfulness metric passes, the answer may be fluent but unsupported. If a safety metric fails while a correctness metric passes, the content may be factually strong but operationally unacceptable. Do not average such results into one composite and then declare the ambiguity resolved. Keep the instruments separate until you understand what each one is measuring.

If multiple metrics oscillate on repeated runs of the same case, the judge model or the runtime is probably unstable. If only one metric oscillates, the rubric may be too narrow or the threshold too close to the decision boundary. Preserve the raw evidence from every run so you can compare the exact failure mode rather than a later paraphrase.

## **Locate prompt and template defects first when the pattern is broad**

When many cases drift at once, the prompt or template is often the first place to look. A small wording change can alter the model's behavior across a whole suite without changing any application code. The same is true for hidden system prompts, instruction ordering, or tool descriptions.

Prompt defects usually show up as broad pattern shifts:

- answers become shorter or more evasive;
- the model starts citing the wrong source type;
- tool calls disappear or appear in the wrong order;
- the model adopts a tone or format that the rubric penalizes;
- scores shift across many tests at once with no retrieval or data change.

You should treat prompt diffs as code diffs. Review them with the same discipline you would apply to a production API change. If a prompt edit intended to improve helpfulness also reduces faithfulness, that may be acceptable only if the metric policy and product risk justify it. The evaluator should reveal the trade-off, not hide it.

### **Inspect retrieval and tool boundaries separately from generation**

When the failure smells like grounding, focus on the evidence pipeline. Retrievers, chunkers, rerankers, tool argument builders, and state managers can each introduce failures that the generation layer then amplifies.

Retrieval regressions often appear as score drops on cases that require narrow evidence. If the retrieved context is missing a critical passage, the generator may still produce a plausible but unsupported answer, and a faithfulness metric will correctly flag it. The remediation is not to relax the metric immediately. First determine whether the retriever changed, whether the corpus changed, or whether the query formulation changed. A stale embedding index, a chunking bug, or a reranker update can all create the same surface symptom.

Tool-boundary failures are similar. If an agent passes malformed arguments or ignores tool output, the end answer may fail for reasons unrelated to the language model's reasoning quality. Inspect the tool trace, not just the final text. If the tool itself returns an error, separate the provider fault from the agent fault. You want to know whether the agent chose badly, the tool answered badly, or the infrastructure prevented the call from completing.

Conversation-state leakage deserves special attention in multi-turn systems. An answer can look locally wrong because it inherited assumptions from an earlier turn. That is not the same as a standalone generation defect. The fix may be to reset state boundaries, not to improve the answer rubric.

### **Decide when the benchmark is actually wrong**

Sometimes the model is fine and the benchmark is wrong. You need criteria for making that claim, because it is easy to reach for it when a difficult test fails.

A benchmark asset is likely flawed when one of the following is true:

- the golden expects behavior the product no longer promises;
- the label is ambiguous under the current product spec;
- the case cannot be satisfied without contradictory instructions;
- the expected output depends on stale external facts;
- the same case is interpreted differently by stable human reviewers;
- the metric cannot distinguish a meaningful failure from a harmless variation.

When that happens, do not quietly delete the test. Retire it with a documented reason, or revise it and keep the previous version in the history

trail. A hard test that exposed a real product limit is valuable even if it is no longer a valid release gate. The right move is to preserve the evidence and annotate why its semantics changed.

DeepEval datasets are most useful when they mix diverse real-world inputs, edge cases, and focused quantitative cases. That mix helps you tell the difference between a true product regression and a benchmark artifact. If all your tests are narrow and synthetic, you will confuse specification drift with model drift. If all your tests are broad and fuzzy, you will lose the signal needed for debugging.

### **Treat judge prompts and rubrics as code, not comments**

Evaluator ambiguity often looks like model instability until you inspect the rubric text. An LLM judge is highly sensitive to instruction wording, example ordering, and hidden priority conflicts. If the rubric says to prefer correctness but the examples reward style, you have built a conflict into the evaluator. If the prompt asks for a binary answer but the cases require graded judgment, you have compressed away too much information.

A good rubric should encode one decision at a time. If it blends factuality, completeness, tone, and compliance into one score, you will struggle to interpret failures. Split metrics when the underlying product risks differ. You may want separate measurements for factual grounding, policy compliance, and conversational quality so that a regression in one dimension does not mask an improvement in another.

If the judge rationale disagrees with the score, inspect whether the prompt gave the model contradictory cues. If multiple judges disagree on the same case, calibrate them against a human-labeled set before drawing conclusions about the application. Judge disagreement is not a nuisance to hide; it is information about the measurement system's variance. Recent work on LLM-as-a-judge systems emphasizes calibration and selective trust precisely because judge outputs should

be treated as measurements with known limitations, not as absolute truth.

### **Quarantine runtime mismatches before you attribute blame**

Not every failure is semantic. A surprising number of eval incidents come from the surrounding environment: changed dependency versions, altered trace schemas, tool permission drift, rate limits, provider timeouts, or serialization mismatches. These issues can look like model regressions because they contaminate the observable output.

If the same test case fails only in one environment, suspect a runtime mismatch before you blame the prompt. Check whether the model version is the same, whether the tokenizer or SDK version changed, whether the trace fields match the evaluator's expectation, and whether the same tool permissions are available. A provider outage or partial timeout can produce truncated outputs that fail a content metric even when the application logic is intact.

When you find an environment problem, quarantine the run and re-run the exact frozen artifact after restoring the environment. That preserves interpretability. If you patch the environment and the benchmark at the same time, you lose the ability to say which change fixed the failure. DeepEval's own evolution has included deprecations and interface renames, so version-annotated runs are essential whenever trace fields or test-case types may shift across releases.

### **Use a remediation rule that matches the layer**

Once you localize the failure, repair the layer that actually caused it.

Patch the application when the product behavior is wrong. That includes prompt changes, retrieval fixes, tool handling, and state management. Keep the benchmark unchanged so you can verify that the fix addresses the same case that failed.

Patch the harness when the evaluator is wrong. That includes ambiguous goldens, poorly chosen thresholds, misleading judge instructions, or a metric that measures the wrong property. Preserve the old metric version if you need historical continuity, and note the semantic change explicitly.

Patch both when the eval uncovered a product weakness and a measurement weakness at the same time. That happens often. A brittle rubric may overreact to a real but low-risk variation, while a product bug exposes a high-risk failure mode that the rubric only partially captured. Fixing both layers is usually the right answer, but you should still keep the original evidence so you can verify the causal chain.

The operational trade-off is speed versus rigor. In a live incident, you may need a quick quarantine to protect the release pipeline. After that, spend the time to understand whether the test should remain blocking, become advisory, or be redesigned. A suite that can tell you the difference between those options is one that still deserves trust.

### **Keep the harness intact while you debug it**

The hardest discipline is resisting the urge to make the suite quiet. Deleting hard goldens, relaxing thresholds, or overfitting prompts to the benchmark phrasing can make the dashboard look healthy while reducing its value as evidence. That is the opposite of what a trustworthy evaluation program needs.

Use the same benchmark asset to debug both the model and the metric, but never rewrite the asset merely to make the failure disappear. If the case is still relevant to the product, preserve it. If it is no longer relevant, retire it with traceable metadata. If the metric is too brittle, redesign it and keep the old version around for comparison. If the runtime changed, annotate the break in comparability instead of smoothing it over.

A disciplined incident workflow leaves you with two things: a repaired system and a measurement record you can still trust. When those are separated cleanly, bad results become actionable instead of confusing, and unstable results become diagnosable instead of mysterious.

### **7.3. Adapting the Harness to DeepEval Capability Milestones and Interface Shifts**

Evaluation harnesses age in a peculiar way: they do not fail when they become unusable, they fail when they remain usable in the wrong structure. A suite that once captured the product accurately can become misleading after the framework exposes richer concepts such as multi-turn conversations, component spans, simulation loops, or agent traces. At that point, the problem is no longer syntax. The deeper issue is that yesterday's benchmark architecture encodes yesterday's assumptions about what the system is and where failures live. DeepEval's 3.x-era evolution makes that distinction explicit, and a trustworthy program has to respond by changing its structure, not merely its imports. The official changelog records deprecations and renamed interfaces, including the move from `LLMTestCaseParams` to `SingleTurnParams` and from `TurnParams` to `MultiTurnParams`, which is exactly the kind of signal that should trigger architectural review rather than mechanical find-and-replace edits.

#### **Start from the capability, not the API**

A framework milestone matters when it changes what you can express cleanly. If the framework now distinguishes single-turn from multi-turn evaluation, supports traces and spans more directly, or makes component-level inspection first-class, then the benchmark should usually change shape. A single-turn template that was adequate for a question answering system may be structurally wrong for a conversa-

tional assistant whose important failures accumulate over turns. Likewise, an end-to-end score can hide a tool-selection bug, a retrieval defect, or a state leak that only becomes visible when you score the component that actually failed.

The practical consequence is that you should classify a framework change by the kind of evidence it lets you collect. Three categories are useful:

- *API-only change*: the way you call the library changes, but the underlying measurement object is the same.
- *Semantic change*: the score now means something different because the evaluator, trace model, or data shape changed.
- *Architecture change*: the benchmark itself should be reorganized because the old structure no longer maps cleanly to the product.

An API-only change can often be isolated behind wrappers. A semantic change requires recalibration and re-baselining. An architecture change requires you to rethink test-case design, trace capture, and metric composition together. If you treat all three as the same migration task, you will either overreact to a harmless signature rename or underreact to a measurement break that invalidates trend lines.

### **Use the model of the system, not the shape of the old suite**

Single-turn and multi-turn programs expose a common trap: teams keep forcing behavior into the evaluation form they already know. That is convenient, but it obscures the failure mode. A multi-turn assistant that forgets a user constraint after turn three is not just a slightly harder single-turn case. It is a different object of measurement. The relevant question is not whether the last answer looked acceptable, but whether

the conversation maintained state, respected prior instructions, and recovered from earlier ambiguity.

DeepEval’s current distinction between single-turn and multi-turn evaluation helps you keep those objects apart. Component-level evaluation is also an architectural boundary: it is currently limited to single-turn evals, so if you try to express a multi-turn system entirely through component metrics, you will miss the stateful behavior that actually matters. That limitation is not a bug in the framework; it is a clue that the harness should represent the system differently, with separate datasets and different evaluation scopes for different failure modes.

For a single-turn workload, you can often localize failure by examining one input, one output, and one context bundle. For a conversational workload, you need a sequence of turns, an explicit memory boundary, and a decision about which turn or span is being scored. For an agentic workload, you may also need tool traces, action sequences, and intermediate observations. Each step increases the dimensionality of the evidence, and the harness should reflect that rather than flattening everything into terminal text.

### Concrete migration example

When you adapt an older benchmark to newer DeepEval interfaces, start with a narrow wrapper that preserves your historical data shape while you separate the semantic change from the API change.

```
from deepeval.test_case import SingleTurnTestCase
from deepeval.test_case import MultiTurnTestCase
from deepeval.test_case import SingleTurnParams
from deepeval.test_case import MultiTurnParams

def make_single_turn(inp, out, exp):
    return SingleTurnTestCase(
        input=inp,
        actual_output=out,
        expected_output=exp,
    )
```

```
def make_multi_turn(turns, exp):  
    return MultiTurnTestCase(  
        turns=turns,  
        expected_output=exp,  
    )
```

That wrapper is not just a compatibility shim. It forces you to decide which cases are truly single-turn and which are conversational by nature. If a test case requires prior context to be meaningful, you should not keep pretending that one input-output pair is enough. The wrapper gives you a migration seam: you can keep the old benchmark runnable while you redesign the evaluation model around the real interaction pattern.

### **Treat traces and spans as architectural primitives**

The more capable the framework becomes at tracing, the less defensible it is to infer failure from only the terminal answer. A trace is an application-level envelope; a span is the unit you can evaluate or inspect inside that envelope. When tracing and observability mature, they change benchmark design because they let you attach evidence to a subproblem rather than to the whole session. That matters when the product contains retrieval, tool use, and generation as distinct steps. If retrieval is wrong, the generator may still sound plausible. If the tool call is malformed, the answer may still look polished. If state leakage corrupts the conversation, the last turn can be a symptom rather than a cause.

The correct response is to place metrics at the level where the failure originates. A retrieval score should not be asked to judge downstream prose style. A generation metric should not be asked to explain a provider timeout. A tool-use metric should not be forced to stand in for application correctness. The more explicitly you map spans to responsibilities, the easier it becomes to preserve evidence through future framework changes.

This is also where observability maturity changes the benchmark itself. If trace capture becomes richer, you may want to split one coarse test into several smaller checks so that each component owns a different contract. The goal is not more metrics for their own sake. The goal is to prevent a single composite score from hiding a defect in the one layer that actually caused the failure. Official DeepEval tracing emphasizes spans as flexible user-defined scopes, which supports this more granular style of evaluation.

### **Simulation and synthetic generation change the benchmark class**

Simulation-driven test generation is not just a convenient way to produce more examples. It changes the benchmark's role. A manually curated golden set tells you whether known cases still behave. A simulation loop lets you probe how the system behaves under structured variation, adversarial perturbation, or sequential interaction. That expands the surface area you can measure, but it also introduces a new trust obligation: you must know which part of the suite came from production evidence and which part came from generated scenarios.

Keep those sources separate. A simulation suite is ideal for coverage discovery, stress testing, and scenario shaping. A human-curated suite is better for release gates and historical comparison. If you mix them silently, you will confuse exploratory drift with regression drift. The stronger architectural move is to maintain multiple suites with different purposes:

- a stable core for release comparison;
- a rotating exploratory set for newly discovered behaviors;
- a simulation-derived set for adversarial or long-horizon scenarios;
- a migration set for validating new framework semantics.

That separation prevents a common failure mode in fast-moving teams: a flood of synthetic tests looks like progress, but the program loses continuity because no one can tell whether a score change reflects a better model, a different generator, or a changed rubric. A trustworthy harness needs all three kinds of data, but it needs them in different lanes.

### **Classify migration by how it affects comparability**

Every framework shift should be tagged by its effect on historical comparison. That classification tells you how much distrust to assign to the old trend line.

An *API-only migration* changes calls, names, or object construction. If the semantics are stable, you can usually keep historical trends intact after validating that the outputs match on a calibration slice.

A *semantic migration* changes what the score means. For example, a renamed parameter may be harmless, but a different default threshold, a new judge prompt, or a redefined trace boundary can invalidate direct score comparisons. In that case, the old and new runs may both be correct within their own semantics while remaining non-comparable to each other.

An *architecture migration* changes the benchmark structure itself. Moving from single-turn to multi-turn evaluation, or from end-to-end scoring to component-level scoring, is usually in this class. You may still preserve some aggregate continuity, but only if you explicitly define the bridge and annotate the break.

That taxonomy helps you avoid two opposite mistakes. One mistake is to treat any framework upgrade as a reason to reset all history, which discards useful signal. The other is to pretend that every new score belongs on the same line as the old one, which produces a false sense of continuity. You want continuity only when the measurement object really stayed the same.

## **Preserve a versioned evaluation record**

A robust migration process depends on metadata that survives long after the code has changed. You should annotate every run with the framework version, judge model version, dataset revision, prompt revision, and any configuration that affects evaluator behavior. If the framework renames a type or changes the recommended interface, record both the old and new semantics in the migration log.

That practice is not bureaucratic overhead. It is what lets you answer the central question after a surprising result: did the product change, or did the instrument change? Without versioned evidence, you cannot tell. A changelog is useful only if it is paired with a local record of what your benchmark meant at the time it ran. DeepEval’s published deprecations make this especially important because interface shifts are expected in a maturing 3.x line, and a benchmark that omits those assumptions will become hard to interpret very quickly.

A practical record should include:

- package version and commit or release tag;
- metric class names and thresholds;
- prompt text used by the judge;
- dataset identifiers and snapshot dates;
- trace schema or span naming conventions;
- notes on any deprecated or renamed interfaces.

If you standardize that metadata early, later migrations become auditable rather than anecdotal.

## **Use calibration slices before full promotion**

When you adopt a new capability milestone, do not promote it directly into release gating. Run it on a calibration slice first. The slice should include stable cases, known edge cases, and a few examples that were historically ambiguous. You want to learn three things before you trust the new structure:

1. whether the new interface reproduces old results where it should;
2. whether the new structure surfaces failures the old one hid;
3. whether the evaluation cost, latency, or complexity is acceptable.

A calibration slice gives you a controlled overlap period. That overlap is especially valuable when the framework adds richer tracing or more explicit conversational modeling. You can compare the old and new structures side by side and decide whether the new model improves interpretability enough to justify the migration. OpenTelemetry's migration guidance for semantic conventions provides a useful precedent here: do not silently change the schema in place. Prefer opt-in stability controls, and when feasible, support a transition period where both old and new forms can be emitted or interpreted. The same discipline applies to evaluation programs.

### **Keep telemetry changes separate from evaluation changes**

As tracing matures, it becomes tempting to treat telemetry schema updates and evaluation changes as one project. Resist that impulse. Telemetry describes how you observe the system; evaluation describes how you judge it. If both shift at once, you will not know whether a score change came from the application, the trace boundary, or the metric logic.

GenAI telemetry semantic conventions are still in development, which means you should expect schema churn and isolate it from outcome comparisons. Keep a small adapter layer between emitted traces and

your evaluator so that schema changes do not force benchmark redesign. If you need to update the trace representation, do it first on an observability branch, verify that the evaluator still sees the same semantic events, and only then compare model behavior. Treat a telemetry migration the way you would treat a database schema migration: you want compatibility windows, not silent breakage.

This separation becomes more important as the harness gains richer component support. A more detailed trace is valuable only if it preserves meaning across versions. If the span boundaries shift without notice, a historical regression may be an artifact of the schema, not the application. That is why capability growth should be accompanied by explicit metadata, not just upgraded decorators or new dashboard fields.

### **Migration strategy by risk level**

The right migration plan depends on how much the old benchmark matters to product decisions. If the suite is advisory, you can tolerate a short period of dual reporting and manual reconciliation. If it is a release gate, you need stricter overlap, explicit baselining, and sign-off on any semantic break. If it supports high-risk behavior, such as safety, compliance, or agentic side effects, the migration should preserve conservative gating until the new structure has been validated on a representative slice.

A useful operating pattern is:

1. freeze the old benchmark as a reference point;
2. implement the new capability or interface behind a migration flag;
3. run both structures on the calibration slice;
4. compare score distributions, disagreement patterns, and run-

times;

5. annotate any non-comparable cases;
6. promote only after the new structure has demonstrated stable behavior.

This approach costs time, but it prevents the much larger cost of losing interpretability. Benchmark architecture is part of your measurement infrastructure. If you change it casually, you do not merely risk a broken test; you risk a false history.

### **Avoid the most common migration anti-patterns**

Several habits make capability shifts look easier than they are:

- *Treating a version bump as a no-op.* If the framework now exposes new abstractions, the benchmark may need a new shape.
- *Collapsing old and new results into one chart.* Without annotation, you erase the meaning of the break.
- *Mixing multi-turn and single-turn assumptions.* That produces misleading averages and hides stateful failures.
- *Ignoring trace schema drift.* You may think you are comparing behavior when you are really comparing representations.
- *Promoting simulation data into the release gate without separation.* Coverage improves, but trust declines because the gate no longer reflects the same evidence class.

The unifying mistake is to assume that a richer framework automatically makes a better benchmark. It does not. It gives you the ability to model the system more faithfully, and it asks you to accept the governance work that comes with that fidelity.

### **Change the structure when the system changes**

A trustworthy DeepEval program does not preserve history by freezing itself. It preserves history by making every structural change legible. When a new milestone clarifies single-turn and multi-turn boundaries, use that distinction. When tracing becomes more expressive, score the component that actually failed. When simulation becomes practical, keep generated scenarios in a distinct lane from human goldens. When interfaces shift, annotate the shift and decide whether the change is API-only, semantic, or architectural.

That discipline keeps the harness aligned with the product as both evolve. It also prevents the most expensive kind of benchmarking error: believing that stable numbers mean stable meaning.

## **7.4. Operating an Evaluation Program Engineers Continue to Trust**

A harness earns trust slowly and loses it slowly. The damaging failures are rarely dramatic; they come from quiet drift. Goldens age out of relevance, thresholds move without explanation, new product surfaces appear without coverage, and one noisy metric starts steering release decisions because no one revisits the measurement policy. The right operating model treats evaluation as a living system with explicit governance, not as a pile of tests that happen to run in CI. Trust holds when offline evaluation, regression testing, and production monitoring feed one another in a closed loop, and when the organization knows which part of the loop owns which decision. That is the difference between a suite that merely runs and a suite that still means something six months later.

### **Start with stable assets and explicit ownership**

You cannot govern what you do not name. Separate the assets that must remain stable enough for historical comparison from the assets that should rotate as the product learns. At minimum, keep a small core of goldens, a human-reviewed calibration set, metric definitions, threshold policy, and environment annotations. Around that core, allow exploratory suites, migration suites, and production-derived samples to evolve more quickly. That separation gives each test a job instead of forcing every test to represent the entire product.

The practical consequence is that a test owner needs to know whether a case is meant to block releases, diagnose drift, or expand coverage. If a test can do all three, it will probably do none of them well. Release gates require stable semantics and disciplined thresholds. Exploratory suites can be noisier if they surface new failure modes. Migration suites are temporary bridges used to compare old and new benchmark structures. If you mix those roles, you will lose the ability to interpret trends.

Ownership matters just as much. Assign who can add a golden, who can retire one, who can change a threshold, and who can approve a metric rewrite. Without ownership, benchmark changes accumulate as invisible policy shifts. With ownership, every change has an accountable reason and a recorded effect on the trend line. NIST's AI RMF frames this as governance and documented risk management: define reasonable risk tolerance, then make the measurement process explicit enough that decisions are auditable.

### **Treat datasets as versioned evidence, not static fixtures**

A dataset in a trustworthy evaluation program is not a frozen file. It is a versioned body of evidence. You should be able to answer where each case came from, what product behavior it represents, why it remains in the suite, and when it last matched the current specification. That means retaining provenance for curated examples, historical traces, and synthetic additions, while also documenting when a case is no

longer representative.

```
from deepeval.dataset import EvaluationDataset
from deepeval.test_case import LLMTestCase

goldens = [
    LLMTestCase(
        input="Explain the refund policy.",
        expected_output="Refunds are available within 30 days.",
    ),
    LLMTestCase(
        input="Can I cancel after shipping?",
        expected_output="Cancellations are not allowed
after shipping.",
    ),
]

dataset = EvaluationDataset(goldens=goldens)
```

That object should not be treated as an immutable truth source. Instead, treat it as a calibrated sample of the product’s risk surface. Feed real failures back into it. Keep cases that represent current product commitments. Retire cases whose assumptions no longer hold, but record why they were retired so historical runs remain explainable. DeepEval’s dataset model supports this style of curation, and the broader chapter’s debugging discipline depends on it: if a case fails, you need to know whether the model regressed or the benchmark aged out. DeepEval also distinguishes single-turn and multi-turn goldens, which means you should not mix those structures inside one evaluation dataset. That boundary matters for both correctness and comparability.

### **Make thresholds a risk policy**

Thresholds only look numeric. In practice, they are policy statements about how much failure you will tolerate at a given stage of the workflow. The same score can justify different actions in different contexts. A borderline result may be acceptable as an advisory signal in a developer loop, unacceptable in a staging gate, and grounds for human

review in a release certification run. You want that policy to reflect product risk, not the aesthetic preference of whoever last edited the dashboard.

A disciplined threshold policy usually has at least three modes:

- Advisory: record the signal, but do not block the pipeline.
- Review: route borderline cases to a human calibration set.
- Blocking: stop release when the metric falls outside the acceptable region.

Risk tiers should control which mode applies. Safety-critical or compliance-sensitive behaviors justify stricter gating and narrower review bands. Low-risk internal tooling can tolerate more false positives if that preserves developer throughput. The point is not to maximize a single score; the point is to choose the right mistake to avoid at the right stage. That is why thresholding should be recalibrated against human-reviewed examples rather than inherited blindly from earlier examples or copied from another team. NIST's TEVV work emphasizes that the evaluation method itself has limits, so you need a threshold policy that acknowledges uncertainty instead of hiding it.

### **Use custom metrics when the built-in rubric is not your rubric**

Built-in metrics are useful until they stop matching the actual failure mode. At that point, you should not force the product into the metric; you should define the metric that matches the product. Custom metrics are valuable when you need explicit control over the rubric, non-LLM scoring, or compound logic that combines several signals into one decision.

That matters for two reasons. First, engineers can inspect what the

harness is actually checking. Second, you can separate concerns that are otherwise collapsed into one opaque judge. A custom metric can encode a compliance rule, a retrieval requirement, or a deterministic structural check, while another metric measures judgment-heavy qualities such as helpfulness or reasoning quality. The result is a more interpretable program because each metric owns a single measurement job.

```
from deepeval.metrics import BaseMetric

class CitationPresenceMetric(BaseMetric):
    def __init__(self, threshold=1.0):
        self.threshold = threshold

    def measure(self, test_case):
        output = test_case.actual_output or ""
        self.score = 1.0 if "[1]" in output else 0.0
        self.reason = "Found citation marker." \
            if self.score else "No citation marker."
        self.success = self.score >= self.threshold
        return self
```

This is intentionally simple, but the operating principle scales. If the failure mode is structural, do not outsource it entirely to a judge model. If the failure mode is semantic, combine the custom rule with an LLM-based metric and keep both visible. That way, a composite score never hides which assumption failed. DeepEval’s official guidance on DIY metrics points in exactly this direction: use custom logic when the built-in judge is not the right instrument for the question you are asking.

### Turn evaluation into a feedback loop with production

Trust grows when offline evaluation and production monitoring inform each other. A benchmark that never sees live failures slowly becomes a museum. Production monitoring that never feeds offline suites becomes noise. The loop works when every meaningful production incident has a path back into the calibration set, and every calibration set refresh is traceable to a real product change or a newly observed risk.

You do not need to copy every production trace into the suite. That would overwhelm the core benchmark with incidental cases. Instead, sample strategically:

- add recurring user failures that the current suite misses;
- add edge cases from support, logs, or incident reports;
- add successful examples that define the intended behavior more sharply;
- keep a small rotating set for monitoring regression over time.

This feedback loop also protects against dashboard theater. If your release gates improve but production complaints rise, your harness is no longer aligned with user experience. A trustworthy program makes that mismatch visible quickly enough to revise the suite before the gap becomes institutionalized. The practical test is simple: can you point from a production alert to a benchmark case, and from a benchmark regression back to a production-relevant risk? If the answer is no, the loop is broken.

### **Calibrate judges before you trust gates**

A CI gate is only as good as the judge behind it. If you use an LLM-based evaluator, calibrate it on labeled examples before you let it decide anything important. That means checking where it agrees with humans, where it diverges, and whether the divergence aligns with product risk or with accidental rubric sensitivity.

Judge calibration is not a one-time ceremony. It is a maintenance task. As the prompt changes, as the product changes, or as the model stack changes, rerun the calibration slice and compare the old and new distributions. If the judge becomes stricter or looser, decide whether that shift is acceptable for the current risk tier. If it is not, change the rubric or the threshold rather than pretending the change did not happen.

This is the same discipline you would apply to any measurement instrument: record its drift, quantify its disagreement with human labels, and preserve a path for human override when the operational cost of an error is high.

### **Maintain comparison continuity across metric changes**

Metric changes are inevitable. The mistake is to change them without preserving interpretability. If you revise a rubric, rename a parameter, or replace a judge model, decide whether you are making an API-only change, a semantic change, or a benchmark-architecture change. That classification tells you how to handle the historical series.

API-only changes can often be wrapped without breaking continuity. Semantic changes require re-baselining or at least an overlap period where old and new metrics run side by side. Architecture changes, such as moving from terminal-output scoring to component-level scoring or from single-turn to multi-turn coverage, deserve a new baseline and explicit annotation in the trend line. A graph that mixes incomparable metrics without labeling the break is not a trend line; it is a story told with numbers.

If you need to preserve both old and new semantics, keep the old metric as a legacy signal for trend continuity and promote the new metric separately. That lets you compare within each era without pretending the eras are identical. DeepEval’s documented renames and deprecations make this especially important: when interface shapes shift, you should update the benchmark record alongside the code so future readers know what the score meant at the time it was collected.

### **Document the operating model, not just the code**

A trustworthy evaluation program is governed by documents as much as by code. Engineers should be able to find the answer to a small set of policy questions:

- What does a pass mean at each risk tier?
- Who can change a threshold?
- What evidence is required to retire a test?
- When does human review override automation?
- Which metric version governs the current release gate?

Keep those answers in a living evaluation changelog. Record dataset revisions, metric revisions, judge prompt changes, model stack changes, and environment changes. If a run becomes surprising later, the changelog should let you reconstruct the conditions that produced it. That record is also what makes a benchmark useful during organization growth: new engineers can understand why a test exists, why a threshold is strict, and why one suite blocks releases while another remains advisory.

This is also where delegated ownership becomes important. Centralized governance gives you consistency, but local team autonomy gives you speed and domain knowledge. The compromise is to let teams propose tests and threshold revisions, while a smaller governing group approves changes that affect shared release policy or historical comparability. That balance keeps the harness from becoming either a bottleneck or an ungoverned sprawl.

### **Quarantine unstable tests instead of hiding them**

Not every test belongs in the release gate forever. Some tests become unstable because the product genuinely has edge-case variability. Others become noisy because the evaluator is under-calibrated or because the environment is not yet deterministic enough. In both cases, the answer is not to delete the evidence. Quarantine the test as a non-blocking diagnostic, annotate the reason, and keep it visible in the observability loop.

Quarantine is better than deletion because it preserves learning. A noisy test may still reveal useful drift if you watch it over time. A hard test may be too unstable for blocking today but still valuable as a signal that the product has an unresolved corner case. If you remove it, the team loses a source of pressure that could later justify a targeted fix. The same logic applies to changing model stacks: if a new model version makes a class of tests noisier, that instability itself is evidence.

### **Keep the suite aligned with product risk**

The strongest trust signal is alignment between the suite and the product's actual risk profile. A high-stakes surface should have stricter gates, tighter calibration, and explicit human review paths. A low-stakes surface can afford broader advisory coverage and more exploratory cases. If the suite's strictness does not match the consequence of failure, engineers will either fight it or ignore it.

That alignment has to be revisited as the product evolves. New features, new tools, new conversational depth, and new agent behaviors all change the failure surface. A program that remains trusted does not preserve the old suite unchanged; it keeps the governing rules legible while revising the evidence base deliberately. The reward is not a static benchmark. The reward is a measurement system whose rules change slowly, whose exceptions are documented, and whose outputs remain meaningful enough to shape releases, investigations, and product decisions.

