Application-Centric AI Evals for Engineers and Technical PMs

Shreya Shankar and Hamel Husain

Summer 2025

Evaluating the complex, often subjective outputs of Large Language Models (LLMs) presents unique challenges distinct from traditional software testing or ML validation. We present a framework for application-centric LLM evaluation: the *Analyze-Measure-Improve* lifecycle.

Contents

How To Read This Book

This book is a companion to the course *AI Evals for Engineers and Product Managers* (https://maven.com/parlance-labs/evals). The course is designed for a broad audience ranging from engineers to technical product managers.

You will encounter code and math throughout the book. If you don't know how to code or are not familiar with the math, that's okay, just focus on the high-level concepts. You will gain tremendous value from understanding the process of evaluation, forming mental models, and recognizing common pitfalls.

We encourage you to engage with the examples and reflect on what makes evaluations work (or fail). Skip sections that feel out of scope from your role. When in doubt, ask questions in the course Discord. Think of this book as a flexible tool to support your learning.

Finally, this book is a rough draft that we will evolve over the next few months in response to student feedback. **We would appreciate it if you kept this book to yourself for now.** We will be working to publish an updated version of this book that is available to the public later this year.

1 Introduction

The past two years have seen rapid advances in the development and deployment of large language models (LLMs). Organizations are now embedding LLMs in critical pipelines in applications: customer service, content creation, decision support, and information extraction, among others (Bommasani et al. 2021; Zaharia et al. 2024). However, adoption is outpacing our ability to systematically evaluate LLM pipelines (Ward and Feldstein 2024).

Unlike traditional software, LLM pipelines do not produce deterministic outputs. Their responses are often subjective, context-dependent, and multifaceted. A response may be factually accurate but inappropriate (i.e., the "vibes are off"). They may sound persuasive while conveying incorrect information. These ambiguities make evaluation fundamentally different from conventional software testing and even traditional machine learning (ML).[1]

The core challenge is as follows: How do we assess whether an LLM pipeline is performing adequately? And how do we diagnose where it is failing? Our course focuses on this gap. Although significant work has been done on model development[2], prompting techniques, and pipeline architectures, rigorous evaluation has received comparatively little attention. In this reader, we will build a practical and theoretical foundation for LLM evaluation, with methods that you can immediately apply to your own projects.[3]

[1] In traditional ML, outputs are well-defined—numbers, classes, or structured predictions. Standard metrics like accuracy or F1 score are widely used. Ground truth labels provide a clear target for evaluation. In contrast, LLM output can be open-ended, unstructured, and often subjective.

[2] If you are interested in foundation model evals, check out Liang et al. (2023), a long paper describing general metrics and evals, and some domain-specific evals for math, science, and coding: Hendrycks et al. (2021); Rein et al. (2024); Jain et al. (2025).

[3] Our course is meant to be tool and model-agnostic; if a new LLM comes out tomorrow, our methodology will still be applicable.

Note to readers

This book focuses on evaluating LLM pipelines in specific applications. *We don't cover general-purpose foundation model training or benchmarking.* Our target audience is developers who integrate LLMs into real-world applications and need reliable evaluation methods. The approaches work with both commercial APIs and fine-tuned models.

1.1 What is Evaluation?

Before we dive into methods, we need a clear definition of evaluation.

Evaluation refers to systematic measurement of quality in an LLM pipeline. A good evaluation produces results that can be easily and unambiguously interpreted. Typically, this means a quantitative score, but it can also take the form of a structured qualitative summary or report.

Throughout this course, we will refer to an evaluation metric as an *eval*. A single pipeline will often use multiple evals to measure different aspects of performance. Evals can be operationalized in several ways. For example:

- *Background Monitoring* tracks evals passively over time to detect drift or degradation, without disrupting the primary workflow.
- *Guardrails* run evals in the critical path of the pipeline. If an eval fails, the pipeline can block the output, retry the operation, or fall back to a safer alternative before showing a result to the user.
- Evals can also be used to *improve pipelines*. For example, evals can label data for fine-tuning LLMs, select high-quality few-shot examples for prompts, or identify failure cases that motivate architectural changes.

Evals are essential not only to catch errors, but also to maintain user trust, ensure safety, monitor system behavior, and enable systematic improvement. Without them, teams are left guessing where failures occur and how best to fix them.

1.2 The Three Gulfs of LLM Pipeline Development

A useful framework for understanding LLM application development challenges is the "Three Gulfs" model (Figure 1), adapted from Shankar et al. (2025) and inspired by Norman (1988). This model captures the major gaps that developers must bridge when building any LLM pipeline.

Example 1 (Email Processing Pipeline)**.** *Consider a pipeline that processes incoming emails sent to an organization. The goal is to extract the sender's name, summarize the key requests, and categorize the emails.*

At first glance, the task described in Example 1 seems simple. But each stage of development exposes new challenges across the Three Gulfs.

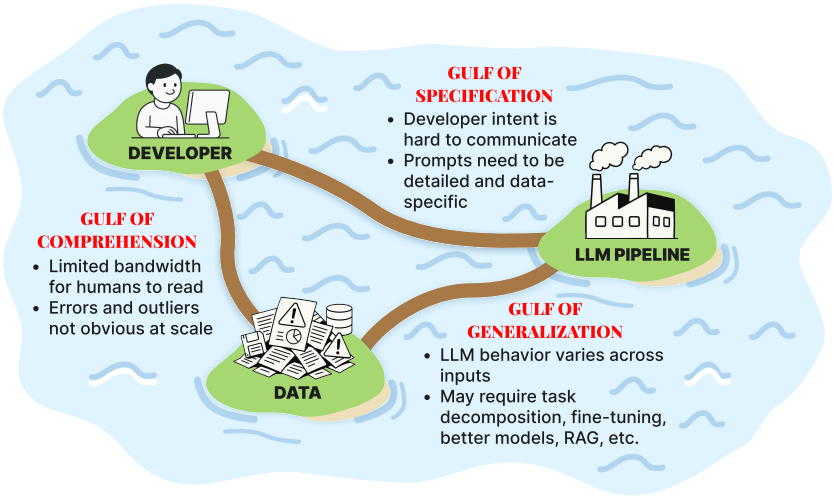


Figure 1: Three Gulfs Model of Challenges in LLM Pipeline Development. Adapted from *Steering Semantic Data Processing with DocWrangler* (Shankar et al. 2025).

The Gulf of Comprehension (Developer → Data) The first gulf separates us, the developer, from fully comprehending both our **data** and our **pipeline's behavior** on that data. Understanding the data itself—the inputs the pipeline will encounter, such as real user queries or documents like the emails in Example 1—is the starting point. At scale, it is difficult to know the characteristics of this input data distribution, detect errors within it, or identify unusual patterns. Thousands of inputs might arrive daily, with diverse formats and varying levels of clarity. We cannot realistically review each one manually. [4]

Moreover, this gulf extends to understanding the LLM pipeline's actual outputs and common failure modes when applied to this data. Just as we cannot read every input, we also cannot manually inspect every output trace generated by the pipeline to grasp all the subtle ways it might succeed or fail across the input space. So the dual challenge is: how can we understand the important properties of our input data, *and* the spectrum of our pipeline's behaviors and failures on that data, *without examining every single example?*

The Gulf of Specification (Developer → LLM Pipeline) The second gulf separates what we mean from what we actually specify in the pipeline. Our intent—the task we want the LLM to perform—is often only *loosely* captured by the prompts you write. Specifying tasks precisely in natural language is hard. Prompts must be detailed enough to remove ambiguity, and aligned with the structure and diversity of the data.

Even prompts that seem clear often leave crucial details unstated. For example, we might write:

> *Extract the sender's name and summarize the key requests in this email.*

[4] **Hamel's Note:** The gulf of comprehension represents the fact that you must know what you want so you can effectively prompt a LLM, evaluate it, and debug it. Additionally, this process is iterative; our research and experience show that people refine their requirements to adapt to the behavior of the LLM. Shankar et al. (2024d). We discuss how to systematically look at data to inform your prompts in future sections.

At first glance, this sounds specific. But important questions are left unanswered:

- Should the summary be a paragraph or a bulleted list?
- Should the sender be the display name, the full email address, or both?
- Should the summary include implicit requests, or only explicit ones?
- How concise or detailed should the summary be?

The LLM cannot infer these decisions unless we explicitly specify them.[5] Without complete instructions, the model is forced to "guess" our true intent, often resulting in inconsistent outputs. The Gulf of Specification captures this gap between what we want and what we actually communicate.[6]

[5] **Hamel's Note:** Underspecified prompts are usually a direct result of not looking at the data (ignoring the "data" island).

[6] **Shreya's Note:** In LLM development, prompt clarity surprisingly often matters as much as task complexity.

The Gulf of Generalization (Data → LLM Pipeline) The third gulf separates our data from the pipeline's generalization behavior. Even if prompts are carefully written, LLMs may behave inconsistently across different inputs.

In the email processing example, imagine an email that mentions a public figure, like *Elon Musk* or *Donald Trump,* within the body text. The model might mistakenly extract these names as the sender, even though they are unrelated to the actual email metadata.[7] This is not a prompting error. It is a generalization failure: the model applies the instructions incorrectly because it has not generalized properly across diverse data.

[7] The pipeline described in `https://www.docetl.org/showcase/ai-rfi-response-analysis` exhibits this failure mode.

The Gulf of Generalization reminds us that even when prompts are clear and well-scoped, LLMs may still exhibit biases, inconsistencies, or unexpected behaviors when encountering new or unusual inputs. Even as models get better, the Gulf of Generalization will always exist to some degree, because no model will ever be perfectly accurate on all inputs (Kalai and Vempala 2024).

1.3 Why LLM Pipeline Evaluation is Challenging

Developing effective evaluations for LLM pipelines is hard. It is a sufficiently different problem from traditional software engineering and machine learning operations (MLOps) that new approaches are required.

First, each application requires bridging the Three Gulfs anew. The specific ways in which understanding fails, prompts fall short, or generalization breaks down are unique to our task and dataset. There are no universal evaluation recipes.

Second, requirements often emerge only after interacting with early outputs. We might initially expect summaries to be written in prose but later realize that bulleted lists are easier for users to scan. Evaluation criteria must evolve alongside system development (Shankar et al. 2024d).

Third, appropriate metrics are rarely obvious at the outset. Unlike traditional software, where correctness is well-defined, LLM pipelines involve tradeoffs: factual accuracy, completeness, conciseness, style, and more. Choosing the right metrics depends on the specific goals of our application—and often requires experimentation.

Finally, there is no substitute for examining real outputs on real data. Generic benchmarks cannot capture the specific failure modes of our pipeline. For Example 1, without inspecting a diverse set of extracted senders, we would not notice that the model sometimes misidentifies public figures as senders. Systematic evaluation requires careful, hands-on analysis of representative examples.[8]

[8] **Hamel's Note:** If you are not willing to look at some data manually on a regular cadence you are wasting your time with evals. Furthermore, you are wasting your time more generally.

1.4 The LLM Evaluation Lifecycle: Bridging the Gulfs with Evaluation

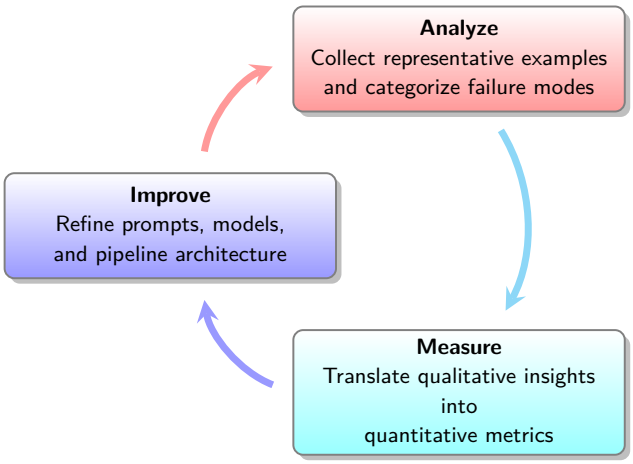


Figure 2: The Analyze–Measure–Improve evaluation lifecycle.

The Three Gulfs—Comprehension, Specification, and Generalization—highlight the core challenges inherent in developing reliable LLM pipelines. **Evaluation provides the systematic means to understand and address these challenges.** In this course, we introduce a practical, iterative approach centered around evaluation: the **Analyze-Measure-Improve** lifecycle, depicted in Figure 2. This lifecycle provides a repeatable method for using evaluation to build, validate, and refine LLM pipelines, specifically aimed at overcoming the Three Gulfs.

The lifecycle begins with **Analyze**, directly tackling the *Gulf of Comprehension*. We inspect the pipeline's behavior on representative data to *qualitatively* identify failure modes. This critical first step illuminates why the pipeline might be struggling. Failures uncovered often point clearly to ambiguous instructions (Specification issues) or inconsistent performance across inputs (Generalization issues).

The findings from Analyze guide our next actions. Failures stemming directly from poor Specification—unclear prompts or ambiguous

instructions—can often be addressed immediately in the **Improve** phase. We simply make the instructions clearer. However, other failures, particularly those suggesting inconsistent generalization or complex interactions, require deeper investigation. Understanding their true frequency, impact, and root causes demands quantitative data before effective improvements can be made.

This need for data drives the **Measure** phase. Here, we develop and deploy specific evaluators (evals) to *quantitatively* assess the failure modes identified in Analyze, especially those needing further scrutiny. Measurement provides hard numbers on failure rates and patterns. This data is crucial for prioritizing which problems to fix first and for diagnosing the underlying causes of tricky generalization failures.

Finally, the **Improve** phase leverages the findings from both previous phases to actively bridge the *Gulfs of Specification and Generalization*. We make targeted interventions. This includes direct fixes to prompts and instructions addressing Specification issues identified during Analyze. It also involves data-driven efforts—guided by the quantitative results from Measure—such as engineering better examples, refining retrieval strategies, adjusting architectures, or fine-tuning models to enhance generalization.

Cycling through Analyze, Measure, and Improve (Figure 2) creates a powerful feedback loop. It uses structured evaluation to systematically navigate the complexities posed by the Three Gulfs, leading to more reliable and effective LLM applications. We will cover each phase of the evaluation lifecycle in greater depth in the upcoming sections.

1.5 Summary

Effective evaluation is the foundation of reliable LLM applications. Without it, development reduces to guesswork. The key takeaways from this lecture are:

- Evaluation is not optional. It is essential for any serious LLM application.
- The Three Gulfs model helps categorize and diagnose different sources of failure.
- Choosing appropriate metrics requires examining our specific data and use case—not relying on generic benchmarks.
- The Analyze-Measure-Improve lifecycle provides a structured, repeatable method for building better evaluations over time.

In the next sections, we will dive deeper into evaluation metrics and systematic error analysis techniques.

About This Preview

The table of contents and this first chapter are a preview of what we teach in our course. **Students enrolled in our course get access to the full version.** The course reader is designed to give students comprehensive notes that supplements their learning. We suggest revisiting the table of contents to get a sense of the breadth of material covered. **To see student reactions to these materials (and secret discount codes), see this page** `https://bit.ly/eval-reviews`**.**

2 LLMs, Prompts, and Evaluation Basics

Note to readers

This section assumes the reader has a foundational understanding of Large Language Models (LLMs): that they are AI systems trained on vast amounts of text data to understand and generate human-like language, and that we interact with and instruct them using textual inputs known as prompts. For a glossary of specific terminology introduced and used throughout this chapter, please refer to Section 2.7.

In this section, we will first understand what LLMs are generally good at and where their limitations lie. Then, we'll explore how we interact with these models through prompting. Finally, we will introduce fundamental concepts of evaluation itself: defining what "good" means through metrics, understanding the important differences between evaluating foundation models versus specific applications, and exploring methods to gather feedback and assess results.

Readers new to this area may find some of the terminology unfamiliar or ambiguous. Rather than interrupt the narrative flow, we've placed definitions of essential terms in a glossary at the end of this chapter.

2.1 Strengths and Weaknesses of LLMs

LLMs are capable at producing fluent, coherent, and grammatically correct text. This extends to editing tasks such as simplifying sentences, adjusting tone, or rephrasing input for clarity. Their ability to make sense of information across a prompt enables effective summarization, translation, and question answering. Because they learn broad statistical patterns during training, they seemingly generalize to new tasks easily. A single prompt or a few demonstrations are often enough to guide the model, without retraining (Brown et al. 2020).

Despite their strengths, LLMs face limitations that stem from their architecture, training objective, and probabilistic nature.

Algorithmic generalization. One way to reason about the limitations of Transformers is to think about the kinds of "algorithms" they cannot do. They lack internal mechanisms for loops or recursion. This prevents reliable execution of tasks needing variable iterative steps. Examples include arbitrary-precision arithmetic or complex graph traversal. Generalization on algorithmic tasks beyond the training distribution is often poor; for example, a model trained on 3-digit addition may fail on 5-digit sums (Qian et al. 2022). Theoretical limits exist on the types of tasks Transformers can perform (Hahn 2020; Weiss et al. 2021; Zhou et al. 2024a).[9]

Moreover, LLM processing is also limited by an *effective* context win-

[9] Techniques involving "test-time computation," such as Chain-of-Thought prompting (Wei et al. 2022), scratchpads (Nye et al. 2021), or equipping models with external tools (like calculators, code interpreters, or dedicated search/retrieval systems) (Schick et al. 2023), aim to mitigate these algorithmic limitations.

dow often shorter than the advertised maximum. While models may technically accept long inputs, empirical studies show that attention quality and output reliability degrade with length (Li et al. 2024).

Reliability and Consistency. LLM outputs are probabilistic. Generation involves sampling from a distribution over possible next tokens. This introduces nondeterminism: even identical prompts can yield different outputs. Such variability is at odds with the expectations of traditional software systems. While deterministic decoding methods like greedy sampling can reduce randomness, they often degrade output quality and accuracy. In addition to nondeterminism, models lack built-in mechanisms for global consistency. They may contradict earlier statements within a single output or across turns in a conversation.

Moreover, LLMs also display significant prompt sensitivity (Sclar et al. 2024). Small changes in phrasing—rewording a question, shifting example order, or introducing irrelevant tokens—can lead to dramatically different completions. It remains an open question whether this reflects shallow pattern matching, overfitting to training distributions, or emergent properties of large-scale optimization.

Factuality. LLMs possess no internal notion of "truth." Their objective is to produce text that is statistically likely—not factually verified. As a result, they can hallucinate (Kalai and Vempala 2024). A model may confidently assert incorrect claims or fabricate details that sound plausible but are entirely false. There is no inherent mechanism to cross-check outputs against trusted knowledge sources or the external world. To achieve factual accuracy, systems must incorporate additional components such as retrieval-augmented generation (RAG) or external tools.

Key Takeaway 2.1

We should treat LLMs as powerful but imperfect components. We can leverage their strengths in language generation and understanding, but always anticipate variability, potential inaccuracies, and limitations in multi-step reasoning.

Given these strengths and weaknesses, how do we effectively interact with LLMs? The primary method is prompting.

2.2 Prompting Fundamentals

Prompting is the act of crafting the input text given to the LLM to elicit the desired output.[10] A well-structured prompt typically includes several key pieces:

1. **Role and Objective:**

[10] For an interactive tutorial on prompt engineering concepts, see the guide by Anthropic: https://github.com/anthropics/prompt-eng-interactive-tutorial, as well as a guide by OpenAI: https://cookbook.openai.com/examples/gpt4-1_prompting_guide.

- Clearly define the persona or role the LLM should adopt and its overall goal. This helps set the stage for the desired behavior.
- *Example: "You are an expert technical writer tasked with explaining complex AI concepts to a non-technical audience."*

2. **Instructions / Response Rules:**
   - This is a core component, providing clear, specific, and unambiguous directives for the task. For newer models that interpret instructions literally, it's vital to be explicit about what to do and what *not* to do.
   - Use bullet points or numbered lists for clarity, especially for multiple instructions.
   - *Example:*
     - "Summarize the following research paper abstract."
     - "The summary must be exactly three sentences long."
     - "Avoid using technical jargon above a high-school reading level."
     - "Do not include any personal opinions or interpretations."
   - For complex instruction sets, consider breaking them into sub-categories (e.g., `### Tone and Style`, `### Information to Exclude`).
3. **Context:**
   - The relevant background information, data, or text the LLM needs to perform the task. This could be a customer email, a document to summarize, a code snippet to debug, or user dialogue history.
   - *Example: "[Insert the full text of the customer email here]"*
   - When providing multiple documents or long context, clear delimiters are crucial (see point 7).
4. **Examples (Few-Shot Prompting):**
   - Provide one or more examples of desired input-output pairs. This is highly effective for guiding the model towards the correct format, style, and level of detail. Examples can also clarify nuanced instructions or demonstrate complex tool usage.
   - *Example: Showing one or two sample emails and their ideal bullet-point action items, or a sample input and the correctly formatted JSON output.*
   - Ensure that any important behavior demonstrated in your examples is also explicitly stated in your rules/instructions.
5. **Reasoning Steps (Inducing Chain-of-Thought):**

- For more complex problems, you can instruct the model to "think step by step" or outline a specific reasoning process. This technique, often called Chain-of-Thought (CoT) prompting, encourages the model to break down the problem and can lead to more accurate and well-reasoned outputs, even for models not explicitly trained for internal reasoning.
- *Example: "Before generating the summary, first identify the main hypothesis, then list the key supporting evidence, and finally explain the primary conclusion. Then, write the summary."*

6. **Output Formatting Constraints:**
   - Explicitly define the desired structure, format, or constraints for the LLM's response. This is critical for programmatic use of the output.
   - *Example: "Respond using only JSON format with the following keys: `sender_name` (string), `main_issue` (string), and `suggested_action_items` (array of strings)." Or, "Ensure your response is a single paragraph and ends with a question to the user."*
7. **Delimiters and Structure:**
   - Use clear delimiters (e.g., Markdown section headers like `### Instructions ###`, triple backticks for code/text blocks, XML tags) to separate different parts of your prompt, such as instructions, context, and examples. This helps the model understand the distinct components of your input, especially in long or complex prompts.
   - A general recommended prompt organization, especially for complex prompts or long contexts, is to place overarching instructions or role definitions at the beginning, followed by context and examples, and potentially reiterating key instructions or output format requirements at the end.

The goal of effective prompting is to bridge the **Gulf of Specification** we discussed in Section 1—making our intent as explicit and unambiguous as possible for the model. What seems obvious to us might be unclear to the LLM. E.g., is "summarize" meant to be extractive or abstractive? Should the tone be formal or informal? Precision in our prompt is key. However, finding the perfect prompt is rarely immediate. It's an iterative process. We'll write a prompt, test it on various inputs, analyze the outputs (using evaluation techniques we'll discuss in future sections), identify failure modes, and refine the prompt accordingly. [11]

An iterative refinement process hinges on having clear ways to judge whether the output is good or bad. This brings us to the concept of evaluation metrics.

[11] **Hamel's Note:** There are many tools that will write prompts for you and optimize them. It's important that you avoid these in the beginning stages of development, as writing the prompt forces you to externalize your specification and clarify your thinking. People who delegate prompt writing to a black box too aggressively struggle to fully understand their failure modes. After you have some reps with looking at your data, you can introduce these tools (but do so carefully).

2.3 Defining "Good": Types of Evaluation Metrics

Evaluation metrics provide systematic measurements of the quality of our LLM pipeline. At a high level, evals fall into two categories: reference-based and reference-free (Yan 2023).

Reference-Based Metrics. These evals compare the LLM's output against a known, ground-truth answer. We often call this known answer a "reference" or "golden" output. To use reference-based metrics, we must prepare the correct reference answers in advance for our test data. This is much like having an official answer key to grade a multiple-choice test. Reference-based metrics are often valuable during the development cycle, e.g., as unit tests. We typically use them in offline testing stages where curating ground truth is feasible, as we will explore further in Section 9.

Simple examples include:

- Check if the LLM output exactly matches the reference string (e.g., for short-answer extraction).
- Verify if the output contains specific keywords expected from the reference.

Beyond simple string comparisons, reference-based checks can become more complex. Sometimes, this involves *executing* the result and comparing the result with a reference output. For example:

- Executing generated SQL and checking if the output result or table matches a known correct result or table.[12]
- Running LLM-generated code against pre-defined unit tests and verifying the results align with expected outcomes.

[12] For LLM-generated SQL, there are often many logically-equivalent but syntactically different SQL queries.

However, complexity can also arise in the comparison step itself, even without execution. We might need a sophisticated method—an "oracle" like an LLM judge or a human—to determine if the output is semantically equivalent or acceptably close to the reference. For example, an LLM itself could check if a generated paragraph conveys the same core meaning as a reference paragraph, even with different wording.

In some cases, we might combine approaches, such as executing generated code and then using an oracle to evaluate whether the runtime behavior aligns with the intended reference behavior.

Reference-Free Metrics. Alternatively, reference-free metrics evaluate an LLM's output based on its inherent properties or whether it follows certain rules, operating without a specific "golden" answer. This approach becomes crucial when dealing with output that is subjective, creative, or

where multiple valid responses can exist. The goal is to check the desired properties of the output directly.

For example, when evaluating a text summarizer, a reference-free metric might involve checking if the summary introduces speculative claims or opinions not actually present in the original source material. For a customer service chatbot, we could assess whether it appropriately refrains from offering definitive medical or financial advice, instead guiding users toward expert consultation. When evaluating generated code, such metrics can verify if the code includes explanatory comments for non-trivial logic, or if it avoids using functions known to be deprecated. In a different domain, like marketing copy, we might use reference-free checks to ensure the text contains a clear call-to-action while also excluding specific types of overly aggressive sales phrases that are inconsistent with brand guidelines. These examples illustrate how reference-free qualities are highly application-specific.

Reference-free checks can also involve execution, but focus on *validity* rather than comparing results. For instance, we can assess if generated SQL executes without syntax errors or if generated code compiles cleanly and passes linter checks. Or, does a generated API call use valid parameter names and types according to the defined schema? Defining acceptable quality—avoiding speculation, using valid code, adhering to brand voice—depends entirely on the specific use case and requirements (Liu et al. 2024a; Shankar et al. 2024c). In production, generating real-time ground truth is often impractical, making these self-contained checks vital (a topic that we revisit in Section 9).

The distinction between reference-based and reference-free metrics highlights a broader point: the way we evaluate depends heavily on *what* we are evaluating.[13]

[13] **Hamel's Note:** Why should you care about reference-free vs based? Reference-based metrics are preferred when it is feasible to have them. You can often derive a reference-based way of evaluating a failure by isolating intermediate steps, or reverse engineering references. For example, for you can construct a reference-based dataset to test retrieval by using a LLM to generate queries for specific documents such that you get (query, document) pairs. More on this later.

2.4 Foundation Models vs. Application-Centric Evals

We often hear about LLMs that achieve impressive results on public benchmarks such as MMLU (Hendrycks et al. 2021), HELM (Liang et al. 2023), or GSM8k (Cobbe et al. 2021). It is important for us to understand how these evaluations differ from those we typically perform for our own applications.

We can think of foundation model evals (the popular benchmarks) as assessing the general capabilities and knowledge of the base LLM itself. The methodology typically involves public datasets that cover areas such as math, coding, and general purpose reasoning.[14] For engineers and product builders, these benchmarks offer a rough sense of a model's overall power. They can be useful for initial model selection; a higher benchmark score might indicate a better starting point for our application.

[14] **Shreya's Note:** Think of foundation model evals as "standardized tests." When hiring people, we rely on more than standardized test scores. The same principle applies when picking LLMs to use in our applications—we need application-specific evals.

In contrast, *application evals* represent our primary day-to-day focus. Their purpose is assessing if *our specific pipeline* performs successfully *on our specific task* using our realistic data. Foundation models undergo extensive alignment (SFT/RLHF) using provider-specific data and preferences. However, this general post-training process and its resulting model "taste" are opaque to us. There is no guarantee that it matches our application's specific requirements. So, we have to rely heavily on metrics that we design ourselves to capture our specific quality criteria. Examples include measuring helpfulness according to our users' goals, ensuring adherence to our specific output formats, or enforcing safety constraints relevant to our domain.[15]

Since application-specific evaluations often involve criteria beyond simple right/wrong answers, we need methods to systematically capture these judgments. How do we actually generate these evaluation signals?

[15] **Hamel's Note:** You should be extremely skeptical of generic metrics for your application. In most cases, they are a distraction and provide illusory benefit. The quickest smell that an evaluation has gone off the rails is to see a dashboard packed with generic metrics like "{hallucination, helpfulness, conciseness, truthfulness}score".

2.5 Eliciting Labels for Metric Computation

How do we actually generate the scores or labels needed to compute our metrics? This question is especially pertinent for reference-free metrics, where there is no golden answer key. How can we systematically judge qualities defined by our application, like "helpfulness" or "appropriateness"? The process often requires methods for eliciting structured judgment, using either human reviewers or sometimes another LLM (as we will discuss in Section 5).

The most common method of evaluations for AI applications is *Direct Grading or Scoring*. Here, an evaluator assesses a single output against a predefined rubric. This rubric might use a scale (e.g., 1-5 helpfulness) or categorical labels (e.g., Pass/Fail, Tone: Formal/Informal/Casual). Evaluators can be human annotators, domain experts, or a well-prompted "LLM-as-judge," or an LLM that has been prompted to assess outputs according to the rubric. Obtaining reliable direct grades demands extremely clear, unambiguous definitions for every possible score or label.[16] Direct grading is most useful when our primary goal is assessing the absolute quality of a single pipeline's output against our specific, predefined standards.

[16] Defining distinct, objective criteria for each point on a 1-5 scale, for example, can be surprisingly difficult. Especially in the beginning of the lifecycle of an AI application, it is not clear what each numeric point means. For this reason, simpler binary judgments (like Yes/No for "Is this summary faithful to the source?") are often easier to define consistently and can be a very effective starting point.

Other methods focus instead on *relative comparisons*. *Pairwise comparison* presents an evaluator with two outputs (A and B) generated for the same input prompt. The evaluator must then choose which output is better based on a *specific, clearly defined criterion or rubric*—for example, "Which response more directly answers the user's question?" or "Which summary is more factually consistent with the source document?" While this still requires unambiguous comparison criteria, making this relative choice between two options is frequently cognitively easier for evaluators than assigning a precise score from a multi-level scale. *Rank-*

ing extends this relative judgement idea. Evaluators order three or more outputs generated for the same input from best to worst, according to the same clearly defined quality dimension specified in the comparison rubric. Ranking provides more granular relative information than pairwise comparison, though it typically requires more evaluator effort. These relative judgment methods are particularly useful when our main goal is to compare different systems or outputs directly. We might use them for A/B testing different prompts, comparing candidate models, or selecting the best response when our pipeline generates multiple options for a single input. Foundation model providers frequently use pairwise comparisons and ranking during post-training. For us developing applications, these relative methods might be most relevant when comparing distinct pipeline versions (perhaps ones that score similarly on direct grading metrics) or if we are fine-tuning models ourselves.

Key Takeaway 2.2 Eliciting Feedback

Generating evaluation data, especially for reference-free metrics, often relies on structured judgment based on clear criteria. Common methods we use include **direct grading** against rubrics (assessing absolute quality, requires clear definitions for each level), **pairwise comparison** (A vs. B based on a criterion, assessing relative quality), and **ranking** multiple outputs (more granular relative assessment).

2.6 Summary

In this section, we built a conceptual foundation for evaluating LLM applications. We came to understand them as powerful but fallible engineering components, possessing distinct strengths and weaknesses stemming from their design and training. We saw that prompting is our primary interface for directing these components. Effective prompting requires care, precision, and iteration. We defined evaluation metrics and explored the distinction between reference-based and reference-free approaches. We contrasted the world of general foundation model benchmarks with the specific, context-dependent needs of application evaluation—the core focus of our work here. We then explored practical ways we can elicit evaluation feedback, particularly for subjective qualities, using methods like grading, pairwise comparisons, and ranking.

With this landscape mapped out, we are now prepared to explore into the practical, iterative process of building effective evaluations. The next chapter introduces the *Analyze-Measure-Improve* lifecycle.

2.7 Glossary of Terms

The following glossary defines core concepts referenced throughout this chapter. If you're encountering unfamiliar terms or want a refresher, this section offers a concise reference.

- **Large Language Model (LLM):** A model trained on massive text corpora to generate and understand human-like language.
- **Foundation Model:** A large, general-purpose model trained on diverse data. During **pre-training**, it learns to predict the next token in a sequence, allowing it to internalize grammar, facts, and language structure.
- **Token:** The smallest unit of text processed by an LLM, often a word, subword, or punctuation mark.
- **Context Window:** The sequence of previous tokens an LLM uses to generate the next one. This is bounded in size, and limits coherence over long passages.
- **Prompt:** The input text used to elicit an output from an LLM. Prompts include instructions, context, examples, and formatting constraints.
- **Evaluation / Metric:** A method for judging output quality. Metrics can be reference-based (using ground truth) or reference-free (checking properties).
- **Pre-training:** The initial training phase where the LLM learns by predicting the next token across vast text datasets. This phase imparts broad linguistic and factual knowledge.
- **Attention:** A mechanism in transformer models that computes relevance scores between tokens, allowing the model to selectively focus on different parts of the input.
- **Post-training:** The stage after pre-training that aligns the model with human intent. This includes Supervised Fine-Tuning (SFT) and Reinforcement Learning from Human Feedback (RLHF), and more recently, Direct Preference Optimization (DPO).
- **Supervised Fine-Tuning (SFT):** Additional training using labeled examples of desired prompt-response behavior to guide the model's outputs.
- **RLHF (Reinforcement Learning from Human Feedback):** A technique where human preferences are used to train a reward model, which then guides LLM fine-tuning.
- **DPO (Direct Preference Optimization):** A recent method that skips the reward model and fine-tunes directly using human preference rankings.

2.8 Exercises

1. **Prompt Engineering.**

You need to summarize customer support emails into a list of action items.

(a) Write a zero-shot prompt including instruction and formatting constraints.

(b) Extend it to a one-shot prompt with an example.

Solution 2.1

(a) Zero-shot:

```
"Summarize the following support email into bullet-point
action items.  Email:  [EMAIL TEXT]"
```

(b) One-shot:

```
"Example:  Email:  'My account locked...'; Action items:  -
Reset password - Verify email on file --- Now summarize:
[EMAIL TEXT]"
```

2. **Metric Classification.**
   For each of these metrics, state whether it is reference-based or reference-free, and justify in one sentence:

(a) Exact match accuracy against ground-truth SQL.

(b) Checking generated code compiles without errors.

(c) ROUGE score against human summary.

(d) Verifying no speculative claims in a generated summary.

Solution 2.2

(a) Reference-based—compares to known SQL. (b) Reference-free—checks validity of code, no ground truth. (c) Reference-based—measures overlap with human summary. (d) Reference-free—checks property of output, not against reference.

3. **Foundation vs. Application Evals.**
   Suppose you choose between GPT-4 and Claude for a legal-document summarizer.

(a) Name one foundation benchmark you would consult and why.

(b) Design an application-specific eval test for fidelity to client needs.

Solution 2.3

(a) MMLU—measures general reasoning across domains. (b) Collect 20 real legal memos, have attorneys rate summaries on accuracy and relevance, binary Pass/Fail.

4. **Eliciting Labels (Travel Assistant).**
   You need to evaluate whether a travel assistant's flight recommendations respect the user's budget constraint.

   (a) Write a direct-grading rubric with two labels: *Within Budget* and *Over Budget*, and clear criteria for each.

   (b) Draft a pairwise-comparison instruction for an annotator to choose which of two flight suggestion lists better adheres to the budget.

   (c) In 2–3 sentences, explain when you would prefer direct grading versus pairwise comparison for this task.

Solution 2.4

(a) Direct-Grading Rubric:
Within Budget: All suggested flights have price $\leq$ user's maxPrice.
Over Budget: At least one suggested flight exceeds the user's maxPrice.

(b) Pairwise-Comparison Instruction:

Sample Prompt

```
You are given a user query: "Find flights under $300
from JFK to LAX." You also have two lists of three
flight options (A and B). Choose which list better
respects the budget constraint and reply with "A" or
"B," followed by a one-sentence justification.
```

(c) When to Use:
Direct grading is ideal when you need an absolute adherence rate and the criterion is unambiguous. Pairwise comparison is preferable when distinguishing among borderline cases or when annotators find relative judgments easier than strict binary labels.

3 Error Analysis

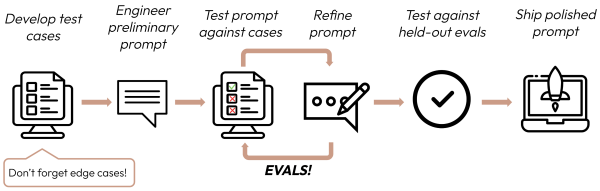


Figure 3: Anthropic's visualization of the iterative process for creating strong empirical evaluations. This cycle emphasizes developing test cases, engineering and refining prompts, and testing against both initial cases and held-out evaluations. Image source: Anthropic (Anthropic 2024).

The process of developing robust evaluations for LLM applications is inherently iterative. It involves creating test cases, assessing performance, and refining the system based on those observations. High-level guides, such as Anthropic's documentation on creating empirical evaluations for Claude (Anthropic 2024), often depict this as a cycle of developing test cases, engineering prompts, testing, and refining (Figure 3).[17] This section, and indeed our overall "Analyze-Measure-Improve" lifecycle (Figure 2), provides a detailed, step-by-step methodology for the **Analyze** portion of this iterative loop—specifically focusing on how we systematically surface failure modes by bootstrapping initial datasets and structuring our understanding of errors.

[17] See, for example, Anthropic's guide: `https://docs.anthropic.com/en/docs/build-with-claude/develop-tests`

We ground our discussion in a running example: a real estate CRM assistant.[18]

[18] This example is taken from Husain (2025)'s "Field Guide to Rapidly Improving AI Products."

Example 2 (Real Estate Agent)**.** *The assistant powers real estate agents' workflows. Given natural language queries, it can generate SQL queries to retrieve listing data, summarize trends, draft emails to clients, and read calendars. Typical user queries might include: "Find me 3-bedroom homes under $600k near downtown. Email the top 2 matches to my client. Figure out if there are showings available for this weekend."*

The pipeline is agentic.[19] An LLM call interprets the user's request and returns structured actions. Each action—querying listings, drafting an email, reading the showings calendar—invokes a downstream tool. Outputs are fed back to the LLM, which may issue further actions based on new information. The process repeats until the system validates and executes all operations.

[19] By *agentic*, we mean an architecture where an LLM decides what the next step in the pipeline should be should be.

In the rest of this section, we will detail the steps of error analysis, as depicted in Figure 4.

3.1 Create a Starting Dataset

Every error analysis starts with *traces*: the full sequence of inputs, outputs, and actions taken by the pipeline for a given input.

Ideally, we want to start with around 100 traces. This gives enough coverage to surface a wide range of failure modes and push toward *theoretical saturation*—the point at which analyzing additional traces is

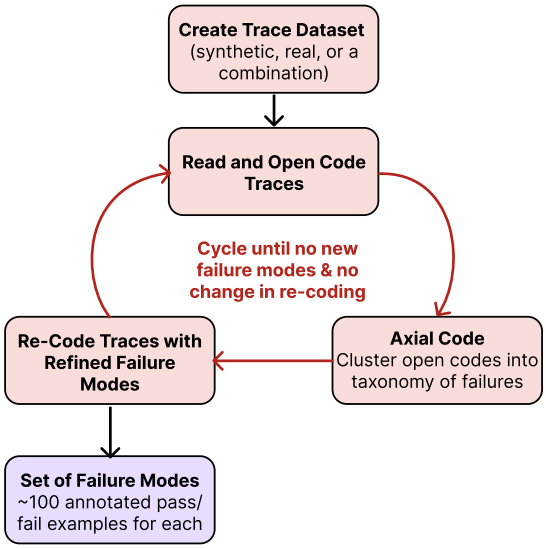


Figure 4: A detailed breakdown of the "Analyze" stage within our evaluation lifecycle (see Figure 2). These steps operationalize the initial part of error analysis, systematically moving from raw data to a structured understanding of pipeline failure modes.

unlikely to reveal suffiently new categories or types of errors, as existing concepts are well-developed and new data largely confirms them (Morse 1995). If 100 real user queries already exist, we sample them directly. If more are available, even better. Diversity matters: traces should stress different parts of the system, not just repeat the same feature path. To achieve this, we can cluster queries using embeddings or keyword heuristics and sample across clusters. Random sampling works too when the query pool is small.

In early-stage pipelines, however, real user traces are often sparse. In those cases, we generate synthetic data—but carefully. We should not simply prompt an LLM to "give us user queries." Naively generated queries tend to be generic, repetitive, and fail to capture real usage patterns. Instead, we will generate synthetic queries more systematically.[20]

First, before prompting anything, we define key *dimensions* of the query space. **Note that dimensions will be different for every application.** These dimensions help us systematically vary different aspects of a user's request.

[20] Interestingly, for many applications, the variety of fundamental user *intents* (i.e., underlying *query types*) is often surprisingly limited– -perhaps on the order of 10 to 20 core types. Understanding these core query types is highly beneficial when bootstrapping a starting dataset, especially if real user traces are sparse. This principle is observable even in complex systems like web search engines: while users can type almost anything into a search bar, a significant portion of queries can be categorized into recognizable underlying intents such as *navigational* (e.g., trying to reach a specific website like "youtube"), *informational* (e.g., seeking an answer like "what is the capital of France"), *transactional* (e.g., looking to perform an action like "buy cheap flights to London"), or *local* queries (e.g., "pizza near me"). Identifying such underlying types helps in structuring the generation or sampling of test cases.

Dimension

A **dimension** is a way to categorize different parts of a user query. Each dimension represents one axis of variation. For example, in a real estate assistant, useful dimensions might include:

- **Feature:** what task the user wants to perform (e.g., property search, scheduling)
- **Client Persona:** the type of client being served (e.g., first-time buyer, investor)
- **Scenario Type:** how clearly the user expresses their intent (e.g., well-specified, ambiguous)

Our application can have many useful dimensions. We recommend starting with at least three. **Do not choose these dimensions arbitrarily!** Instead, choose the dimensions that describe where your AI application is likely to fail.[21] For example, in the real estate example, we might have discovered through usage or qualitative data that certain personas (e.g., investors) are experiencing issues when using specific features (e.g., property search).

[21] If you do not know where your product might fail, build intuition by using the product yourself. Alternatively, recruit a few people to use the product and use those failures to motivate the dimensions.

Once we've defined our dimensions, we create structured combinations of them.

Tuple

A **tuple** is a specific combination of values—one from each dimension—that defines a particular use case. For example:

- Feature: `Property Search`
- Client Persona: `Investor`
- Scenario Type: `Ambiguous`

This tuple describes a case where an investor is searching for properties, but the request may be underspecified or vague. We use these tuples as inputs to generate realistic queries.

Here are two example queries sketched from different tuples:

- **Tuple:** (`Feature`: 'Property Search', `Persona`: 'First Time Buyer', `Scenario`: 'Specific Query')
  Query: *Find 3-bedroom homes under $600k near downtown that allow pets.*

- **Tuple:** (`Feature`: 'Property Search', `Persona`: 'Investor', `Scenario`: 'Vague Query')
  Query: *Look up showings for good properties in San Mateo County.*

Writing 20 of these tuples by hand helps us understand our problem better. If we're unsure, we can ask domain experts for example tuples. Once we have a small set of example tuples and queries, we use an LLM to scale up to 100 or more.[22] But instead of having the LLM generate full queries directly—which often leads to repetitive phrasing—we break the process into two steps. First, we sample structured tuples: feature, persona, and scenario. Then, for each tuple, we generate a natural-language query using a second prompt. This separation leads to more diverse and realistic results than simply asking an LLM to generate synthetic data for our task in one single prompt. We illustrate the full process in Figure 5.

[22] **Hamel's Note:** Don't begin generating synthetic data without a hypothesis about where your AI app will fail. At the outset, synthetic data should target these anticipated failures. After gaining familiarity, you can expand the scope beyond your hypotheses to discover new errors.

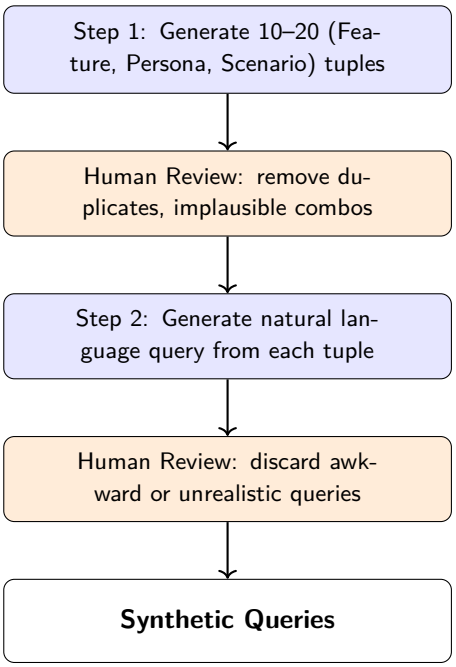


Figure 5: LLM pipeline for structured synthetic query generation, with human supervision after each stage. LLM steps shown in blue; human review steps in orange.

In the first step, we prompt the LLM like this:

Sample Prompt

```
Generate 10 random combinations of (feature, client persona,
scenario) for a real estate CRM assistant.
The dimensions are:
Feature: what task the agent wants to perform. Possible
values: property search, market analysis, scheduling, email
drafting.
Client persona: the type of client the agent is working with.
Possible values: first-time homebuyer, investor, luxury
buyer.
Scenario: how well-formed or challenging the query is.
Possible values:
```

- exact match (clearly specified and feasible),

- ambiguous request (unclear or underspecified),
- shouldn't be handled (invalid or out-of-scope).

Output each tuple in the format: (feature, client persona, scenario)
Avoid duplicates. Vary values across dimensions. The goal is to create a diverse set of queries for our assistant.

For each LLM-generated tuple, we then generate a full query in natural language. The second prompt might look like:

Sample Prompt

```
We are generating synthetic user queries for a real estate CRM
assistant. The assistant helps agents manage client requests
by searching listings, analyzing markets, drafting emails, and
reading calendars.
Given:
Feature: Scheduling Client Persona: First-time homebuyer
Scenario: Ambiguous request
Write a realistic query that an agent might enter into the
system to fulfill this client's request. The query should
reflect the client's needs and the ambiguity of the scenario.
Example:
"Find showings for affordable homes with short notice
availability."
Now generate a new query.
```

We continue sampling, generating, and filtering until we reach 100 high-quality, diverse examples. [23] When phrasing is awkward or content is off-target, we discard and regenerate. It is better to be aggressive in quality control here: downstream evaluation is entirely based on the representativeness and realism of these traces.

With a strong synthetic dataset in place, we're ready to start generating and reading traces to identify failure modes.

[23] Why 100? This is a rough heuristic that helps people get started. We will share more intuition behind this number later. The most important part is to being the flywheel of looking at data to improve your product.

3.2 Open Coding: Read and Label Traces

With a data set of queries in hand, the next step is to run the assistant on all queries and collect complete traces.

A *trace* records the entire sequence of steps taken by the pipeline: the initial user query, every LLM output, each downstream tool invocation (such as database queries, email drafts, calendar proposals), and the final user-facing results. We collect all intermediate and final steps, not just the surface output. Failures often arise inside the chain, not only at the end. Once traces are collected, we begin systematically reading and labeling them.

This process is adapted from *grounded theory* (Glaser and Strauss

2017), a methodology from qualitative research that builds theories and taxonomies directly from data.[24] Rather than starting with a fixed list of error types, we observe how the system behaves, label interesting or problematic patterns, and let the structure of failures emerge naturally.

In grounded theory, *coding* refers to assigning short descriptive labels to parts of the data that seem important (Strauss et al. 1990). We adopt the same idea here. For each trace, we read carefully and write brief notes about what we observe: where outputs are incorrect, where actions are surprising, or where the behavior feels wrong or unexpected.[25] Each note is a potential signal of a failure mode or quality concern.

When beginning, we recommend examining the entire trace as a whole and noting the first (most upstream) failure observed. Traces can be complex, and a single upstream error (e.g., misinterpreting the user's intent) often causes a cascade of downstream issues. Focusing on the root cause is more efficient and prevents us from getting bogged down in cataloging every single symptom. Furthermore, in many products, fixing the first error is all that is needed to resolve the entire chain of subsequent failures.

We record each trace and its corresponding open-coded note into a simple table or spreadsheet. When traces are long, it is often desirable to use an observability tool that can render traces in a more human-readable fashion out of the box. We will discuss this more in Section 10. At this stage, the structure is minimal: one column for the trace identifier, a column for the trace, and a column for the free-form annotation based on the point of first failure.

We illustrate with a few realistic traces from the real estate CRM assistant in Table 1. Note that for brevity, we only present a summary of each trace step.

At this stage, we do not attempt to group or formalize errors. We simply collect a rich set of raw observations. We can also assign each trace a holistic binary judgment: *acceptable* or *unacceptable*—a fourth column in our spreadsheet. This column is optional. Making a clear yes or no decision forces sharper thinking than vague scores (e.g., 3 out of 5). Even when the judgment feels borderline, we pick a side, and this process will clarify our evaluation criteria. However, the column may not be used in future steps, so it is okay to disregard making a holistic binary judgement.

Sometimes, when starting with initial labeling, it can be challenging to articulate precisely what feels "off" about a trace, or how to describe an observed failure.[26] If we find ourselves stuck, a helpful strategy is to switch temporarily to a more "top-down" approach. Instead of waiting for themes to purely emerge, we can consider a list of common LLM failure categories and actively look for their manifestations in our specific application's traces. For example, knowing that "hallucination" (generating

[24] Arawjo (2025b) also espouse thinking about error analysis through the lens of grounded theory.

[25] Grounded theory calls this process "open coding."

[26] Indeed, some failure modes or thresholds for what constitutes "bad" output only become clear after observing enough data. For instance, the GitHub Copilot team found that auto-generating excessively long code completions could be undesirable. While initially determining a precise length threshold for "too long" might seem arbitrary, an empirical approach can be taken: collect a set of completions that are weakly labeled as problematic (e.g., they correlate with low user acceptance rates or other negative feedback), and then analyze this set—perhaps by computing the average length—to establish a heuristic cutoff. This illustrates how

User Query	Trace Steps (Summary)	First-Pass Annotation
Find 3-bedroom homes under $600k near downtown that allow pets.	▪ Generate SQL query (missing pet constraint) ▪ Return 2 listings ▪ Draft email to client	Missing constraint: pet-friendly requirement ignored.
Set up showings for these two homes this weekend.	▪ Parse dates (Saturday, Sunday) ▪ Calendar API shows agent unavailable Saturday ▪ Propose showings both days	Invalid action: proposed unavailable times.
Send a property list to my investor client in San Mateo.	▪ Search for high ROI properties ▪ Draft casual email about starter homes	Persona mismatch: wrong tone and property type.

Table 1: Examples of early traces and open-coded annotations. Each trace surfaces a distinct failure pattern before formal categorization.

plausible but false information) is a common LLM issue, we could specifically inpsect each trace for instances where the assistant might be inventing facts, misrepresenting source information, or fabricating details—and when annotating, we can specifically describe how the hallucination occurred. Similarly, we could look for issues related to structured output (e.g., malformed JSON, incorrect list formatting), adherence to length constraints, or maintaining stylistic consistency, drawing inspiration from common output control types developers often need (Liu et al. 2024a). Vir et al. (2025) provide a large dataset of assertion criteria that can be used for inspiration.[27]

This initial round of annotation continues until we have surfaced a sufficiently broad set of failures. As a rule of thumb, we proceed until at least 20 bad traces are labeled and no fundamentally new failure modes are appearing. This point is known as *theoretical saturation* (Morse 1995): when additional traces reveal few or no new kinds of errors.

Reaching saturation depends primarily on the system's complexity, not merely on input diversity. Simple query types may saturate quickly. More complex agentic behavior (such as multi-step calendar and client personalization workflows) often requires deeper exploration.

Once a sufficient body of open-coded traces and early failure notes has been collected, we are ready to move to the next step: organizing these observations into structured failure modes.

[27] The goal isn't to force our traces into these predefined failure modes, but rather to use them as lenses to help identify and articulate specific problems that might otherwise be hard to pin down during initial labeling or open coding.

3.3 Axial Coding: Structuring and Merging Failure Modes

Open labeling or coding produces a valuable but chaotic collection of observations. Each trace yields its own raw notes: missing constraints, invalid actions, inappropriate tones, hallucinated facts. At this stage, the data is rich but unstructured. Without further organization, we cannot meaningfully quantify failures.

To impose structure, we draw on the next phase of grounded theory methodology: *axial coding* (Strauss et al. 1990; Glaser and Strauss 2017). Axial coding is the process of identifying relationships among codes generated in the first pass (i.e., open codes) and grouping them into higher-order categories. It moves us from isolated observations to a coherent list of unique, recurring failure types.

In our context, axial coding means reading through the body of open-coded traces and clustering similar failure notes together. Some patterns are obvious. Traces where the assistant proposes showings for weekends when the real estate agent is marked unavailable, or drafts emails listing properties outside a buyer's stated budget, cluster naturally into a broader failure mode: *violation of user constraints*.

Other failures reveal deeper distinctions only after reading several traces. In early coding, hallucinations of property features—claiming a home has solar panels when it does not—and hallucinations of client activity—scheduling a tour that the user never requested—were initially grouped together. But over time, it becomes clear they differ meaningfully: one misleads about external facts; the other fabricates user intent. We split them into *hallucinated listing metadata* and *hallucinated user actions*.

We present another example of splitting failure modes. An email drafted for a first-time buyer that uses investor jargon ("high cap rate returns") is not the same as a casual, slang-filled email sent to a luxury client ("sick penthouse deal coming in hot"). Both seem like "bad communication" at first. On closer review, the first reflects *persona misidentification*; the second reflects *inappropriate tone and style*—and we can classify them differently.

Axial coding requires careful judgment. When in doubt, consult a domain expert. The goal is to define a **small, coherent, non-overlapping set of binary failure types**, each easy to recognize and apply consistently during trace annotation.

While early clustering is often done manually, it is also possible to use a language model to assist the process. For example, after open coding 30–50 traces, we can paste the raw failure notes into our favorite LLM (e.g., ChatGPT, Claude) and ask it to propose preliminary groupings. A simple, effective prompt might look like:

Sample Prompt

```
Below is a list of open-ended annotations describing failures
in an LLM-driven real estate CRM assistant.  Please group
them into a small set of coherent failure categories, where
each category captures similar types of mistakes.  Each group
should have a short descriptive title and a brief one-line
definition.  Do not invent new failure types; only cluster
based on what is present in the notes.
```

LLM-generated groupings can help organize initial ideas, but they should not be accepted blindly. Proposed clusters often require manual review and adjustment to accurately reflect the system's behavior and the application's specific requirements.

At the end of axial coding, we have a list of distinct, binary failure modes, along with representative examples illustrating each one. These examples provide essential reference points as we move into the next stage: systematically labeling and quantifying failures across all traces. [28]

[28] There are other ways to cluster queries using classical machine learning techniques, but we have found that using a LLM to categorize annotations is both effective and pragmatic.

3.4 Labeling Traces after Structuring Failure Modes

At this stage, we have two artifacts:

1. A collection of traces, each with its initial, freeform "first-pass annotations."
2. A defined list of structured, binary failure modes (from axial coding), which represent the higher-order categories of errors we've identified.

Our next goal is to systematically apply these structured failure modes to each trace, creating a dataset ready for quantification. This means for every trace, and for each defined failure mode, we determine if that specific failure is present (1) or absent (0).

The process of assigning these structured labels bridges the gap from qualitative observation to quantitative data:

- **Review and Map Open Codes to Structured Failures:** We revisit our initial spreadsheet of traces and their first-pass annotations. For each trace, we compare its freeform annotation against our defined list of structured failure modes.
  - If we manually derived the failure modes in the previous step (axial coding), this involves applying that derived mapping.
  - If an LLM assisted in generating the failure mode categories, we now need to ensure these categories are consistently applied back to each trace's original annotation. This might involve the engineer carefully reviewing each trace's open code and assigning the appropriate structured failure mode(s). Alternatively, one could

prompt an LLM for each trace: *"Given this annotation on a trace: '[open_annotation here]' and our list of defined failure modes: '[list_of_failure_modes]', which failure modes apply?"* Any LLM-assisted mapping requires human review and spot-checking to ensure accuracy and consistency, and to refine the failure mode definitions if discrepancies arise. This review process itself is valuable, as it deepens our understanding of the failure modes and their manifestations

- **Populating the Structured Data Table:** We augment our spreadsheet (or database table) by adding new columns, one for each structured failure mode. For each trace, we then populate these columns with a 1 if the failure mode is present for that trace (based on the mapping above) and a 0 if it's absent.

For example, if a first-pass annotation for a trace was "SQL query missed the budget constraint and also used an aggressive tone in the generated email," and our structured failure modes include "Missing SQL Constraint" and "Inappropriate Tone," this trace would get a 1 in both those columns and 0s in others.

During this process, it is common to discover inconsistencies or edge cases that force a reevaluation. Some traces initially annotated as problematic may, on closer inspection, not cleanly match any defined failure mode. Other traces may reveal nuances that suggest refinements to the categories. We allow ourselves to adjust annotations or revise failure mode definitions as needed.

Once labeling is complete, we can quantify the prevalence of each failure mode. We can compute error rates for each failure mode, if in a spreadsheet (e.g., using a formula like `SUM(column)` to count occurrences, then dividing by the total number of traces to get a rate). Python libraries like Pandas, or features like pivot tables in spreadsheets, allow for more flexible aggregation, filtering, and calculation of failure rates across different segments of the data. This quantification allows us to see, for each failure mode, how often it appears across the dataset, which is critical for prioritization in the Improve phase.

3.5 Iteration and Refining the Failure Taxonomy

Iteration in error analysis is important. It's common to conduct two or three rounds of reviewing traces and refining the failure mode taxonomy. Early rounds often uncover initially missed failure modes, or reveal that the initial definitions for failure categories need adjustment. As we analyze more traces—either by labeling existing unlabeled ones or by sampling new queries as described in Section 3.1—our understanding of the data (thereby bridging the Gulf of Comprehension as in Figure 1) improves. We

continually refine the taxonomy by merging similar categories, splitting overly broad ones, or clarifying definitions as new error patterns emerge.

Note to readers

For multi-turn traces, we began by labeling only the "point of first failure" for each trace. This is the most efficient way to build our initial failure taxonomy and identify the most common root causes of errors.
But as we iterate and our evaluation needs mature, we may find this initial approach is no longer sufficient. For instance, if we need to rigorously measure the impact of a fix for a specific error like "Inappropriate Tone," we need to know every time it occurs, not just when it happens to be the first failure in a chain. This requires us to revisit our dataset and perform a more exhaustive labeling pass, annotating all instances of our defined failure modes within each trace. While this is more time-consuming, it provides a complete picture. It also helps explain why an automated evaluator (which we'll discuss in Section 5) might flag errors that our initial human labels missed; these are often real issues that simply weren't the first failure.
Therefore, a practical workflow is to start with "point of first failure" for broad analysis. Then, for specific, high-priority failure modes, we can re-annotate a subset of data exhaustively to get an accurate baseline before and after implementing a fix.

In the real estate CRM assistant project, a second round of analysis might expose a new error not apparent in the first batch; the assistant may occasionally misinterpret location names by defaulting to local cities without explicit clarification. For example, perhaps a user asking about "Springfield" listings received results from the wrong state entirely. This would surface the need for stronger disambiguation rules in both query generation and downstream SQL templates, leading us to add *location ambiguity errors* as a distinct failure mode.

Iteration is not endless. In practice, two serious rounds of open coding and re-annotation are often sufficient to approach theoretical saturation. Beyond that point, additional sampling typically yields diminishing returns, with few genuinely new failure types emerging.

3.6 Common Pitfalls

The most common mistake in early error analysis is failing to test on representative data. If the initial query set does not reflect the diversity and difficulty of real user behavior, the traces produced are uninformative. Either no serious failures occur, or the few failures that appear are not the ones that would arise in production.

A second common failure is skipping open coding altogether. Instead of reading real traces and observing how the system actually fails, teams often default to generic categories pulled from LLM research: "hallucination," "staying on task," "verbosity." These broad labels may sound reasonable, but without grounding in real examples, they often miss critical application-specific issues.[29] In the real estate CRM assistant, for instance, errors like proposing unavailable showings or misidentifying client personas are far more damaging than generic verbosity.

[29] **Hamel's Note:** The abuse of generic metrics is endemic in the industry as many eval vendors promote off the shelf metrics, which ensnare unsuspecting engineers into superfluous metrics and tasks.

Another frequent pitfall is the inappropriate use of Likert scales during early annotation. A Likert scale asks annotators to rate qualities (such as relevance or helpfulness) on a numeric scale, typically 1 to 5 or 1 to 7. While widely used in LLM evaluations (Chiang et al. 2023; Zheng et al. 2023; Kim et al. 2023), Likert scales introduce substantial noise when applied without a detailed rubric. This phenomenon has been widely observed in annotation and survey research: when rubrics are underspecified, rating scales lead to lower inter-annotator agreement and higher subjective variance (Artstein and Poesio 2008). In contrast, forcing binary decisions about specific failure modes—whether a problem occurred or not—produces more reproducible annotations (Husain 2025; Shankar et al. 2024c; Yan 2024).

Finally, treating initial annotations and failure modes as fixed is a critical error.[30] It is normal for annotation schemas to evolve after reviewing more data. New examples may reveal edge cases that require refining definitions. Freezing the schema too early locks evaluation infrastructure around an incomplete understanding of system behavior.

[30] A critical misstep in error analysis is excluding domain experts from the labeling process, especially for applications that require deep subject matter knowledge. Outsourcing this task to those without domain expertise, like general developers or IT staff, often leads to superficial or incorrect labeling. Making the data and labeling process accessible to domain experts is important; building or tailoring simple annotation interfaces can be helpful for this, as we discuss further in Section 10.

Overall, a careful, iterative error analysis phase avoids these mistakes. It ensures that evaluation metrics are shaped by the real behaviors of the system under realistic conditions, not by assumptions or borrowed taxonomies.

3.7 Summary

The **Analyze** phase is the cornerstone of effective LLM evaluation, providing the deep, qualitative understanding necessary before any meaningful measurement or improvement can occur. This section detailed a systematic, iterative approach to error analysis. We began by emphasizing the importance of **bootstrapping a diverse initial dataset** (Section 3.1), using real user queries when available or, if not, a structured method for generating representative synthetic queries based on important application dimensions.

We then walked through the process of **reading and labeling traces** (Section 3.2) using techniques adapted from grounded theory. This involves open coding—making detailed, first-pass annotations on observed behaviors and potential failures—and continuing until theoretical satura-

tion is approached. We also discussed strategies for when annotators get stuck, such as using a top-down approach by checking for known LLM failure types.

Next, we covered how to move from these raw observations to a manageable taxonomy by **structuring failure modes** (Section 3.3) through "axial coding." This step involves clustering similar open codes into a small, coherent, and non-overlapping set of binary failure categories.

Finally, we detailed the process of **quantifying failure modes** (Section 3.4). Throughout, we highlighted the importance of **iteration** (Section 3.5) in refining this failure taxonomy and cautioned against common pitfalls (Section 3.6) such as using unrepresentative data or skipping deep qualitative review.

Ultimately, the Analyze phase is not primarily about calculating performance scores. Instead, its critical output is a well-understood, application-specific **vocabulary of failure**: a clear, consistent set of defined failure modes that allows us to precisely describe, and subsequently measure, how and why our LLM pipeline isn't meeting expectations. This foundation is essential before proceeding to measure these failures at scale.

3.8 Exercises

1. **Synthetic Data Generation (Travel Assistant).**
   Define three key dimensions for a travel booking assistant. Then:

   (a) List at least three values for each dimension.

   (b) Write a prompt to generate 10 random structured tuples.

   (c) Write a prompt to convert one example tuple—e.g. (Find Flight, Luxury Traveler, Flexible Dates)—into a natural-language query.

Solution 3.1

Dimensions and values:

- *Task Type:* {Find Flight, Find Hotel, Find Flight+Hotel, Activity Recommendation, General Inquiry}
- *Traveler Profile:* {Budget Traveler, Business Traveler, Family Vacationer, Luxury Traveler, Solo Backpacker}
- *Date Flexibility:* {Exact Dates, Flexible Dates, Open-Ended}

(b) Prompt to generate tuples:

Sample Prompt

```
Generate 10 random combinations of (Task Type, Traveler
Profile, Date Flexibility) for a travel booking
assistant. Possible values: Task Type: Find Flight,
Find Hotel, Find Flight+Hotel, Activity Recommendation,
General Inquiry. Traveler Profile: Budget Traveler,
Business Traveler, Family Vacationer, Luxury Traveler,
Solo Backpacker. Date Flexibility: Exact Dates,
Flexible Dates, Open-Ended. Output each as: (Task
Type, Traveler Profile, Date Flexibility) Ensure no
duplicates and good coverage.
```

(c) Prompt to convert one tuple to a query:

Sample Prompt

```
We are generating synthetic user queries for a travel
assistant. Given: Task Type: Find Flight Traveler
Profile: Luxury Traveler Date Flexibility: Flexible
Dates Write a realistic query a luxury traveler might
use, reflecting flexibility in travel dates. Example
(for another tuple): (Find Hotel, Business Traveler,
Exact Dates) → "Need a 4-star hotel near the airport
in Seattle from Sept 10-12, must have airport shuttle."
Now generate a query for (Find Flight, Luxury Traveler,
Flexible Dates).
```

2. **Synthetic Data Generation (E-Commerce Chatbot).**
Define three key dimensions for an e-commerce chatbot. Then:

(a) List at least three values for each dimension.

(b) Write a prompt to generate 10 random structured tuples.

(c) Write a prompt to convert one example tuple into a natural-language query.

Solution 3.2

Dimensions and values:

- *Intent:* {Search Product, Compare Prices, Check Availability, Track Order, Return Request}
- *Customer Segment:* {Bargain Hunter, Brand-Loyal, Gift Shopper, Bulk Buyer}
- *Urgency:* {Immediate Need, Planned Purchase, Browsing}

(b) Prompt to generate tuples:

Sample Prompt

```
Generate 10 random combinations of (Intent, Customer
Segment, Urgency) for an e-commerce chatbot.  Intent:
Search Product, Compare Prices, Check Availability,
Track Order, Return Request.  Customer Segment:
Bargain Hunter, Brand-Loyal, Gift Shopper, Bulk Buyer.
Urgency:  Immediate Need, Planned Purchase, Browsing.
Output each as:  (Intent, Customer Segment, Urgency) No
duplicates; maximize variety.
```

(c) Prompt to convert one tuple to a query:

Sample Prompt

```
We are generating synthetic user queries for an
e-commerce chatbot.  Given:  Intent:  Compare Prices
Customer Segment:  Bargain Hunter Urgency:  Planned
Purchase Write a realistic query a bargain hunter
planning ahead might enter.  Example (for another
tuple):  (Search Product, Gift Shopper, Immediate Need)
→ "I need a birthday gift--best-rated wireless earbuds
under $100, today."  Now generate a query for (Compare
Prices, Bargain Hunter, Planned Purchase).
```

3. **Open Coding Practice (Travel Assistant).**
   Below is a multi-step agent trace. Perform open coding: write purely observational notes on each issue and identify the first (upstream) significant failure.

```
User Query:
  Book a budget flight from NYC to Rome on July 15 and a hostel nearby.

Trace Steps:
1. LLM → FlightSearch API
   FlightSearch(origin="NYC",
                destination="Rome",
                date="2025-07-15",
                class="economy",
                maxPrice=100)
2. API → LLM
   []  // no matching economy flights under $100
3. LLM → FlightSearch API
   FlightSearch(origin="NYC",
                destination="Rome",
                date="2025-07-15",
                class="business")
```

```
4. API → LLM
   [{flight:"BA100",class:"business",price:350},
    {flight:"UA200",class:"business",price:400}]
5. LLM → HotelSearch API
   HotelSearch(location="Trastevere",
               date="2025-07-15",
               maxPrice=80)
6. API → LLM
   ["Hostel A - Trastevere, $75/night",
    "Hostel B - Trastevere, $65/night"]
7. LLM → User
   Here are business-class flights and hostels in Trastevere...
```

Solution 3.3

- "Second flight search call drops the budget filter."
- "All flights returned are business class at $350+."
- "Hotel suggestions are in Trastevere, not near the Colosseum."

First significant failure: Step 3's FlightSearch call no longer enforces the user's budget or economy preference.

4. **Open Coding Practice (E-Commerce Chatbot).**
Perform open coding on this trace:

```
User Query:
  Find noise-cancelling headphones under $200
  with at least 20h battery life.

Trace Steps:
1. LLM → ProductSearch API
   ProductSearch(category="headphones",
                 filters={price<=200})
2. API → LLM
   [{id:123,name:"Model X",price:250,battery:22}]
3. LLM → User
   "Model X: $250, 22h battery life."
```

Solution 3.4

- "Search results include a $250 model despite price<=200."
- "Battery-life filter was never applied."

- "Final suggestion ignores budget constraint."

First significant failure: Step 1's ProductSearch call omitted the battery-life filter, leading to out-of-budget results.

5. **Axial Coding Practice (Cross-Domain).**
   You have collected these raw annotations:
   - "Omitted budget filter in flight search."
   - "Returned business class instead of economy."
   - "Suggested hostel in wrong area."
   - "Missed battery-life constraint."
   - "Recommended out-of-stock item."
   - "Used casual tone for a business inquiry."
   - "Scheduled hotel check-in on an unavailable date."
   - "Provided inaccurate product specs."
   - "Ignored shipping urgency."
   - "Generated flight date different from request."

   Cluster these into 3–4 structured, binary failure modes. For each, give a title and one-line definition.

Solution 3.5

- **Constraint Violation (Search):** Fails to apply user-specified filters (price, battery, availability) in retrieval steps.
- **Action/Timing Conflict:** Proposes actions or dates that violate known constraints (availability, dates).
- **Persona/Tone Mismatch:** Uses an inappropriate style or tone for the user's context.
- **Factual Inaccuracy:** Invents or misreports data (features, specs, locations) not supported by sources.

6. **Failure Mode Instance Augmentation.**
   Suppose a particular failure mode—e.g. "Location Ambiguity"—appears very infrequently in your collected traces. Describe how you would generate more queries likely to trigger that failure and then run them through your pipeline to collect additional instances. For one domain (travel or e-commerce):

   (a) Write a prompt template that generates queries targeting that failure mode.

(b) Provide one example prompt and the synthetic query it produces.

Solution 3.6

Strategy: Use an LLM prompt to create user queries containing ambiguous place names, so that running them through the travel assistant yields mislocalized results.

(a) Prompt template (Travel, Location Ambiguity):

Sample Prompt

```
Generate 15 user queries that mention a city name with
multiple possible locations (e.g. "Springfield,"
"Paris," "Bristol") without specifying state or
country.  The goal is to trigger the assistant's
wrong-city interpretation.  Output one query per line.
```

(b) Example:

Sample Prompt

```
Find me a flight from Springfield to Chicago on June
20.
```

When this query is run through the pipeline, it should produce a trace where the assistant uses the wrong "Springfield" (e.g. Springfield, MO instead of MA), giving another instance of the Location Ambiguity failure.

4 Collaborative Evaluation Practices

Our Analyze-Measure-Improve evaluation lifecycle (Figure 2) depends critically on "correct" human judgment. Sometimes human judgement is unreliable or inconsistent. For many evaluation criteria—especially those involving subjective qualities like helpfulness, tone, or creativity—this consistency can be difficult to achieve. In larger organizations or complex domains, relying on a single person's perspective is often insufficient or infeasible.

This section introduces collaborative evaluation practices—systematic approaches for involving multiple human experts or stakeholders. Our goal is to define, refine, and apply evaluation criteria collaboratively and reliably.

4.1 "Benevolent Dictators" Are Sometimes Preferable

In small to medium sized companies, it is often useful to appoint a principal domain expert.[31] In these organizations, there are usually one (maybe two) key individuals whose judgment is crucial for the success of the AI product. These are the people with deep domain expertise or represent our target users. Identifying and involving this Principal Domain Expert early in the process is critical.

[31] It's less about the number of employees and more about whether authority can meaningfully be delegated to a single individual. Whenever possible, appointing a trusted expert as the decision-maker is an effective way to unblock the labeling bottleneck that many teams face. This isn't always feasible or appropriate—it's a judgement call depending on the circumstances. This is generally more feasible the smaller the company is.

Why is finding the right domain expert so important?

1. **They Set the Standard**: This person not only defines what is acceptable technically, but also helps us understand if we're building something users actually want.
2. **Capture Unspoken Expectations**: By involving them, we uncover their preferences and expectations, which they might not be able to fully articulate upfront. Through the evaluation process, we help them clarify what a "passable" AI interaction looks like.
3. **Consistency in Judgment**: People in the organization may have different opinions about the AI's performance. Focusing on the principal expert removes annotation conflicts.
4. **Sense of Ownership**: Involving the expert gives them a stake in the AI's development. They feel invested because they've had a hand in shaping it. In the end, they are more likely to approve of the AI.

Some examples of Principal Domain Experts include:

- A psychologist for a mental health AI assistant.
- A lawyer for an AI that analyzes legal documents.
- A customer service director for a support chatbot.
- A lead teacher or curriculum developer for an educational AI tool.

Note to readers

In a smaller company, the Principal Domain Expert might be the CEO or founder. If you are an independent developer, you should be the domain expert (but be honest with yourself about your expertise). If you must rely on leadership, you should regularly validate their assumptions against real user feedback.

Many developers attempt to act as the domain expert themselves, or find a convenient proxy (e.g., their superior). This can be a recipe for disaster. People will have varying opinions about what is acceptable, it is impossible to make everyone happy.

Although the benevolent dictator model can work well, larger or cross-functional organizations may need to involve multiple stakeholders. We now turn to practical methods for doing this effectively.

4.2 A Collaborative Annotation Workflow

This section outlines a structured process for collaboratively defining and refining an evaluation criterion—such as a specific failure mode identified in Section 3. The goal is to produce a clear, consistent rubric that can be reliably applied by both humans and automated evaluators.

The workflow follows these steps:

1. **Assemble the Annotation Team:** Start by selecting two or more annotators with relevant domain expertise or stakeholder perspective.[32] Clarify their roles and the scope of the annotation task from the outset.

2. **Draft an Initial Rubric:** Create a first version of the rubric. This includes a working definition of the evaluation criterion (e.g., "Is the email tone appropriate for the client persona?") and a few illustrative Pass/Fail examples.[33]

3. **Select a Shared Annotation Set:** Curate a common set of 20–50 representative traces that all annotators will label. Choose examples that are directly relevant and include borderline or ambiguous cases to test the limits of the rubric.

4. **Label Independently:** Each annotator labels all examples on their own using the draft rubric. No discussion is allowed during this phase. The goal is to surface differences in how annotators interpret the rubric—not to reach consensus yet.

5. **Measure Inter-Annotator Agreement (IAA):** After labeling, compute inter-annotator agreement (IAA) scores using metrics discussed in Section 4.3. Low agreement flags areas where the rubric may be vague or inconsistent.

[32] In smaller teams, this may be just one person acting as a "benevolent dictator."

[33] This draft may emerge from earlier collaborative analysis—e.g., merging independently proposed failure modes from Section 3.

6. **Facilitate Alignment Session(s):** Bring annotators together to discuss cases where they disagreed. These sessions, described in Section 4.4, are used to understand the source of disagreement and improve the rubric—not just to revise past labels retroactively.

7. **Revise Rubric:** Update the rubric based on insights from the alignment discussion. Clarify definitions, refine examples, and adjust labeling criteria as needed.

8. **Iterate on the Process:** Iteratively repeat the process—selecting new examples, labeling independently, measuring IAA, and revising—until agreement reaches acceptable levels.

9. **Finalize Rubric and Labels:** Once agreement is consistently high, document the final rubric and generate a consensus-labeled dataset. This "gold standard" set can be used to fit and validate automated evaluators, and the rubric itself becomes the specification passed to the LLM-as-Judge (Section 5.3).

4.3 Measuring Inter-Annotator Agreement (IAA)

The simplest way to measure agreement between two annotators is **Percent Agreement**—the proportion of items where both annotators assigned the same label. Also called accuracy in classification tasks, this observed agreement, denoted P_o, is defined as:

$$P_o = \frac{1}{N} \sum_k n_{kk}$$

Here, N is the total number of items, and n_{kk} is the count of items both raters labeled with category k.

While easy to compute, P_o can be misleading because it doesn't account for agreement that could happen by chance—especially when one label dominates (e.g., mostly "Pass" as system quality improves).

To correct for this, we use *Cohen's Kappa* (κ) (Cohen 1960), the standard measure of inter-annotator agreement for categorical data. Kappa adjusts for chance agreement by comparing the observed agreement P_o to the expected agreement P_e, which is the probability that both annotators independently assign the same label by chance.

To compute P_e, we calculate the marginal probabilities: p_{1k} and p_{2k}, the fractions of items that annotators 1 and 2 assign to category k, respectively. The expected chance agreement is:

$$P_e = \sum_k p_{1k} \cdot p_{2k}$$

Cohen's Kappa is then:

$$\kappa = \frac{P_o - P_e}{1 - P_e}$$

A value of $\kappa = 1$ indicates perfect agreement, $\kappa = 0$ means agreement is what chance would predict, and $\kappa < 0$ implies systematic disagreement.

Python Implementation

```python
from collections import Counter
import numpy as np

def cohens_kappa(rater1, rater2):
    assert len(rater1) == len(rater2), "Annotation lists must have the same length."
    n_items = len(rater1)
    if n_items == 0:
        return 1.0  # Convention: 1 if nothing to disagree on

    # Observed agreement
    observed_agreement = sum(1 for i in range(n_items) if rater1[i] == rater2[i])
    po = observed_agreement / n_items

    # Expected agreement
    labels = set(rater1) | set(rater2)
    count1 = Counter(rater1)
    count2 = Counter(rater2)
    pe = sum((count1.get(label, 0) / n_items) * (count2.get(label, 0) / n_items)
             for label in labels)

    # Handle edge case
    if pe == 1.0:
        return 1.0 if po == 1.0 else 0.0
    else:
        return (po - pe) / (1 - pe)

# Example usage
# annotations1 = ['Pass', 'Fail', 'Pass', 'Pass', 'Fail', 'Pass', 'Fail']
# annotations2 = ['Pass', 'Fail', 'Fail', 'Pass', 'Fail', 'Pass', 'Pass']
# print(f"Cohen's Kappa: {cohens_kappa(annotations1, annotations2):.3f}")
```

Interpretation Following Landis and Koch (1977), values of κ are often interpreted as:

- < 0: Poor
- $0.00–0.20$: Slight
- $0.21–0.40$: Fair
- $0.41–0.60$: Moderate
- $0.61–0.80$: Substantial
- $0.81–1.00$: Almost perfect

In practice, we aim for $\kappa \geq 0.6$ to ensure labeling reliability.

Notes and Caveats. Cohen's Kappa is intended for measuring agreement between two human annotators who are peers. It is *not* used to evaluate LLM-as-Judge outputs against human labels that we consider "ground-truth."[34] Once human ground truth is established, we evaluate the LLM as a classifier, using metrics like TPR, TNR, FPR, and FNR (Section 5.3).

In cases with more than two annotators, use *Fleiss' Kappa* (Fleiss 1971). For ordinal, interval, or missing data, *Krippendorff's Alpha* (Krippendorff 2011) may be more appropriate. But for most binary or categorical labeling tasks with two raters, Cohen's Kappa remains a solid choice.

[34] **Shreya's Note:** You can use Cohen's Kappa to quantify whether LLM Judges agree with each other, or, between LLM and human judge. This will tell you how likely the human and LLM Judge are to agree. However, in practice, I see most people misuse Cohen's Kappa. If your human ratings are the ground truth, you should treat the LLM Judge as a classifier that you are trying to align with the ground truth, and follow standard ML metrics like TPR and TNR.

4.4 Facilitating Alignment Sessions and Resolving Disagreements

Alignment sessions are key to improving the annotation rubric by resolving differences between annotators. To prepare, calculate IAA scores and share them with the group. Highlight the traces where disagreements occurred. Appointing a neutral moderator can help keep the discussion on track.

The goal of the session is not to force agreement on past labels, but to understand why annotators interpreted things differently. We focus on the rubric itself: What part of the definition caused confusion? Was the example unclear, or does the rubric miss an edge case? A useful guiding question is: *What changes would make future annotators agree on this case?*[35]

[35] Timeboxing each disagreement can help maintain session momentum.

Use the following techniques to guide discussion and revise the rubric:

- **Clarify Wording:** Rewrite vague or overloaded terms to reduce ambiguity.
- **Add Examples:** Include concrete Pass/Fail examples, especially for edge cases that caused disagreement.
- **Add Rules:** Define clear decision rules for tricky situations (e.g., "If the response lacks feature X, label as Fail").
- **Split Criteria:** If one rubric covers multiple ideas, break it into smaller, more specific ones to reduce confusion.
- **Avoid Majority Vote (When Possible):** While voting can resolve individual labels, it doesn't improve the rubric. Use it only as a last resort for dataset labeling—not rubric design.
- **Escalation Path:** For persistent disagreement, escalate to a project lead or senior reviewer who can make a final call.

Document all rubric updates and the reasoning behind them. This record helps maintain clarity over time and supports consistent application as the project evolves.

4.5 Connecting Collaborative Labels to Automated Evaluators

The outputs of collaborative annotation—the refined rubric and gold-standard labeled traces—are critical for building strong automated evalua-

tors (Section 5.3).

First, a clearer rubric improves the behavior of the LLM-as-Judge. With explicit, well-tested instructions, the model is more likely to follow the intended criteria reliably.

Second, the consensus-labeled dataset provides a trusted ground truth for evaluating the LLM judge's accuracy. When we calculate metrics like True Positive Rate (TPR) or True Negative Rate (TNR) (Section 5.5), we can be confident in their validity, knowing the human labels were consistent. This makes our corrected estimates of failure rate ($\widehat{\theta}$ in Equation (3)) more reliable, which is essential for monitoring system performance.

Finally, the gold-standard dataset can be valuable beyond evaluation. It may later be used to fine-tune a smaller classifier or specialized model. While we don't cover fine-tuning in this book, it's worth saving all well-annotated traces and labels—they could support future training, auditing, or analysis tasks.

4.6 Common Pitfalls in Collaborative Evaluation

While collaborative evaluation can offer advantages, the process requires careful management to avoid common pitfalls that can undermine its effectiveness.

A frequent mistake is *skipping independent annotation*[36] If annotators discuss traces or the rubric extensively before labeling independently, their initial judgments become biased by the group discussion. This renders subsequent IAA measurements potentially meaningless reflections of "groupthink" rather than genuine initial agreement levels, masking real ambiguities in the rubric.

Second, starting with a poorly defined initial rubric also causes problems. Overly vague or incomplete definitions inevitably lead to low initial agreement, making the first annotation round inefficient and potentially frustrating for the annotators.[37]

Third, relying solely on percent agreement and ignoring the effect of chance agreement can provide a false sense of security regarding consistency. Employing metrics like Cohen's Kappa, which account for chance, gives a more realistic and trustworthy assessment.

Finally, a critical error is failing to "close the loop." The improved rubrics and gold-standard labels are valuable assets for validation. Neglecting to actually use them to improve LLM-as-Judge prompts means that much of the benefit derived from the collaborative effort is ultimately lost.

[36] As mentioned in Section 4.1, it can often be desirable to appoint a principal domain expert, in which case there is only one annotator. However, if we do have multiple annotators, it is important that we collect their labels independently.

[37] During alignment sessions, discussions can easily focus on changing past labels rather than refining the rubric for the future. The primary goal must remain improving the evaluation rules and definitions to ensure future consistency, not simply "winning an argument" about how a specific past trace should have been labeled.

4.7 Summary

Collaborative evaluation is an essential practice for rigorously assessing LLM outputs, particularly when dealing with criteria involving inherent subjectivity or ambiguity, or in domain-specific settings. By incorporating multiple perspectives, measuring agreement quantitatively (e.g., using Cohen's Kappa), and facilitating structured alignment sessions, teams can develop shared, consistent, and reliable evaluation criteria.

This meticulous process directly improves the quality and trustworthiness of human labeling efforts. Moreover, it provides a much stronger and more reliable foundation for building, validating, and deploying the automated evaluators required for effective, scalable monitoring of LLM pipelines in production.

4.8 Exercises

1. **Understanding IAA Metrics (Multiple Choice).**
   Why is Cohen's Kappa (κ) generally preferred over simple Percent Agreement (P_o) when measuring inter-annotator agreement for rubric development?

   (a) Cohen's Kappa is computationally simpler to calculate.

   (b) Percent Agreement can only be used for tasks with more than two categories.

   (c) Cohen's Kappa accounts for agreement that could have occurred purely by chance.

   (d) Percent Agreement is more sensitive to the number of annotators.

   Solution 4.1

   (c) Cohen's Kappa accounts for agreement that could have occurred purely by chance.

2. **Calculating Cohen's Kappa (Mathematical).**
   Two annotators (Rater A, Rater B) evaluated 10 summaries generated by an LLM for "informativeness" (Pass/Fail). Their ratings are below:

Trace ID	Rater A	Rater B
1	Pass	Pass
2	Fail	Fail
3	Pass	Fail
4	Pass	Pass
5	Fail	Fail
6	Pass	Pass
7	Fail	Pass
8	Pass	Pass
9	Pass	Fail
10	Pass	Pass

(a) Calculate the observed percent agreement (P_o).

(b) Calculate the expected chance agreement (P_e).

(c) Calculate Cohen's Kappa (κ).

(d) Based on Landis & Koch (1977) benchmarks, how would you interpret this Kappa score?

Solution 4.2

(a) Observed Percent Agreement (P_o):
Agreement occurs on traces 1, 2, 4, 5, 6, 8, 10 (7 traces).
$P_o = \frac{7}{10} = 0.70$

(b) Expected Chance Agreement (P_e):
Contingency Table:

		Rater B		
		Pass	Fail	Total
Rater A	Pass	5	2	7
	Fail	1	2	3
	Total	6	4	10

Proportions for Rater A: $p_{A,Pass} = \frac{7}{10} = 0.7$, $p_{A,Fail} = \frac{3}{10} = 0.3$
Proportions for Rater B: $p_{B,Pass} = \frac{6}{10} = 0.6$, $p_{B,Fail} = \frac{4}{10} = 0.4$
$P_e = (p_{A,Pass} \times p_{B,Pass}) + (p_{A,Fail} \times p_{B,Fail})$
$P_e = (0.7 \times 0.6) + (0.3 \times 0.4)$
$P_e = 0.42 + 0.12 = 0.54$

(c) Cohen's Kappa (κ):

$$\kappa = \frac{P_o - P_e}{1 - P_e} = \frac{0.70 - 0.54}{1 - 0.54} = \frac{0.16}{0.46} \approx 0.3478 \quad (1)$$

(d) Interpretation:
A Kappa score of approximately 0.348 falls into the "Fair" agreement range (0.21–0.40) according to Landis & Koch. This suggests that while there's some agreement beyond chance, the rubric for "informativeness" likely needs significant refinement as the annotators are not consistently applying it.

3. **Rationale for Collaboration (Short Answer).**
Your team is developing an LLM-based tool to generate marketing slogans. Why would a collaborative evaluation approach be particularly valuable for assessing the "catchiness" of these slogans, compared to relying on a single product manager's judgment? Provide two distinct reasons.

Solution 4.3

Two distinct reasons include:

(a) **Subjectivity of "Catchiness":** "Catchiness" is highly subjective and can vary based on individual preferences, cultural background, and even current trends. A single product manager might have a specific taste that doesn't represent the broader target audience or other stakeholders (e.g., marketing experts, sales). Collaboration allows for diverse interpretations to be considered, leading to a more robust and widely applicable definition of what makes a slogan "catchy" for the intended purpose.

(b) **Reducing Individual Bias and Identifying Blind Spots:** The product manager, like any individual, might have unconscious biases or particular stylistic preferences (e.g., favoring puns, or preferring shorter slogans) that could skew the evaluation. Collaborative evaluation with multiple annotators (perhaps including copywriters, brand strategists, or even representatives of the target demographic) helps to surface these individual biases and ensures that the final evaluation criteria are more balanced and comprehensive, reducing the risk of relying on a narrow or idiosyncratic definition.

4. **Applying the Workflow (Free Response).**
Imagine you are leading a team to develop evaluation criteria for "empathetic responses" from an LLM-powered mental health companion app. You've just assembled your annotation team (Step 1). Briefly describe the next three steps (Steps 2, 3, and 4) of the collaborative annotation workflow you would guide your team through.

Solution 4.4

The next three steps would be:

2. **Draft Initial Rubric:** I would work with the team, or assign a small subgroup, to draft an initial rubric for "empathetic responses." This rubric would include a clear definition of what

we mean by empathy in this context (e.g., acknowledging feelings, validating concerns, avoiding platitudes, offering gentle encouragement). It would also feature preliminary examples of responses that clearly "Pass" (are empathetic) and "Fail" (lack empathy or are actively unhelpful), based on our current understanding and perhaps some initial problematic outputs we've seen.

3. **Select a Common Set of Traces:** We would then curate a diverse set of approximately 20–50 real or realistic LLM-generated responses from the companion app to various user inputs. This set would be intentionally designed to test the rubric. It would include some straightforward cases, but also some ambiguous or challenging interactions where empathy might be particularly nuanced or difficult to express, to really "stress-test" our draft rubric. All annotators will receive this same set.

4. **Independent Annotation:** Each member of the annotation team would then independently review and label every trace in the common set using the draft rubric. They would assign a judgment (e.g., "Empathetic: Pass" or "Empathetic: Fail") to each response. I would strongly emphasize that this step must be done without any discussion or consultation among annotators to ensure that their initial judgments purely reflect their interpretation of the current rubric.

5. **Identifying Pitfalls (Scenario).**
A team developing an LLM for summarizing legal documents has two lawyers independently annotate 50 summaries for "factual accuracy against the source document." Before starting, they have a long meeting where they extensively discuss their individual approaches to identifying inaccuracies and agree on several specific examples. Their subsequent independent annotations show a Cohen's Kappa of 0.85. They are thrilled with this "Almost Perfect" agreement.

   What common pitfall from Section 4.6 might be at play here, and why might their high Kappa score be less indicative of true rubric robustness than they believe?

Solution 4.5

The primary pitfall at play here is **compromising or skipping truly independent annotation** due to the extensive pre-annotation discussion and agreement on examples.
While initial discussion to draft a rubric is good, their "long meet-

ing where they extensively discuss their individual approaches...and agree on several specific examples" before *any* independent annotation on a common set means their subsequent individual judgments are likely heavily influenced by the pre-established consensus, rather than being independent interpretations of a *draft* rubric.
The high Kappa score of 0.85 might be less indicative of true rubric robustness because:

- It could reflect "groupthink" or convergence on specific interpretations discussed beforehand, rather than the rubric itself being inherently clear and unambiguous enough to guide independent annotators to the same conclusion.
- If the rubric still contained subtle ambiguities that weren't explicitly covered in their pre-discussion, the high agreement might be due to the annotators implicitly relying on shared understandings developed during that meeting, understandings that aren't actually codified in the rubric. This means a new annotator, given only the rubric, might not achieve the same level of agreement.

Essentially, they might have measured agreement after already implicitly aligning, rather than using the IAA to identify areas *needing* alignment and rubric refinement. The true test of the rubric's clarity comes from its ability to guide annotators who haven't had such extensive pre-alignment discussions on the specific test set.

6. **Kappa and Chance Agreement (Mathematical/Conceptual).** Suppose two annotators are rating LLM responses for "politeness" (Polite/Not Polite). They rate 100 responses.

   Rater 1 rates 90 responses as "Polite" and 10 as "Not Polite."

   Rater 2 rates 95 responses as "Polite" and 5 as "Not Polite."

   They agree on 88 of the "Polite" responses and 3 of the "Not Polite" responses.

   (a) Calculate their observed agreement (P_o).

   (b) Without calculating P_e or Kappa fully, explain why their P_e (expected chance agreement) would likely be high in this scenario.

   (c) How would a high P_e affect their Cohen's Kappa score, assuming their P_o remains as calculated?

Solution 4.6

(a) Observed Agreement (P_o):
Total agreements = (agreed Polite) + (agreed Not Polite) = 88 + 3 = 91.
Total items = 100.
$P_o = \frac{91}{100} = 0.91$
(b) Why P_e would likely be high:
P_e (expected chance agreement) would likely be high because both raters have a strong tendency to classify responses as "Polite."
Rater 1: 90% Polite ($p_{1,Polite} = 0.9$)
Rater 2: 95% Polite ($p_{2,Polite} = 0.95$)
The chance agreement for the "Polite" category alone would be $0.9 \times 0.95 = 0.855$. Since the "Polite" category is so dominant for both raters, the overall P_e (which also includes the smaller chance agreement on "Not Polite") will be heavily influenced by this high value. When both raters predominantly use one category, the probability of them agreeing on that category by chance alone is high.
(c) Effect of high P_e on Kappa:
Cohen's Kappa is calculated as $\kappa = \frac{P_o - P_e}{1 - P_e}$.
If P_e is high (e.g., close to P_o), the numerator $(P_o - P_e)$ becomes small. This means that even with a high P_o (like 0.91 here), if a large portion of that agreement is attributable to chance (high P_e), the Kappa score will be substantially lower than P_o. A high P_e reduces the Kappa value because Kappa measures agreement *beyond* chance. If most of the agreement could have happened by chance, then the actual level of true, deliberate agreement on the rubric is considered lower by the Kappa statistic.

7. **Alignment Session Techniques (Short Answer).**
During an alignment session for a "helpfulness" rubric, annotators consistently disagree on traces where the LLM provides factually correct information but in a very technical way that a novice user might not understand. Describe two distinct techniques from Section 4.4 that the moderator could use to help resolve this disagreement and improve the rubric.

Solution 4.7

Two distinct techniques are:

(a) **Clarify Language / Add Decision Rules:** The moderator could lead a discussion to clarify what "helpfulness" means

in the context of the target user. This might involve adding a decision rule to the rubric, such as: "For a response to be considered 'Helpful: Pass,' it must not only be factually accurate but also presented in language accessible to a novice user. If highly technical jargon is used without explanation, the response should be marked 'Helpful: Fail' for accessibility, even if accurate." This directly addresses the point of contention.

(b) **Add Examples:** The moderator could suggest adding specific examples to the rubric that illustrate this exact scenario. They could include:

- An example of a response that is technically accurate but rated "Helpful: Fail" due to jargon, along with a rationale.
- An example of a response that is technically accurate and rated "Helpful: Pass" because it explains complex terms or uses simpler language.

These concrete examples would provide clearer guidance for future annotations of similar cases.

8. **Benefits of Collaborative Evaluation for Automated Systems (Free Response).**
Explain how the two main outputs of a successful collaborative annotation workflow (a refined rubric and a gold-standard labeled dataset) contribute to building more reliable LLM-as-Judge automated evaluators, as discussed in Section 4.5.

Solution 4.8

A successful collaborative annotation workflow produces:

(a) **A Refined, Unambiguous Rubric:**
This refined rubric serves as a much clearer and more precise specification when prompting an LLM-as-Judge. The explicit definitions, decision rules, and illustrative examples (especially for edge cases) that were agreed upon by multiple human experts can be directly incorporated into the judge's prompt. This reduces ambiguity for the LLM judge, leading it to make more consistent and accurate automated judgments that better align with the nuanced human understanding of the evaluation criterion.

(b) **A Gold-Standard Labeled Dataset:**
This dataset, where labels have been applied consistently based on the refined rubric (evidenced by high IAA), provides a highly

trustworthy ground truth. This gold-standard data is crucial for:

- **Validating the LLM-as-Judge:** When measuring the LLM-as-Judge's performance (e.g., its TPR, TNR), using these reliable consensus labels ensures that the calculated performance metrics for the judge are themselves more accurate and dependable.
- **Calibrating the LLM-as-Judge:** A few high-quality examples from this gold-standard set can be used as few-shot examples in the LLM-as-Judge's prompt, further grounding its behavior.
- **More Accurate Failure Rate Estimation:** Higher confidence in the LLM-as-Judge's TPR/TNR (because they were validated against better labels) leads to more reliable estimates of the true pipeline failure rate ($\widehat{\theta}$) when using corrections like the Rogan-Gladen formula.

In essence, collaborative evaluation ensures the human foundation upon which automated evaluators are built and validated is solid, leading to more trustworthy automated evaluation systems.

9. **Choosing an IAA Metric (Short Answer).**
Your team has three annotators (Alice, Bob, and Charles) independently labeling LLM outputs for "adherence to brand voice" using a predefined rubric with three categories: "Fully Adherent," "Partially Adherent," "Not Adherent." Which IAA metric mentioned in Section 4.3 would be more appropriate to use here than Cohen's Kappa, and why is Cohen's Kappa not the best fit?

Solution 4.9

Fleiss' Kappa would be more appropriate here.
Cohen's Kappa is specifically designed to measure agreement between **two** raters. Since there are three annotators (Alice, Bob, and Charles), Cohen's Kappa cannot be directly applied to assess the overall agreement among all three simultaneously (though pairwise Kappas could be calculated). Fleiss' Kappa is a generalization of Kappa that can be used to assess agreement among three or more raters when they are all rating the same items using the same categorical scale.

5 Implementing Automated Evaluators

In Section 3, we presented a framework to identify failures as part of the "Analyze" phase of our evaluation lifecycle (Figure 2). Here, we present how to approach the **Measure** phase. Measurement is about estimating the prevalence of our application's failure modes. The ability to quickly and reliably measure changes in success or failure rates is fundamental to an effective development loop. When we modify a prompt, adjust a retrieval strategy, or swap a model, we need to see concrete evidence of whether that change helped, harmed, or had no effect on specific issues. Manually re-evaluating a large set of traces after every pipeline change is slow and prone to inconsistency. Therefore, this section focuses on building **automated evaluators.**

Automated evaluators can compute various types of metrics.[38] Some metrics will be reference-free (as defined in Section 2), assessing inherent qualities of an output or its adherence to certain rules without needing a "golden" or ground-truth answer. Others will be reference-based, comparing the pipeline's output to a known correct or ideal response. For many failure modes, it is beneficial to conceptualize and, where feasible, implement both reference-free and reference-based checks.

In this section, we will first discuss *what* metrics to measure (Section 5.1), focusing on how to translate our categorized failure modes into precise, automatable metrics and how to approach the curation of reference examples when needed. Subsequently, we will explore *how to implement* automated evaluators for metrics (Section 5.2), covering both straightforward code-based checks, as well as sophisticated LLM-as-Judge techniques.

We continue with the same running example, Example 2, from Section 3: a real estate chat-based CRM assistant that interprets natural language queries from real estate agents, issues SQL, drafts client emails, reads calendars, and performs internet searches.

[38] **Shreya's Note:** https://github.com/huggingface/evaluation-guidebook is a nice online resource for automated evaluators and corresponding Jupyter notebooks (Fourrier and Community 2024).

5.1 Defining the Right Metrics (What to Measure)

Effective measurement begins by defining precise, quantifiable metrics for each failure mode identified during our error analysis. To illustrate, Table 2 lists several categories of failures we might have uncovered for our real estate CRM assistant (Example 2) during such an analysis.

After listing the failure modes, we can identify which ones can be simple to fix, due to ambiguity in our prompt specifications. Recall the Three Gulfs model from Figure 1, where developers struggle to write LLM applications due to having to navigate gulfs of Comprehension (i.e., understanding their data, as described in Section 3), specifying good prompts and pipelines, and LLM Generalization. Gulf of Specification

Failure Mode Category	Description
Missing SQL Constraints	Omits user-specified filters (e.g., `pets_allowed`, `max_price`) in SQL.
Incorrect SQL Aggregation	Generates SQL grouping by day instead of week due to ambiguous phrasing.
Invalid Tool Usage	Fabricates non-existent tool names or actions (e.g., `book_showings`).
Incomplete Email Content	Fails to include key client details (budget, location) in communications.
Persona-Tone Mismatch	Uses language unsuitable for the client's persona (e.g., informal with luxury client).
Incorrect Tool Sequencing	Attempts email composition before retrieving necessary listing data.
Unjustified Tool Calls	Issues actions (scheduling, messaging) not requested or grounded in prior steps.
Location Ambiguity	Interprets a common place name (e.g., "Springfield") with the wrong region.

Table 2: Example Failure Modes from Real Estate CRM Assistant Error Analysis

errors, stemming from ambiguous or incomplete instructions we provided to the LLM, can be easy to resolve if we edit the prompt. The evals that we write should mainly target the Gulf of Generalization.

In other words, we want to **fix ambiguity first, then measure generalization.** There are two reasons for this. First, for *efficiency reasons*: many specification failures can be resolved rapidly, often by simply adding clarity or detail to an existing prompt.[39] It can feel like a waste of time to build an automated evaluator for a failure mode that is easily resolved by a modification to a prompt.[40] Second, more fundamentally, we want our evaluation efforts to accurately reflect the LLM's ability to generalize correctly from clear instructions, not its capacity to decipher our potentially ambiguous intents. Building evaluators for poorly specified tasks essentially measures how well the LLM can "read our minds," which isn't a scalable or reliable indicator of robust performance.[41]

Consider an example from Table 2: "Incomplete Email Content" might be a Specification Failure if our email generation prompt doesn't clearly instruct the LLM on mandatory fields to include based on the context. We would need to improve our prompt. In contrast:

- "Missing SQL Constraints," if the prompt *clearly* stated the constraint (e.g., "max_price: $600k") but the LLM still omitted it, points to a Generalization Failure. This is a good candidate for an automated evaluator.
- "Persona-Tone Mismatch," assuming the client persona was clearly provided to the LLM, would also be a Generalization Failure.
- "Invalid Tool Usage" (calling a non-existent tool) is almost always a Generalization Failure (or a more fundamental model issue) if the available tools are well-defined to the LLM.

[39] **Shreya's Note:** I find that teams can rush to build evals for preferences that they never specified in the prompt—like concise responses or a specific structure (e.g., topic sentence, two supporting points, conclusion); a better approach is to first include such instructions and only create an evaluator if the LLM still fails to follow them.

[40] **Hamel's Note:** All evals incur non-negligible cost to create. You must assess if the effort of creating an automated eval outweighs the benefit of monitoring this specific error. If you have identified an obvious oversight in your prompt that resolves the issue, it may not be worth it to create an eval. Ultimately, it is a judgment call. For engineering related failures (e.g. bugs in your retrieval system), you should still lean on classic software engineering principles. Also, see later discussion on reference free vs. reference based metrics, as the former is costlier to verify and maintain than the later.

[41] **Hamel's Note:** Another large class of things I "just fix" when doing evals are tool specification errors, where the fix is glaringly obvious. For example, suppose the customer wants to reschedule an appointment but there is no proper tool exposed to the LLM for rescheduling, only for scheduling. The fix might be as simple as adding a rescheduling tool.

Once we have this refined list of Generalization Failures, we design specific, automatable metrics for each. We try to design both a reference-based and a reference-free metric for each significant failure mode. For any chosen failure mode, we must then carefully consider what an automated evaluator should specifically measure and, if pursuing a reference-based approach, curate the necessary high-quality reference examples.

For reference-based metrics, first, we need to curate input-output pairs where the LLM output exemplifies the desired failing behavior. The automated evaluator will then measure how closely the pipeline's actual output aligns with this "golden" reference. For example, if a failure mode is "Missing SQL Constraints" (Table 2), our reference data would consist of user queries and the corresponding ideal SQL, which *does* include all necessary constraints. The evaluator would check if the pipeline-generated SQL matches or correctly incorporates these constraints as seen in the reference.

Reference-based metrics are invaluable during iterative development, such as prompt engineering or model fine-tuning, and are often key components of Continuous Integration (CI) checks, which we will discuss later in Section 9.

For reference-free metrics, we define intrinsic properties or rules related to the failure mode. The evaluator measures adherence to these defined properties. For instance, continuing with "Missing SQL Constraints," a reference-free check might involve parsing the generated SQL to verify the presence of keywords associated with expected constraints (e.g., if a price was mentioned in the input, does the SQL reference a price-related column?). It could also check for general SQL validity.

For "Persona-Tone Mismatch," a reference-free approach might involve using an external judgment process (initially, perhaps, by a human expert)[42] to classify the tone of the generated email (e.g., "formal," "casual," "data-driven") and then check if this classification aligns with the target persona's known preferences. We could also check for the presence or absence of specific jargon or phrases indicative of a mismatch.

[42] We will detail methods for automating such nuanced judgments, including the use of other LLMs as evaluators, in Section 5.2.

Reference-free metrics are particularly powerful because they can often be adapted to run efficiently at scale on new, unlabeled data, enabling broader monitoring and assessment, potentially even in online settings (as we will discuss in Section 9).

As part of both types of metrics, we might incorporate *executability checks*. This means the metric goes beyond static analysis of the generated text to assess its functional correctness. For instance, if the LLM generates a SQL query, the evaluator might execute it against a test database to ensure it runs without error and yields plausible results. If it generates tool calls, the evaluator might simulate these calls to verify their logical consistency and potential to achieve the intended outcome. Such checks confirm whether the output is not just textually plausible but

also functionally sound.

Table 3 presents sample reference-based and reference-free metrics for some of the failure modes identified in Table 2.

Failure Mode	Reference-Based Metric	Reference-Free Metric
Missing SQL Constraints	Compare generated SQL AST against a golden AST to verify presence of all expected WHERE clauses	Regex or AST traversal to detect mandatory filter keywords (e.g., 'pets_allowed', 'max_price')
Invalid Tool Usage	Match sequence of tool calls to a reference trace containing only valid tool identifiers	Static schema check: ensure each invoked tool name exists in the registered tool registry
Incomplete Email Content	Token or word-level overlap between generated email and a reference email containing all required fields	Keyword presence check for mandatory sections (budget, location, client name) using regex patterns
Persona-Tone Mismatch	Compare judge's tone label on generated email against human-labeled references	LLM-as-Judge prompt for binary tone classification (Pass/Fail) based on persona definitions
Location Ambiguity	Compare disambiguated location entity IDs against ground-truth region codes	Geocoding-based check: map mentioned place to coordinates and verify consistency with user-specified region

Table 3: Reference-Based and Reference-Free Metrics for Real Estate CRM Failures

5.2 Implementing Metrics (How to Measure)

The goal of automated evaluators is to estimate the prevalence of each failure mode across a set of traces. Manual labeling at this scale is too slow and expensive, so we turn to automation. The challenge is ensuring each evaluator is well-grounded in the human-defined failure criteria—otherwise, the metrics are meaningless.

Each failure mode should have its own dedicated evaluator. Depending on the nature of the failure, this can be implemented either with code or with an LLM. **Code-based evaluators** are ideal when the failure definition is objective and can be checked with rules. For example:

- Parsing structure (e.g., checking JSON validity or SQL syntax).
- Using regex or string matching to detect required or forbidden phrases.
- Counting or verifying structural constraints (e.g., a summary has 3 bullet points).
- Executing the tool call generated by an LLM, and checking that the execution did not raise any errors.
- Logical checks (e.g., if the query asks for a dog-friendly apartment, does the output mention "pets allowed"?)

Code evaluators are fast, cheap, deterministic, and interpretable. They can support both reference-free checks (e.g., length limits) and reference-based ones (e.g., value extraction compared to a gold set).

LLM-as-Judge evaluators are used when the failure mode involves interpretation or nuance that code can't capture (Zheng et al. 2023). For

Failure Mode	Programmatic Evaluator	LLM-as-Judge Evaluator
Persona-Tone Mismatch	N/A (subjective)	Few-shot LLM prompt with definitions and examples to judge tone alignment
Unjustified Tool Calls	Regex-based filter to flag any action not grounded in input context	Prompt to assess if every tool call is requested or justified by prior steps

Table 4: Programmatic vs. LLM-as-Judge Evaluators for Real Estate CRM Failures

instance: Was the tone appropriate? Is the summary faithful? Is the response helpful? In these cases, we use a separate LLM—distinct from the main application—to judge outputs for a single failure mode. Each metric may require a different LLM-as-Judge evaluator, tailored to its specific failure definition. This is expected—and desirable—when evaluating multiple aspects of quality. LLM judges work best for narrowly defined, binary tasks (e.g., when the answer can be PASS or FAIL). Open-ended ratings or feedback are harder to supervise and align. However, using LLMs for evaluation comes with risks: bias, inconsistency, and inference cost. Prompt design, clear criteria, and calibration against human judgments are important.

Table 4 outlines how we implement different evaluators for different failure modes, choosing programmatic checks when definitions are objective and LLM-as-Judge prompts when nuance or interpretation is required.

Figure 6 illustrates the workflow for building an LLM-as-Judge evaluator, that we will subsequently expand upon.

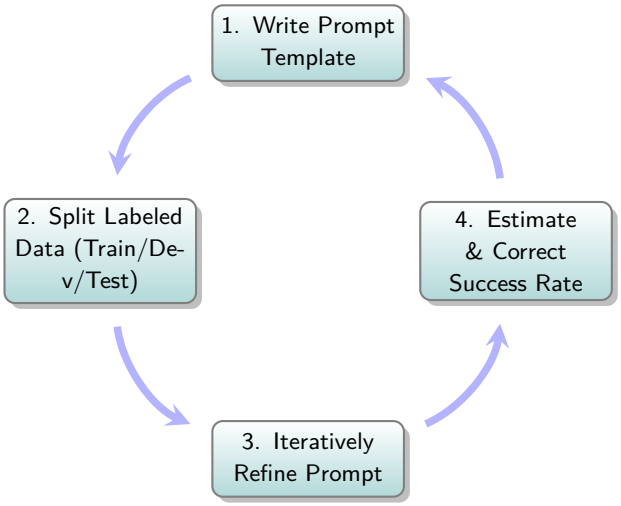


Figure 6: Workflow for building an LLM-as-Judge evaluator.

5.3 Writing LLM-as-Judge Prompts

The term "LLM-as-a-judge" was introduced by Zheng et al. (2023), who trained specialized LLM judges using large amounts of human preference

data. This fine-tuning process aligns the judge's behavior with human expectations and has shown strong performance in benchmark settings.

In practice, however, such fine-tuning is rarely feasible. Most applications lack sufficient labeled preference data, especially in early stages of development. Instead, we use off-the-shelf LLMs as judges and rely on prompt engineering to align their outputs with our intended failure definitions.

We now introduce a framework for designing LLM-as-Judge evaluators that are both reliable and grounded. Each evaluator corresponds to a single metric, which targets a specific failure mode identified during error analysis. We restrict the task to binary outputs—PASS/FAIL or YES/NO.[43] A well-structured LLM-as-Judge prompt contains four essential components:

1. **Clear task and evaluation criterion.** Each prompt should focus on one well-scoped failure mode. Vague tasks lead to unreliable judgments. Instead of asking whether an email is "good," we ask whether "the tone is appropriate for a luxury buyer persona."
2. **Precise Pass/Fail definitions.** We define what counts as a `Pass` (failure absent) and a `Fail` (failure present), directly based on the failure descriptions developed during error analysis (Section 3).
3. **Few-shot examples.** Examples help calibrate the judge's decision boundary. We include labeled outputs that clearly `Pass` and clearly `Fail`. These are best drawn from human-labeled traces, as discussed in Section 5.4.[44]
4. **Structured output format.** The judge should respond in a consistent, machine-readable format—typically a JSON object with two fields: `reasoning` (1–2 sentence explanation) and `answer` ("Pass" or "Fail"). This structure improves both accuracy and interpretability.

The example below shows a prompt template for evaluating tone appropriateness in a real estate CRM assistant (Example 2).[45] Each LLM-as-Judge evaluator follows this structure but is tailored to its specific failure mode.

Sample Prompt

```
You are an expert evaluator assessing outputs from a real
estate assistant chatbot.
Your Task: Determine if the assistant-generated email to
a client uses a tone appropriate for the specified client
persona.
Evaluation Criterion: Tone Appropriateness
Definition of Pass/Fail:
```

- `Fail: The email's tone, language, or level of formality`

[43] Restricting evaluation tasks to be binary helps us make alignment more tractable, especially because we are not fine-tuning the LLM judges.

[44] While we primarily focus on binary (Pass/Fail) judgments for clarity and alignment, if using finer-grained scales (e.g., 1–3 severity), it's critical to include examples for *every* point on the scale.

[45] **Shreya's note:** We pick a failure mode and examples that are easy to interpret for readers, but in practice, your criteria might be more complex and your few-shot examples might be "borderline" or difficult cases to evaluate.

is inconsistent with or unsuitable for the described client persona.

- **Pass:** The email's tone, language, and formality align well with the client persona's expectations.

Client Personas Overview:

- **Luxury Buyers:** Expect polished, highly professional, and deferential language. Avoid slang or excessive casualness.
- **First-Time Homebuyers:** Benefit from a friendly, reassuring, and patient tone. Avoid overly complex jargon.
- **Investors:** Prefer concise, data-driven, and direct communication. Avoid effusiveness.

Output Format: Return your evaluation as a JSON object with two keys:

1. reasoning: A brief explanation (1-2 sentences) for your decision.
2. answer: Either "Pass" or "Fail".

Examples:

Input 1:
Client Persona: Luxury Buyer
Generated Email: "Hey there! Got some truly awesome listings for you in the high-end district. Super views, totally posh. Wanna check 'em out ASAP?"
Evaluation 1: "reasoning": "The email uses excessive slang ('Hey there', 'awesome', 'totally posh', 'ASAP') and an overly casual tone, which is unsuitable for a Luxury Buyer persona.", "answer": "Fail"

Input 2:
Client Persona: First-Time Homebuyer
Generated Email: "Good morning! I've found a few properties that seem like a great fit for getting started in the market, keeping your budget in mind. They offer good value and are in nice, welcoming neighborhoods. Would you be interested in learning more or perhaps scheduling a visit?"
Evaluation 2: "reasoning": "The email adopts a friendly, reassuring tone ('great fit for getting started', 'nice, welcoming neighborhoods') suitable for a First-Time Homebuyer, and clearly offers next steps.", "answer": "Pass"

Now, evaluate the following:
Client Persona: {{CLIENT_PERSONA_HERE}}
Generated Email: {{GENERATED_EMAIL_HERE}}
Your JSON Evaluation:

5.4 Data Splits for Designing and Validating LLM-as-Judge

Designing an LLM-as-Judge closely resembles training a classifier—except the "training" happens through prompt engineering, not parameter tuning. Instead of learning from data via gradient descent, we manually select examples and write instructions that guide the model's behavior in-context. To ensure the resulting evaluator generalizes and doesn't overfit, we divide our labeled traces (from Section 3) into three disjoint sets:

- **Training Set.** A pool of labeled examples we may draw from when constructing the prompt. These examples are candidates for few-shot demonstrations—typically clear-cut `Pass` and `Fail` cases that illustrate the boundaries of the failure mode.[46]
- **Development (Dev) Set.** A separate, larger set of labeled traces used to refine the prompt. After each edit—whether changing instructions, rewording criteria, or swapping few-shot examples—we evaluate judge outputs on the dev set by comparing them to human labels. **dev set examples must never appear in the prompt itself.** This ensures we can measure how well the judge generalizes beyond the training examples.
- **Test Set.** A held-out set we use only to compute the alignment of the LLM judge, *after* finalizing the LLM judge prompt. We never look at this set during judge prompt development. It gives us an unbiased estimate of the judge's real-world accuracy—metrics like True Positive Rate and True Negative Rate—which we later use to adjust estimated success rates in production traces (Section 5.6).

[46] Few-shot demonstrations are concrete examples embedded in the prompt to guide the model's reasoning.

Reusing examples across splits—especially from Dev or Test in the prompt—leads to overfitting and inflated accuracy estimates.[47] Unlike traditional supervised learning, we don't need large training sets. In-context learning typically saturates after a small number of well-chosen examples.[48] As a result, we allocate more data to evaluation. **A typical split might assign 10-20% of labeled traces to the training set, and 40-45% each to dev and test.**

To reliably estimate judge performance, both dev and test sets should include a balanced number of `Pass` and `Fail` examples—ideally 30–50 of each. These proportions may not reflect real-world prevalence, especially when failures are rare. This imbalance is acceptable and deliberate, as we explain in Section 5.6.

[47] **Shreya's Note:** A common mistake I see in industry is inadvertently including evaluation cases as few-shot examples in the prompt. This leaks information and undermines the validity of reported metrics. Many people on X (formerly, Twitter) and in blog posts also incorrectly describe their evaluation: they either don't use a hold-out set at all, or they report accuracy on the development set as if it reflects the judge's true performance. This leads to highly inflated and misleading estimates of success rates.

[48] Studies suggest that LLM performance often plateaus after 1–8 examples (Min et al. 2022). More examples can even degrade accuracy or exceed context length limits.

5.5 Iterative Prompt Refinement for the LLM-as-Judge

After splitting labeled data into training, development, and test sets, we enter the core of judge construction: iteratively refining the prompt to align the LLM's decisions with expert labels. This process mirrors the

tuning loop of a classifier, but instead of adjusting parameters, we revise prompt text and examples. The loop proceeds as follows:

1. **Write a Baseline Prompt.** Start with an initial prompt using the components outlined earlier: task description, clear definitions, structured output format, and a few-shot example set drawn from the training set.
2. **Evaluate on dev set.** Run the LLM-as-Judge over all examples in the development set. Compare each judgment (`Pass` or `Fail`) to the human-provided ground truth.
3. **Measure agreement.** Let P be the total number of dev examples labeled Pass, of which p were judged Pass; let F be the total number labeled Fail, of which f were judged Fail. Then compute:

$$\text{TPR} = \frac{p}{P} \quad \text{(true positive rate)}, \qquad \text{TNR} = \frac{f}{F} \quad \text{(true negative rate)}. \tag{2}$$

   We refer to a positive as a pass, and a negative as a fail.
4. **Inspect disagreements.** Review false passes (judge said Pass but human said Fail) and false fails (judge said Fail but human said Pass) to identify ambiguous criteria or missing edge cases.
5. **Refine the Prompt.** Based on dev set errors:
   - Clarify task wording or tighten Pass/Fail criteria.
   - Swap in more illustrative few-shot examples from the training set.
   - If the dev set has some failure modes that are not in the training set, add representative traces to the training set and consider using them in the prompt for the next round.
6. **Repeat.** Re-evaluate the revised prompt on the same dev set and recalculate metrics. Continue until performance stabilizes or improves acceptably.

When to Stop Refining We stop when TPR and TNR reach satisfactory levels (e.g., >90%). Thresholds depend on application needs—missing a real failure may be costlier than flagging a false one.[49]

While tools like DSPy (Khattab et al. 2024) can automate this loop by optimizing prompts over a dev set, we recommend manual iteration first. It builds intuition about both the failure mode and the judge's behavior.[50]

If Alignment Stalls. If the judge continues to perform poorly—e.g., low TPR and TNR—consider one of the following strategies. We could use a more capable LLM: a larger or newer model may resolve subtle or context-sensitive errors. We could also decompose the criterion, or break

[49] This pragmatic approach of setting TPR/TNR targets is often more actionable than optimizing a single combined metric like an F1 score, though the latter can be considered if relative costs are well-defined.

[50] Another reason to manually iterate on LLM-as-Judge prompts is to be able to change labels—sometimes we might rethink our failure definition and decide a failing trace should actually pass, and vice versa.

a complex failure mode into smaller, more atomic checks, each with its own judge. Or, we could improving our labeled data, or add more diverse, high-quality examples to the training set—especially for edge cases.[51]

[51] **Hamel's Note:** It is important to make sure that we trust our labels. A common mistake companies make is to outsource labeling to people (or even LLMs) who do not have *any* context on the application.

5.6 *Estimating True Success Rates with Imperfect Judges*

Note to readers

This section is more mathematical than earlier ones. But we walk through it step by step and provide Python code to make it easy to apply, even without a background in statistics. PM's can skip to the end of this subsection, or use the code directly.

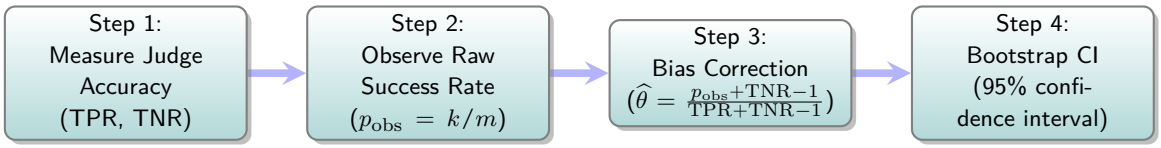


Figure 7: Workflow for estimating the true success rate with bias correction and confidence intervals.

After achieving consistent, high TPR and TNR on the dev set, we fix the LLM-as-Judge prompt and run it on the test set. This gives us two things. First, an estimated pass or success rate on the test set: the fraction of examples the judge labels as "Pass." Second, estimates of the judge's TPR and TNR, computed by comparing its predictions to ground-truth human labels.

But this only tells us how the judge behaves on the test split. In practice, we want to estimate how often a failure mode appears in a much larger (often unlabeled) dataset—such as new traces from a production pipeline. The problem is that our judge is imperfect. If we run it over thousands of new outputs and simply count the "Pass" predictions, we'll get a biased estimate of the true pass rate, because the judge occasionally misses failures or flags passes incorrectly.

This section shows how to:

1. Use the judge's TPR and TNR to correct its raw predictions and estimate the true pass rate θ for a metric—that is, the fraction of examples a human would consider passing the evaluation criterion.

2. Quantify the uncertainty around this estimate by constructing a 95% confidence interval.[52] If the upper bound is low, we can trust the LLM pipeline is performing acceptably. If the interval is wide, we may need to improve the TPR and TNR of the judge.

We now present a procedure (depicted in Figure 7) to estimate the true pass or success rate θ of an evaluation metric over new, unlabeled traces.[53]

[52] A **confidence interval** is a range that expresses our uncertainty about a number—in this case, the true rate of our LLM pipeline's failures. A 95% confidence interval means: if we repeated this whole evaluation process many times, 95% of the resulting intervals would contain the actual success rate.

[53] 🌶️ **Hamel's Note:** This is not just an academic exercise—it is the foundation for building trust in the judge's outputs. Without this trust, others will struggle to trust your product or your work. Many LLM-as-a-Judge guides imply that prompt tuning alone is sufficient. Don't fall into that trap.

Step 1: Measure Judge Accuracy. On our held-out test set, we compare judge predictions to human labels and compute TPR and TNR, as in Equation (2).

Step 2: Observe Raw Success Rate. We run the judge on m new, unlabeled traces and let k be the number it labels "Pass." The raw success rate is $p_{\text{obs}} = \frac{k}{m}$.

Step 3: Correct the Observed Success Rate. Because the judge is imperfect, p_{obs} is biased. We adjust it to estimate the true success rate (Rogan and Gladen 1978):

$$\widehat{\theta} = \frac{p_{\text{obs}} + \text{TNR} - 1}{\text{TPR} + \text{TNR} - 1} \quad \text{(clipped to } [0,1]\text{)}. \tag{3}$$

If $\text{TPR} + \text{TNR} - 1 \approx 0$ (e.g., 50% TPR and TNR), then the judge is no better than random chance, and the correction is invalid. In practice, $\widehat{\theta}$ is clipped to the range $[0,1]$ to handle numerical noise.

Step 4: Quantify Uncertainty with a Bootstrap. We quantify uncertainty in our corrected success rate estimate by bootstrapping over the test set's judge-vs-human labels: each iteration samples (with replacement) the full set of $(\text{human label}, \text{judge prediction})$ pairs, recomputes TPR and TNR on that sample, applies the correction to obtain a new $\widehat{\theta}^*$, and records it. After B iterations, the 2.5th and 97.5th percentiles of these $\{\widehat{\theta}^*\}$ values form our 95% confidence interval.

All together, Algorithm 1 describes the procedure for estimating the true success rate for an evaluation metric, along with a 95% confidence interval.

Algorithm 1: Using an LLM-as-Judge to estimate the success rate for some evaluation metric.

Input: Test set of N examples $\{(y_i, \hat{y}_i)\}_{i=1}^{N}$, where $y_i \in \{\text{Pass}, \text{Fail}\}$ is human label and $\hat{y}_i$ is judge prediction;
Unlabeled batch: m traces with k judged Pass;
Number of bootstraps B
Output: Point estimate $\widehat{\theta}$ and 95% CI $[L, U]$ for true success rate

`// Step 1: Judge accuracy on test set`
$\text{TPR} \leftarrow \dfrac{\left|\{i : y_i = \text{Pass}, \hat{y}_i = \text{Pass}\}\right|}{\left|\{i : y_i = \text{Pass}\}\right|}$;
$\text{TNR} \leftarrow \dfrac{\left|\{i : y_i = \text{Fail}, \hat{y}_i = \text{Fail}\}\right|}{\left|\{i : y_i = \text{Fail}\}\right|}$;
`// Step 2: Raw observed success rate`
$p_{\text{obs}} \leftarrow k/m$;
`// Step 3: Correct Judge bias`
$\widehat{\theta} \leftarrow \dfrac{p_{\text{obs}} + \text{TNR} - 1}{\text{TPR} + \text{TNR} - 1}$;
$\widehat{\theta} \leftarrow \text{clip}(\widehat{\theta}, 0, 1)$;
`// Step 4: Bootstrap CI`
$\mathcal{S} \leftarrow \{\}$;
for $b \leftarrow 1$ **to** B **do**
  draw a bootstrap sample of size N from the test set (with replacement);
  recompute TPR^* and TNR^* on that sample as above;
  $\theta^* \leftarrow \dfrac{p_{\text{obs}} + \text{TNR}^* - 1}{\text{TPR}^* + \text{TNR}^* - 1}$;
  $\theta^* \leftarrow \text{clip}(\theta^*, 0, 1)$;
  add θ^* to $\mathcal{S}$;
end
sort $\mathcal{S}$;
$L \leftarrow$ 2.5th percentile of $\mathcal{S}$;
$U \leftarrow$ 97.5th percentile of $\mathcal{S}$;
return $\widehat{\theta}$, L, U;

5.7 *Python Code for Estimating Success Rates*

Note to readers

We have open-sourced `judgy`, a small Python library (only depends on numpy), that contains the Python function below. Check it out at https://github.com/ai-evals-course/judgy.

Here, we provide a Python code snippet that computes a point estimate of the LLM pipeline's success rate $\widehat{\theta}$, as well as a 95% confidence interval for the true success rate θ.

```python
import numpy as np

def estimate_success_rate(
    test_labels,
    test_preds,
    unlabeled_preds,
    B=20000
):
    """
```

```
    Args:
      test_labels: array-like of 0/1, human labels on test set (1 = Pass).
      test_preds: array-like of 0/1, judge predictions on test set (1 = Pass).
      unlabeled_preds: array-like of 0/1, judge predictions on unlabeled data (1 = Pass).
      B: number of bootstrap iterations.

    Returns:
      theta_hat: point estimate of true success rate.
      L, U: lower and upper bounds of a 95% bootstrap CI.
    """
    test_labels    = np.asarray(test_labels, dtype=int)
    test_preds     = np.asarray(test_preds, dtype=int)
    unlabeled_preds = np.asarray(unlabeled_preds, dtype=int)

    # Step 1: Judge accuracy on test set
    P = test_labels.sum()
    F = len(test_labels) - P
    TPR = ((test_labels == 1) & (test_preds == 1)).sum() / P
    TNR = ((test_labels == 0) & (test_preds == 0)).sum() / F

    # Step 2: Raw observed success rate
    p_obs = unlabeled_preds.sum() / len(unlabeled_preds)

    # Step 3: Correct estimate
    denom = TPR + TNR - 1
    if denom <= 0:
        raise ValueError("Judge accuracy too low for correction")
    theta_hat = (p_obs + TNR - 1) / denom
    theta_hat = np.clip(theta_hat, 0, 1)

    # Step 4: Bootstrap CI
    N = len(test_labels)
    idx = np.arange(N)
    samples = []
    for _ in range(B):
        boot_idx = np.random.choice(idx, size=N, replace=True)
        lbl_boot = test_labels[boot_idx]
        pred_boot = test_preds[boot_idx]
        P_boot = lbl_boot.sum()
        F_boot = N - P_boot
        if P_boot == 0 or F_boot == 0:
            continue
        TPR_star = ((lbl_boot == 1) & (pred_boot == 1)).sum() / P_boot
        TNR_star = ((lbl_boot == 0) & (pred_boot == 0)).sum() / F_boot
        denom_star = TPR_star + TNR_star - 1
        if denom_star <= 0:
            continue
        theta_star = (p_obs + TNR_star - 1) / denom_star
        samples.append(np.clip(theta_star, 0, 1))

    if not samples:
        raise RuntimeError("No valid bootstrap samples; check inputs")

    L, U = np.percentile(samples, [2.5, 97.5])
    return theta_hat, L, U
```

To build intuition for how the success rate estimate varies with the

LLM-as-Judge's error rates, we present a small, illustrative thought experiment. ***These plots are not meant to model a real-world data collection process exactly, but to help readers reason about how success rate estimates are corrected and how judge imperfections impact uncertainty.*** For teaching purposes, we will assume a fixed number of human-labeled examples (50 successes and 50 failures) and a "ground truth" success rate of 80% on a larger unlabeled set (note that in practice, this 80% number is unknown).

In practice, our estimates of TPR and TNR are themselves empirical and become more reliable as we add more labeled data. However, to isolate the effects, we will hold one error rate constant while varying the other.

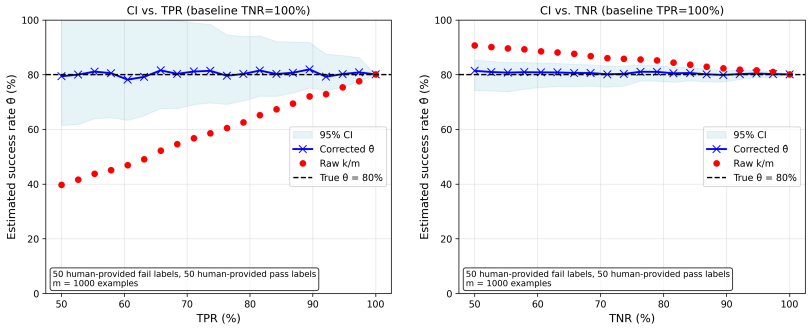
In the left plot in Figure 8, we assume perfect failure classification (i.e., TNR = 100%) and vary the judge's success detection (i.e., TPR) from 50% to 100%. When TPR is 50%, the judge is no better than a random guess for positive cases, and the correction formula is unstable; this is the conceptual equivalent of having zero information, so the behavior at this extreme should not be interpreted literally. As the judge improves, the uncorrected success rate (red) climbs toward 80%, its 95% CI (shaded region) contracts, and the bias-corrected estimate (blue) remains stable near the true value.

In the right plot, we hold true success detection (i.e., TPR) at 100% and vary TNR from 50% to 100%. At 50% TNR, half of real failures are mislabeled as successes, driving the uncorrected rate far above 80%. As TNR improves, the raw rate falls back toward the true 80%, its confidence interval tightens, and the bias-corrected line again correctly tracks the true rate.

Improving the TPR—the judge's ability to correctly identify true successes—tends to narrow the confidence interval for our estimated true success rate ($\hat{\theta}$) the most. To understand why, let's consider the scenarios depicted in Figure 8. When $\mathrm{TNR} = 1$, Equation (3) simplifies to

$$\widehat{\theta} = \frac{p_{\mathrm{obs}}}{\mathrm{TPR}}.$$

In this case, because the judge never mislabels failures, we recover the true pass rate by dividing the observed pass rate p_{obs} by the true positive rate. If TPR is small (i.e., judge misses many successes), this division "boosts" the underestimated p_{obs} by a correspondingly large factor. When TPR is small, even minor absolute variations in its bootstrapped estimate during the confidence interval calculation can lead to large relative changes in $1/TPR^*$, and thus to large fluctuations in the corrected $\hat{\theta}^*$. This increased variance in $\hat{\theta}^*$ naturally results in a wider confidence interval, reflecting greater uncertainty.



(a) When the judge misses many real successes (left), the raw success rate (red dots) starts far below the true 80% and the uncertainty (shaded) is huge. As it learns to catch more success (moving right), the raw rate rises toward 80% and the interval narrows, while the corrected line stays fixed on 80%.

(b) When the judge classifies many real failures as success (left), the raw success rate sits above the true 80% with a bit of uncertainty. As it improves at rejecting failures (moving right), the raw rate falls back to 80% and the interval shrinks.

Figure 8: How different judge errors bias the raw pass rate and how bias-correction recovers the true 80%. **Note: These plots assume fixed TPR/TNR for illustration. In practice, these rates are estimated from our data and their reliability improves as we label more examples.**

Key Takeaway 5.1 LLM-as-Judge Alignment Priorities

- The correction in Equation (3) reliably removes bias in both extreme error modes.
- Judge errors mainly inflate uncertainty (wider confidence intervals), rather than shifting the corrected estimate.
- Improving TPR, or the judge's ability to identify true successes, narrows the confidence interval the most.

5.8 Optional: Group-wise Metrics for Evaluating Multiple Outputs

Pipelines often generate multiple candidates per input—whether it's multiple completions, retrieved documents, or answer options. In these cases, we need metrics that evaluate the quality of the *group* of outputs, not just each one in isolation.

One key metric is **Success@k** (also called **Pass@k** in code generation). It answers a yes/no question: did *at least one* of the top k outputs meet the success criteria? For example, if a code generator produces five versions, Pass@5 checks whether any of them pass the tests (Jain et al. 2025; Chen et al. 2021).

Precision@k measures how many of the top k outputs are correct. It's the fraction of relevant items in the top k. For example, if an LLM proposes 5 API calls and 3 are valid, Precision@5 $= 0.6$. This tells us how good the outputs near the top are.

Recall@k measures coverage: how many of the total relevant items

are found in the top k. If 3 out of 10 correct answers appear in the top 3 suggestions, Recall@3 = 0.3. High precision with low recall means the system is right when it guesses, but misses a lot.

Beyond binary judgments, we can measure how *semantically close* the outputs are to the desired response. To do this, we use **embeddings**—vector representations of text. We can compute cosine similarity between the embedding of each candidate output and a reference (like the input or an ideal answer). This gives a score that reflects how well the meaning matches, even if the wording differs.

We may also care about **diversity**. Especially in generative settings, similar outputs are undesirable. For example, a fashion recommender should avoid suggesting nearly identical outfits every day. One common diversity metric is **average pairwise semantic similarity**.

Average Pairwise Similarity

Let $O = \{o_1, ..., o_N\}$ be the outputs, and e_i the embedding for o_i. Define a set P of index pairs to compare. Then:

$$\text{AvgSimilarity} = \frac{1}{|P|} \sum_{(i,j)\in P} \text{cos\_sim}(e_i, e_j)$$

where $\text{cos\_sim}(u, v) = \frac{u \cdot v}{\|u\|\|v\|}$.

A lower average similarity means higher diversity. But the choice of which pairs (i, j) to include in P matters. Here's a simple implementation in code:

```python
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Example: 5 embeddings, each 384-dim
embeddings = np.random.rand(5, 384)

# Pairwise similarity matrix
sim_matrix = cosine_similarity(embeddings)

# Average of unique pairs (exclude diagonal)
triu_indices = np.triu_indices_from(sim_matrix, k=1)
avg_similarity = sim_matrix[triu_indices].mean()

print(f"Average pairwise similarity: {avg_similarity:.4f}")
```

For example, in a fashion stylist application, we might only care about avoiding similarity between today's suggestion (o_i) and those from the past week (o_j where j indexes recent days). In that case, P would include pairs (i, j) where j ranges over the past 7 outputs. This results in roughly 7 comparisons per day, targeting short-term repetition. Averaging over all pairs would dilute this focus and miss near-duplicate suggestions.

In summary: when evaluating groups of outputs, use metrics like Success@k, Precision@k, and Recall@k for correctness, semantic similarity for relevance, and average pairwise similarity for diversity. Define what matters for your task and choose metrics accordingly.

5.9 Common Pitfalls

The most common mistake when implementing LLM-as-Judge evaluators is **omitting examples from the prompt entirely.** Without concrete examples, the model lacks grounding in what constitutes a failure for the task at hand. This often leads to vague or inconsistent behavior, even when the task definition is otherwise clear.

A second pitfall is attempting to do too much in a single prompt. Some teams try to evaluate multiple criteria at once—tone, content correctness, next steps—in a single Pass/Fail decision. This introduces ambiguity and makes it harder to diagnose errors. Breaking complex metrics into narrower, more specific metrics and prompts yields better alignment and more reliable judges.

Another common issue is skipping the alignment step altogether. Teams often assume the LLM-as-Judge will "just work" out of the box. Sometimes, that is true—especially for simple or broadly familiar metrics like sentiment polarity (i.e., positive or negative). But many evaluation metrics are domain-specific or tied to the unique "vibe" of a product: tone alignment for luxury clients, completeness of a listing summary, justification of a tool call. These require effort to align. Judges don't come pretrained on a product's values—we have to teach them. Prompt refinement and human-labeled validation are essential to ensure our evaluators actually reflects what matters in our pipelines. But aligning a judge—by refining prompts and validating its agreement with human labels—is an investment that pays off enormously. An aligned judge can replace expensive human evaluation across thousands of traces, making pipeline-wide monitoring feasible.

A related pitfall is *overfitting* the LLM-as-Judge prompt to their labeled traces. This often happens when teams include those same traces as examples and as part of the evaluation set used to measure alignment. This contaminates the metrics: TPR and TNR may appear high, not because the judge is generalizing well, but because it has memorized specific examples. Any trace used in the prompt must be excluded from the evaluation set.

Moreover, even after alignment, many teams fail to revisit the process. Production data can drift. New failure modes may emerge, LLM updates may shift behavior (Chen et al. 2024b), and evaluation metrics may evolve (Shankar et al. 2024d). We recommend re-running the alignment process regularly (e.g., weekly): continue labeling a handful

of traces, recomputing TPR and TNR, and checking whether confidence intervals remain acceptably tight. If not, retrain the judge. This is not a one-off task—it's part of the ongoing lifecycle of LLM-powered applications, much like monitoring in MLOps (Sculley et al. 2015; Shankar et al. 2024b). When observing new failure modes in production, we can add examples of them to all three splits (train, dev, and test).

5.10 Summary

Once we've identified failure modes, implementing evaluators allows us to measure them systematically and at scale. We define precise, failure-mode-specific metrics, prefer code-based checks when possible, and rely on LLM-as-Judge setups for more complex, nuanced, or domain-specific judgments. By aligning our judges through prompt refinement and validation against human labels, we ensure that automated evaluations reflect what we actually care about.[54]

Evaluation is not a one-time setup. Like any robust system, it must be maintained: revisited as data shifts, retrained as definitions evolve, and monitored for continued alignment. This work is essential for any LLM-powered product that aims to be reliable, interpretable, and improvable.

In the next section, we explore how to evaluate collaboratively—by bringing in multiple human perspectives, aligning evaluation criteria across stakeholders, and co-constructing judges that reflect shared standards.

[54] To further optimize the human annotation effort involved in validating and aligning evaluators, some recent research explores methods for intelligently selecting which data points to label (Gligorić et al. 2024).

5.11 Exercises

1. **Metric Definition: Specification vs. Generalization.**
   Your real estate CRM assistant (Example 2) sometimes fails to include a "closing costs estimate" when drafting offer letters, even though agents expect it.

   (a) Describe a scenario where this would be primarily a **Specification Failure**. How would you address it *before* building an evaluator?

   (b) Describe a scenario where this would be primarily a **Generalization Failure**. What kind of automated evaluator (reference-based or reference-free) might you design for it?

   Solution 5.1

   (a) Specification Failure:
   If the prompt for drafting offer letters never explicitly mentions including "closing costs estimates" or only vaguely implies it (e.g., "include all relevant financial details"). **Address before evaluator:** Refine the prompt to clearly instruct the LLM: ‘ Always include a section titled 'Estimated Closing Costs' and provide

a placeholder or request the agent to fill it in if an automated estimate is not available." **(b) Generalization Failure:** If the prompt clearly states to include closing cost estimates, but the LLM omits it for offer letters for a specific property type (e.g., condos but not single-family homes) or when the client is of a certain persona (e.g., an investor, where the LLM incorrectly assumes they don't need it). **Automated evaluator:** A **reference-based** evaluator could compare the generated letter to a golden reference letter that *does* include the closing cost section. A **reference-free** programmatic evaluator could use string matching to check for the presence of keywords like 'closing costs," ' estimate," or 'settlement charges" in the generated offer letter.

2. **Programmatic vs. LLM-as-Judge.**
   For each failure mode below from various applications, state whether a programmatic (code-based) evaluator or an LLM-as-Judge would be more appropriate, and briefly justify your choice. If both could work, explain the tradeoffs.

   (a) **E-commerce Chatbot:** Fails to apply a discount code to the cart total.

   (b) **Code Generation Assistant:** Generates Python code that produces a 'SyntaxError' when run.

   (c) **Travel Booking Assistant:** Suggests a travel itinerary that is "uninspired" or "lacks creativity" according to user feedback.

   (d) **Automated Summarizer:** Produces a summary of a news article that misses the main point.

Solution 5.2

(a) E-commerce Chatbot (Discount Code): Programmatic. We can simulate adding items to a cart, applying the discount code, and checking if the final total reflects the discount. This is a deterministic, rule-based check. **(b) Code Generation Assistant (SyntaxError): Programmatic**. The evaluator can attempt to execute or compile the generated code and catch 'SyntaxError' exceptions. This is an objective check. **(c) Travel Booking Assistant (Uninspired Itinerary): LLM-as-Judge**. "Uninspired" or "lacking creativity" are subjective qualities. An LLM-as-Judge, prompted with definitions and examples of inspiring vs. uninspired itineraries (perhaps based on user preferences or common travel patterns), would be better suited. A programmatic check would be very difficult to design. **(d) Automated Summarizer (Misses Main Point): LLM-as-Judge**. Determining the "main

point" of an article and whether a summary captures it requires semantic understanding. A code-based check (e.g., keyword overlap) might be too superficial. An LLM-as-Judge could be given the original article and the summary, and asked if the summary accurately reflects the core message, possibly with reference to a human-written "golden" summary or key points.

3. **LLM-as-Judge Prompt Design (Math Word Problem Solver).** You are designing an LLM-as-Judge to evaluate if an LLM-powered math tutor correctly solves elementary algebra word problems and, crucially, if its **explanation is step-by-step and easy to follow** for a student. Draft the key sections of the LLM-as-Judge prompt, focusing on the "clarity of explanation" aspect. Include:

   (a) Clear Task Description and Specific Evaluation Criterion (for the explanation).

   (b) Precise Definition of Pass/Fail for the explanation clarity.

   (c) One clear "Fail" example (problem, LLM's unclear explanation, reasoning) and one clear "Pass" example (problem, LLM's clear explanation, reasoning).

   (d) The desired structured output format (JSON, including assessment of both correctness and explanation).

Solution 5.3

You are an expert evaluator assessing an LLM math tutor's responses to algebra word problems. **Your Task:** 1. Determine if the final answer to the math problem is correct. 2. Specifically evaluate if the step-by-step explanation provided by the tutor is clear, logical, and easy for a student to follow.
Evaluation Criterion (for Explanation): Clarity and Logical Flow of Step-by-Step Explanation.
Definition of Pass/Fail (for Explanation Clarity):

- **Fail (Explanation):** The explanation skips crucial steps, uses overly complex vocabulary for the level, contains logical leaps that would confuse a student, or is poorly structured.
- **Pass (Explanation):** The explanation breaks the problem down into simple, understandable steps, defines variables clearly, shows all work logically, and uses age-appropriate language. Each step clearly follows from the previous one.

Output Format: Return your evaluation as a JSON object with these keys:

(a) `is_answer_correct`: Boolean (true/false).

(b) `explanation_evaluation`: A nested JSON object with keys:

- `reasoning`: Brief explanation (1-2 sentences) for your assessment of the explanation's clarity.
- `clarity_assessment`: Either "Pass" or "Fail".

Examples: —— Input 1 (Explanation Fail Example):
Problem: "Maria has 3 more apples than John. Together, they have 15 apples. How many apples does Maria have?"
Tutor's Explanation: "Let M be Maria's apples, J be John's. M=J+3. M+J=15. So, (J+3)+J=15, 2J=12, J=6. Maria has 9."
Evaluation 1: "is_answer_correct": true, "explanation_evaluation": "reasoning": "The explanation is too terse and skips the substitution and simplification steps. It doesn't clearly show how '2J=12' was derived from '(J+3)+J=15', which could confuse a student.", "clarity_assessment": "Fail" —— **Input 2 (Explanation Pass Example):**
Omitted for brevity... **Evaluation 2:** Omitted for brevity... —— **Now, evaluate the following:** Problem: {{MATH_PROBLEM_HERE}}
Tutor's Explanation: {{TUTOR_EXPLANATION_HERE}} Your JSON Evaluation:

4. **Data Discipline for LLM-as-Judge.**
You have 100 human-labeled traces for the "Persona-Tone Mismatch" failure mode (50 "Pass", 50 "Fail"). Explain how you would partition this data into Training, Development, and Test sets.

(a) What is the purpose of each set in the context of developing your LLM-as-Judge?

(b) Provide a reasonable split (e.g., number of examples for each set) and justify why this split is appropriate for LLM-as-Judge development.

(c) What is a critical mistake to avoid regarding the Test set examples during prompt engineering?

Solution 5.4

(a) Purpose of each set:

- **Training Set:** Provides a few high-quality examples to be included directly in the LLM-as-Judge prompt as few-shot demonstrations to guide its understanding of the task and the "Persona-Tone Mismatch" criterion.

- **Development (Dev) Set:** Used to iteratively evaluate and refine the LLM-as-Judge prompt. The judge's performance on this set (compared to human labels) guides adjustments to instructions, definitions, or the choice of few-shot examples taken from the training set.
- **Test Set:** A held-out set used for a final, unbiased evaluation of the finalized LLM-as-Judge's accuracy (TPR, TNR) after all prompt engineering is complete. Its performance on this set gives the most reliable measure of how well the judge will perform on unseen data.

(b) Reasonable split and justification: A reasonable split for 100 labeled traces (50 Pass, 50 Fail) could be:

- **training set:** 10 examples (e.g., 5 Pass, 5 Fail). This provides a small but diverse set of examples for in-context learning without overwhelming the judge or using too much of the limited labeled data.
- **Development (Dev) Set:** 40 examples (e.g., 20 Pass, 20 Fail). This provides a good number of samples to get statistically meaningful feedback during prompt iteration.
- **test set:** 50 examples (e.g., 25 Pass, 25 Fail). This leaves a substantial portion for a robust final evaluation.

This split prioritizes data for dev and test because LLM-as-Judge relies on in-context learning with few examples, rather than extensive training data for parameter updates. More data for dev and test allows for better tuning and more reliable final performance assessment. **(c) Critical mistake to avoid:** Never include examples from the test set (or the dev set) as few-shot demonstrations within the LLM-as-Judge prompt during its development and tuning. Doing so would mean the judge is "seeing the answers" for those cases, leading to inflated and unrealistic performance metrics on those sets, and a poor estimation of its true generalization ability. The Test set must remain completely unseen until the final evaluation.

5. **Judge Performance Metrics Calculation (Code Generation).** Your LLM-as-Judge for a "Correctness of Generated Python Function" evaluator issues a binary Pass/Fail verdict based on whether the function passes a set of unit tests. Here, *positive* means the judge predicts `Pass` (success), and *negative* means it predicts `Fail`. On a dev set of 60 human-verified generated functions:

- 25 functions actually **Fail** the tests; of these, the judge correctly labels 20 as `Fail`.
- 35 functions actually **Pass** the tests; of these, the judge correctly labels 30 as `Pass`.

Calculate:

(a) True Positive Rate (TPR): fraction of actual `Pass` the judge labels as `Pass`.

(b) True Negative Rate (TNR): fraction of actual `Fail` the judge labels as `Fail`.

(c) False Negative Rate (FNR) for `Pass`.

(d) False Positive Rate (FPR) for `Pass`.

Solution 5.5

Let

$$P = \text{number of actual Passes} = 35,$$
$$N = \text{number of actual Fails} = 25,$$
$$\mathsf{TP} = 30 \quad (\text{Passes correctly predicted}),$$
$$\mathsf{TN} = 20 \quad (\text{Fails correctly predicted}).$$

(a) $\mathrm{TPR} = \dfrac{\mathsf{TP}}{P} = \dfrac{30}{35} \approx 0.857.$

(b) $\mathrm{TNR} = \dfrac{\mathsf{TN}}{N} = \dfrac{20}{25} = 0.80.$

(c) $\mathrm{FNR} = 1 - \mathrm{TPR} = 1 - 0.857 = 0.143.$

(d) $\mathrm{FPR} = 1 - \mathrm{TNR} = 1 - 0.80 = 0.20.$

6. **Estimating True Success Rate (E-commerce Scenario).**
You built an LLM-as-Judge to detect when the assistant *fails to recommend* an alternative if a product is out of stock. Now we flip perspective and ask: what fraction of interactions *succeed* (i.e., the judge predicts `Pass` =success)? On a test set you measured:

$\mathrm{TPR} = 0.90$ (correctly identifies actual successes), $\mathrm{TNR} = 0.85$ (correctly identifies actual failures).

You run this judge on $m = 500$ new out-of-stock interactions. It flags $k = 60$ as `Fail`, so it flags $m - k = 440$ as `Pass`.

(a) What is the observed success rate p_{obs}?

(b) Using the correction,

$$\widehat{\theta} = \frac{p_{\text{obs}} + \mathrm{TNR} - 1}{\mathrm{TPR} + \mathrm{TNR} - 1},$$

compute the bias-corrected estimate of the true success rate $\widehat{\theta}$.

(c) If your SLO is a true success rate above 90%, what does your estimate suggest? How do TPR/TNR influence your confidence?

Solution 5.6

(a)

$$p_{\text{obs}} = \frac{\#\text{ judge-Pass}}{m} = \frac{440}{500} = 0.88.$$

(b)

$$\widehat{\theta} = \frac{0.88 + 0.85 - 1}{0.90 + 0.85 - 1} = \frac{0.73}{0.75} \approx 0.9733.$$

(c) A corrected success rate of 97.3% exceeds the 90% SLO comfortably. Since TPR=90% and TNR=85% are both high, the correction is reliable. However, any remaining uncertainty (quantified via a bootstrap CI) should still be checked before finalizing.

7. **Interpreting Confidence Intervals for Success Rate.**
You applied the bootstrap procedure (with $B = 20{,}000$) to estimate the true success rate of an LLM tutor solving algebra problems. The output was:

$$\widehat{\theta} = 0.985, \quad L = 0.970, \quad U = 0.993.$$

(a) What does the 95% interval $[0.970, 0.993]$ mean in plain language?

(b) If your minimum acceptable true success rate is 99%, what conclusion do you draw? What actions would you take?

(c) How could you narrow this interval (keeping $\widehat{\theta}$ fixed), based on the sensitivity analyses in Figure 8?

Solution 5.7

(a) We are 95% confident that the LLM's true success rate lies between 97.0% and 99.3%. Repeating this evaluation many times, 95% of such intervals would contain the real success rate.

(b) Because the upper bound (99.3%) exceeds 99% but the lower bound (97.0%) does not, we cannot guarantee the system meets the 99% bar. We should improve either the tutor itself or the judge's accuracy (TPR/TNR), then re-estimate.

(c) Two primary levers:

- *Increase test-set size* (more human-labeled examples), which

tightens both TPR/TNR estimates.

- *Improve judge accuracy* (especially reducing false positives by raising TNR), which narrows the bootstrap sampling variability.

8. **Common Pitfall Discussion.**
A team building an LLM-powered customer service chatbot is developing an LLM-as-Judge to detect "Offensive Language." They meticulously create a prompt with a detailed definition of offensive language and provide ten clear examples of outputs that are offensive (Fail) and ten that are not (Pass), all drawn from a set of 50 hand-labeled examples. They then run this judge on the same 50 examples and find it achieves 100% TPR and 100% TNR. They conclude their judge is perfect.

(a) What specific pitfall(s) from Section 5 (Common Pitfalls subsection) does this scenario illustrate?

(b) Why are the 100% TPR/TNR metrics likely misleading in this case?

(c) What should the team do differently to get a more realistic assessment of their LLM-as-Judge's performance?

Solution 5.8

(a) Pitfall(s) Illustrated: This shows both *overfitting the LLM-as-Judge prompt to the labeled traces* (using the same examples for few-shot grounding and evaluation) and *skipping a proper train/dev/test split*—the judge is tested on data it already "saw."
(b) Why metrics are misleading: Because the few-shot examples are part of the evaluation set, the judge essentially memorizes those cases. Perfect TPR/TNR on "seen" data does not guarantee any generalization to new, unseen inputs.
(c) What to do differently: Partition the 50 labeled examples into three disjoint sets:

(a) **Training (few-shot) set:** A small handful (e.g. 5–10) used only for in-prompt demonstrations.

(b) **Development set:** A separate group (e.g. 20) to iteratively refine the prompt—never included in the prompt itself.

(c) **Test set:** A held-out set (e.g. remaining 20–25) used only to compute the final, unbiased measurement of TPR/TNR.

Evaluating on an unseen test set ensures you measure true generalization rather than memorization.

6 Evaluating Multi-Turn Conversations

Until now, we've focused on evaluating single-turn interactions with LLMs. But many real-world applications—especially assistants and chatbots—require robust handling of multi-turn conversations. These longer interactions introduce new challenges: the system must maintain context, follow instructions over time, and respond coherently across turns. Here, we treat a single trace as the entire sequence of exchanges within a conversation: all user inputs, LLM responses, any tool calls, and intermediate steps taken from start to finish.

In this **quick section**, we mention a few strategies for evaluating multi-turn conversations. The core evaluation lifecycle—Analyze, Measure, Improve (Figure 2)—still applies. The key principles we have talked about in previous sections still hold. We begin with qualitative error analysis. We favor simple, often binary evaluation criteria. But multi-turn interactions add new layers of complexity—both in what we measure and how we collect meaningful evaluation data.

6.1 Evaluating at Different Levels

We find it helpful to approach multi-turn evaluation at three levels.

At the session level, we ask whether the full conversation achieves the user's intended goal. This broad assessment is often binary—Pass or Fail—and gives us a high-level view of system effectiveness.[55] For example, if a peer feedback assistant ends a session without eliciting any useful feedback from the user, we might mark the session as a failure, even if individual turns were well-formed. The overall task was not accomplished.

[55] We recommend starting with this kind of binary labeling, as discussed in earlier sections. It is easy to apply and forces us to clearly define what success means for a given task. Over time, these definitions can evolve—what starts as a single Fail category might be split into more specific failure modes or evaluated using more granular criteria. But starting with simple, high-level labels helps us quickly identify problematic sessions and build shared intuition about what bad looks like.

At the turn level, we can assess individual responses for quality, just as we would in a single-turn evaluation. We examine relevance, correctness, tone, etc. However, we typically reserve turn-level analysis for debugging specific failures. **It is not efficient to evaluate every turn in every trace.**

Finally, we assess conversational coherence and memory. Within a session, we check whether the assistant remembers what the user said in earlier turns. Across sessions, we examine whether the assistant appropriately retains knowledge through stored summaries or memory tools. These coherence failures are subtle but can be very important to our application—and they often only emerge after a few turns.

6.2 Practical Strategies for Multi-Turn Evaluation

We begin by collecting an initial dataset of multi-turn traces. A lightweight chat interface is often enough. Our team members simulate user behavior and engage in 10–15 tasks each, trying out different intents and personas. With just a few participants, we can quickly gather 100 or

more diverse traces.

We then manually review these conversations. We collect examples of both successes and failures. We ask simple questions: Did the assistant accomplish the user's goal? Did it follow instructions throughout? Where did things go wrong?

When we find failures, we try to isolate them. Can we reproduce the same issue using a simpler, reduced test case? For example, suppose that a shopping assistant gives the wrong return policy on the fourth turn of a conversation:

User: Hi, I want to know about returns.

Bot: Sure, what product are you asking about?

User: The X1000.

Bot: The return window for X1000 is 15 days. *(Incorrect; it should be 30.)*

We can simplify this into a single-turn test:

User: I have a question about the return policy for product X1000. What is the return window?

If the LLM still responds with "15 days," we have shown that failure is not necessarily multiturn. It's likely a retrieval or grounding issue, and we can debug it without the full multi-turn conversation context. But not all failures simplify cleanly. Some are inherently tied to the conversational setup. Consider a case where the assistant forgets the user's preference:

User: I'm looking for a movie.

Bot: Great! Any preferred genre?

User: I love sci-fi.

Bot: Okay, sci-fi. Any actors you like?

User: Not particularly.

Bot: How about *The Love Llama*, a heartwarming romantic comedy?

Here, the assistant clearly forgot the "sci-fi" constraint.[56] To isolate

[56] In practice, it can take 10 or more turns to run into such an egregious error. These errors are also more likely with non-frontier LLMs, or smaller or open-source LLMs.

the issue, we can truncate the trace and focus on just the final generation:

User: I'm looking for a movie.

Bot: Great! Any preferred genre?

User: I love sci-fi.

Bot: Okay, sci-fi. Any actors you like?

User: Not particularly.

(End here to sample the LLM's next response)

Given this partial conversation, we can sample the LLM multiple times to observe how it responds in the next turn. This allows us to evaluate whether the assistant retains and correctly uses user preferences stated just a few turns earlier. We can construct a dataset of these "N-1 turn" prompts, either by extracting prefixes from real conversations or generating them synthetically, to systematically test short-term memory and consistency.

We've also found it useful to introduce perturbations into real traces: modify the user's goal midway, add ambiguity, or correct the assistant mid-conversation. This helps surface robustness issues that don't appear in unmodified logs.[57] For example, we can take a real trace and insert a shift in the user's goal:

User: Hi, I'm looking for a restaurant recommendation.

Bot: Sure! What kind of cuisine are you in the mood for?

User: Maybe something Italian.

Bot: Got it. How about Trattoria Roma? It's well-rated and authentic.

(Perturbation: user adds a new constraint mid-conversation)

User: Actually, my friend is vegetarian. Can you suggest something vegetarian-friendly instead?

Bot: Your friend might enjoy the tiramisu. It's one of the best in town and is vegetarian.

[57] While simulating a user with a LLM can be helpful in targeting specific phenomena (e.g., "You are a user who gets increasingly frustrated"), simulations should be used cautiously. We have found that real or lightly modified conversations tend to reflect user behavior the best. As the capabilities of the models increase, user simulation may become an increasingly feasible approach.

This response is coherent and acknowledges the word "vegetarian," but it fails to address the user's revised goal. The user asked for a new restaurant recommendation that suits their friend's dietary needs. Instead, the assistant offers a dessert at the same restaurant, sidestepping the actual request. Perturbations like this help us evaluate whether the assistant can correctly interpret updated constraints and adapt its behavior, which is critical for realistic multi-turn interactions.

6.3 Automated Evaluation of Multi-Turn Traces

Once we've identified clear failure types, we can start building automated evaluators. These may take the form of LLM-as-Judge models, programmatic filters, or a hybrid of both.

An evaluator might operate at the session level, answering: Did the assistant meet the user's goal? Or it might target specific multi-turn behaviors: Did the assistant contradict itself? Did it retain important facts? Did it maintain the specified persona?

Not every evaluator needs to assess every turn. In fact, for many systems, we only need turn-level analysis when debugging specific paths. Session-level evaluation provides a broader performance signal with less overhead.

6.4 Addressing Common Pitfalls

We pay special attention to failure modes that correlate with position in the conversation. Many systems behave well in the first two or three turns, but then begin to degrade—either by forgetting context, ignoring instructions, or contradicting earlier responses. Our test sets include conversations of varying lengths to detect these issues.

We also balance holistic and granular evaluation. Session-level scores help us track overall system performance. Granular turn- or span-level analyses help us debug specific weaknesses. In practice, we often combine both: we identify failing sessions first, then perform deep dives on a handful of representative traces.

6.5 Summary

Evaluating multi-turn conversations requires extending single-turn evaluation techniques to account for memory, flow, and long-term success. Our process begins with qualitative analysis of real traces. We prioritize clear, binary judgments. We isolate failures to make them easier to reproduce and fix. And we automate only where patterns are stable and well-understood. By starting with real user behavior and building up, we ensure that our evaluation approach reflects actual usage—and drives meaningful improvement.

In the next section, we will talk about more complex architectures for agents that generate multi-turn traces, like retrieval-augmented generation (RAG) and tool calling.

7 Evaluating Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a foundational architecture in modern LLM systems. RAG is now pervasive across applications such as customer support, enterprise search, scientific QA, and coding assistance.

Evaluating RAG requires more than measuring final answer correctness. Each stage of the pipeline—query construction, retrieval, reranking, and generation—can independently introduce failure modes.[58] Moreover, these stages interact in complex ways: a weak retriever may lead the generator astray, while an imprecise question may cause retrieval to fail entirely.

[58] In some systems, these stages are executed agentically: an LLM agent proposes queries or reranking strategies, observes retrieval results, and iteratively refines its actions in a loop.

In this chapter, we focus on two parts of the evaluation loop: **analyze** and **measure**. We show how to:

- Analyze where failures arise—retrieval, generation, or their interaction,
- Measure retrieval quality using metrics like Recall@k, MRR, and NDCG@k,
- Evaluate generation using context-sensitive metrics such as faithfulness and relevance.

We begin by constructing evaluation datasets for retrieval, then introduce diagnostic metrics and methods for attributing failures across the pipeline.

7.1 Overview

A basic RAG pipeline retrieves context from a knowledge base (e.g., a document corpus, structured table, or the web) based on a query, and feeds it into an LLM for generation. Unlike traditional search, the consumer of this context is the LLM itself, which can process many passages simultaneously. This shifts the retrieval objective: high *recall* becomes critical to ensure we include all potentially relevant content.

However, simply retrieving as many documents as possible to stuff into an LLM's prompt is not ideal. LLMs have limited context windows, and irrelevant text can degrade performance. Moreover, LLM APIs charge per token, so excess tokens can significantly increase the cost of the request. To balance coverage and quality, most modern RAG systems use a multi-stage retrieval approach, as shown in Figure 9.[59]

[59] Having multiple stages of retrieval can improve both computational efficiency and relevance.

The first stage of retrieval scans the entire corpus of documents or records in a database and returns a broad set of candidates—which are the initial items (such as entire documents, specific text chunks, or database records) selected as potentially relevant to the query. We use fast, lightweight methods here: embedding-based search (e.g., Sentence-Transformers (Reimers and Gurevych 2019)) with approximate nearest neighbor algorithms, or traditional sparse retrieval methods like BM25.

Our goal in this stage is high *recall*, even at the cost of precision. Since we process the full corpus, speed is essential.

The second stage, *re-ranking*, refines the candidate set. We apply slower, more accurate models—such as cross-attention models—to re-order the candidates and select the most relevant documents. Some teams even use generative LLMs themselves to re-rank the documents; the prompt contains the retrieved documents and an instruction to output a new ordering. Rerankers better capture fine-grained relationships between the query and each candidate. Here, we prioritize precision and ranking quality.

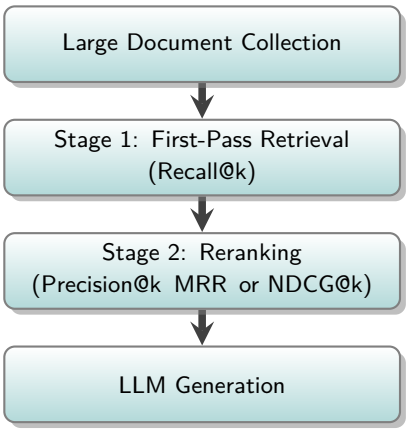


Figure 9: Two-stage retrieval in RAG. Stage 1 retrieves a broad set of candidates, prioritizing high recall of the relevant content. Stage 2 reranks and selects a smaller, focused subset for the LLM. Each stage is evaluated independently using appropriate metrics.

We start by evaluating the retrieval component on its own (Lauro et al. 2025). Before we assess the quality of generated answers, we need to verify that the system consistently finds the right context.

7.2 Synthetically Generating Query-Answer Pairs

To evaluate retrieval, we need a dataset of queries paired with the specific document *chunks* that contain the correct answers.[60] The most reliable option is to manually curate a set of realistic questions and map each to the exact chunk(s) in which the answer appears. While this provides high-quality supervision, it is often expensive and time-consuming to produce.

To scale more easily, we can generate synthetic question-answer-contexts using an LLM. We start by dividing the documents into chunks, which serve as retrieval "targets." For each chunk, we prompt the LLM to identify a salient fact and then generate a question that can only be answered using that fact. An example prompt to synthetically generate a query is as follows:

[60] A chunk is a contiguous span of text used as a retrieval unit. Chunks can range in size from a few sentences to entire documents, depending on how the corpus is partitioned. Common strategies include fixed-length token windows, section-based splits, or semantically coherent segments.

Sample Prompt

```
we are a helpful assistant generating synthetic QA pairs for
retrieval evaluation.
Given a chunk of text, extract a specific, self-contained
fact from it.  Then write a question that is directly and
unambiguously answered by that fact alone.  Return our output
in the following JSON format:
{ "fact":  "...", "question":  "..." }
Chunk:  "{text_chunk }"
```

To create more challenging synthetic queries, we can instruct the LLM to formulate questions that deliberately resemble content in multiple chunks but are only correctly answered by one of them. The process involves:

1. **Select a target chunk.** Choose a chunk A from the corpus that contains a clear, fact-based answer.
2. **Identify similar chunks.** Use an embedding model or keyword search to find chunks $B, C, \ldots$ that are similar to A but do *not* contain the answer.
3. **Prompt the LLM to generate a discriminative question.** Ask the model to write a question that is *only* answered by Chunk A, but which reuses terminology or themes that might appear in B or C. This encourages surface-level similarity while preserving ground truth uniqueness.

Example: Adversarially Generating Synthetic Questions

Chunk A (Correct): "In April 2020, the company reported a 17% drop in quarterly revenue, its largest decline since 2008."
Chunk B (Similar): "The company experienced significant losses in 2008 during the financial crisis but rebounded by the end of 2009."
Generated Question: "When did the company experience its largest revenue decline since the 2008 financial crisis?"
Only Chunk A contains the answer, but the presence of overlapping phrases in Chunk B makes it a plausible distractor.

This method helps us test whether the retriever can isolate the precise chunk needed to answer subtle or closely phrased questions, even when multiple candidates appear relevant.

Since synthetic questions can sometimes be unrealistic or off-domain, we should also filter them, like we filter unrealistic tuples in Section 3.1. We present one way to filter questions, grounded in our human expertise but automated by an LLM. First, we manually review a subset and rate each question for realism and relevance. Then, we could use these labeled

examples in a few-shot prompt to another LLM, instructing the LLM to score the rest of the synthetic queries. We keep only those that are deemed realistic, improving the dataset's alignment with real user queries.

Here is an example prompt for an LLM to filter synthetic queries:

Sample Prompt

```
we are an AI assistant helping us curate a high-quality
dataset of questions for evaluating a real estate information
retrieval system.  We have generated synthetic questions
and need to filter out those that are unrealistic or not
representative of typical user queries.
Here are examples of realistic and unrealistic user queries we
have manually rated:
Realistic Queries (Good Examples):

- "What are the property taxes for 123 Main St?"
  Rating:  5 Explanation:  Very typical user query--concise,
  information-seeking, property-specific.
- "Show me 3-bedroom houses in the North Berkeley area with a
  backyard."
  Rating:  5 Explanation:  Natural and detailed filter-based
  query for a real estate search tool.
- "When is the next open house for the condo on Elm Avenue?"
  Rating:  4 Explanation:  Reasonable and time-bound query;
  very plausible in a real estate context.
- "Does the property at 456 Oak Rd have a garage?"
  Rating:  5 Explanation:  Direct, fact-based query typical
  of prospective buyers.

Unrealistic Queries (Bad Examples):

- "Tell me about the color of the third brick from the left
  on the chimney of the house that was sold yesterday."
  Rating:  1 Explanation:  Overly specific and unnatural--no
  real user would ask this.
- "If I combine the square footage of all houses listed in
  zip code 94704, what is the total?"
  Rating:  2 Explanation:  Odd aggregation not representative
  of typical search behavior.

our Task:  For the following generated question, please:

- Rate its realism as a typical user query for a real estate
  application on a scale of 1 to 5 (1 = Very Unrealistic, 3 =
  Neutral/Somewhat Realistic, 5 = Very Realistic)
- Provide a brief explanation for our rating, comparing to
  the examples above if helpful.

Generated Question to Evaluate:  "{question_to_evaluate}"
Output Format:  Rating:  [our 1-5 rating] Explanation:  [our
brief explanation]
```

Note that we use a Likert-style 1–5 rating here because the goal is to fuzzily rank queries by realism and domain relevance, not to measure failure rates or judge correctness as in LLM-as-Judge or automated evaluator settings. This task does not require precise accuracy; instead, it supports the construction of a "good enough" dataset to begin testing our LLM pipeline.

7.3 Metrics for Retrieval Quality

Once we have a dataset of queries and their corresponding relevant document chunks, we can evaluate our retriever using standard ranking metrics. The choice of metric depends on what matters for our downstream application.

Precision@k and **Recall@k** are the most basic metrics to evaluate in a retrieval system. They evaluate the quality of the top k documents retrieved by the system in response to a query, where k might be user defined.[61]

- **Precision@k** indicates the proportion of the top k retrieved documents that are actually relevant. It is calculated as:

$$\text{Precision@k} = \frac{\text{Number of relevant documents in the top } k \text{ results}}{k}$$

- **Recall@k** indicates the proportion of all known relevant documents (for a given query) that are found within the top k retrieved documents. It is calculated as:

$$\text{Recall@k} = \frac{\text{Number of relevant documents in the top } k \text{ results}}{\text{Total number of relevant documents for the query}}$$

These metrics intrinsically operate on a binary notion of relevance: a document is either considered relevant or not. If our evaluation dataset uses graded relevance scores (e.g., a scale from 0 for irrelevant to 3 for highly relevant), we would first binarize these scores by applying a threshold. For example, all documents with a score > 1 might be treated as "relevant" for the computation of Precision@k and Recall@k, while others are treated as 'not relevant'.

In practice, we often care more about recall than precision for the first stage of RAG. Modern LLMs attend more strongly to salient tokens, so they can often ignore irrelevant content if the key information is present. But if that information is missing altogether—i.e., low recall—then the generator has no way to produce a correct answer. In early retrieval stages, ensuring that relevant content is included is far more important than excluding noise.

Mean Reciprocal Rank (MRR) measures how early the first relevant document appears in the ranking. It assigns a score of $1/\text{rank}$ if a relevant document is found in the top k results, and 0 otherwise. MRR is especially useful when only one key fact is required to answer a query. Even

[61] For some applications, k can vary between different queries. For instance, in a RAG system for real estate information, when evaluating retrieval for a query like, "What is the property tax for 123 Main St?", k might be kept small (e.g., $k = 1$ to 2). This is because the LLM likely needs only a single, specific fact from one core document (the property listing). However, for a query such as, "Summarize recent market trends for 3-bedroom houses in downtown and list comparable new properties under $1.2M," evaluation with a larger k (e.g., $k = 5$ to 10) would be more appropriate. This ensures the LLM receives sufficient diverse context (e.g., multiple market reports, several property listings) to synthesize a comprehensive answer.

though the LLM may process multiple retrieved chunks, ranking the correct one first can reduce noise and simplify reasoning. This is particularly helpful in cases where the LLM might over-prioritize earlier context (Liu et al. 2024b) or where the query implicitly asks for a single specific fact (e.g., "When are showings available for property X?"). For multi-hop or multi-source reasoning, however, MRR is less informative than Recall@k or NDCG@k.[62]

[62] A multi-hop question is one that requires information from 2 or more distinct, non-contiguous sections of documents.

Normalized Discounted Cumulative Gain (NDCG@k) extends ranking evaluation to settings with graded relevance. It rewards placing more relevant items higher in the list. First, we compute:

$$\text{DCG@k} = \sum_{i=1}^{k} \frac{\text{rel}_i}{\log_2(i+1)}$$

Here, rel_i is the relevance score of the document at rank i, e.g., from a scale such as 0 (irrelevant) to 3 (highly relevant) (Järvelin and Kekäläinen 2017). We then compute the ideal DCG (IDCG@k) by sorting the same set of documents in order of decreasing relevance. To compute the IDCG@k, we sort these retrieved documents by their true relevance scores in descending order. This sorted list represents the "ideal" order for these specific retrieved items. If our system retrieved more than k documents, we consider only the top k from this ideal sorted list for the IDCG@k calculation. Then, we apply the DCG formula to this ideally ordered list of k documents:

$$\text{IDCG@k} = \sum_{i=1}^{k} \frac{\text{rel}_{\text{ideal},i}}{\log_2(i+1)}$$

Here, $\text{rel}_{\text{ideal},i}$ is the true relevance score of the document found at position i in our "ideal" ordering of the retrieved documents. Essentially, IDCG@k quantifies the maximum possible DCG@k score our system could have obtained for the specific documents it retrieved, assuming it had ranked them in the perfect order of relevance. This makes NDCG@k a fair metric, as it normalizes our system's actual DCG@k score against the best possible score achievable with that particular retrieved set of documents.

Finally, NDCG@k is the ratio:[63]

[63] See Question 7.4 for an example of how to compute NDCG@k.

$$\text{NDCG@k} = \frac{\text{DCG@k}}{\text{IDCG@k}}$$

Why might NDCG@k be important for RAG? For one, sometimes retrieved chunks aren't equally useful. For a query like "Summarize market trends for luxury condos downtown" (related to the real estate agent assistant example described in Example 2), a recent market analysis report is highly relevant, while a general city economic overview is less relevant. NDCG@k rewards systems that rank the market report highest.

Another reason involves practical LLM constraints. Context windows are not infinite. If the total size of the top k retrieved documents exceeds the limit, only the first few documents will be used. NDCG@k is critical here because it ensures the most relevant documents are prioritized and make it into the truncated context.

7.4 Evaluating and Optimizing Chunking Strategies

The way documents are divided into smaller pieces, or "chunks," is a critical decision in any RAG system. This strategy directly impacts both retrieval metrics and the final quality of the generated answer. Key parameters include:

- **Chunk Size:** The number of tokens or sentences in each chunk.
- **Overlap:** The number of tokens or sentences shared between consecutive chunks to preserve context across boundaries.
- **Chunking Method:** The technique used to define chunk boundaries, such as fixed-length windows, splitting by sentences or paragraphs, or more advanced semantic segmentation.

One question we face when building RAG systems is: **What is the optimal chunking strategy for our specific use case?** There's no single best answer; the ideal strategy depends on the nature of our documents and the types of questions we expect. For many applications, a straightforward approach using **fixed-size chunking** with some overlap is a reasonable starting point. However, the effectiveness of this method hinges on finding the right size and overlap values. This is a classic hyperparameter tuning problem that we can address with a **grid search**.

In a grid search, we define a range of candidate values for each parameter and then systematically evaluate the retrieval performance for every possible combination. For example, we might test:

- **Chunk Sizes:** 128, 256, 512 tokens
- **Overlap Percentages:** 0%, 25%, 50%

For each configuration, we would re-index our entire document corpus and measure retrieval metrics like `Recall@5` and `NDCG@5` on our evaluation dataset. The results can be summarized in a table to identify the best-performing combination.

Chunk Size (tokens)	Overlap (tokens)	Recall@5	NDCG@5
128	0	0.82	0.69
128	64	0.88	0.75
256	0	0.86	0.74
256	**128**	**0.89**	**0.77**
512	0	0.80	0.72
512	256	0.83	0.74

In this example, a chunk size of 256 tokens with 50% overlap (128

tokens) yields the best retrieval performance.

While grid search can optimize a simple chunking strategy, **fixed-size chunking often fails when dealing with complex or heterogeneous documents.** The problem is that arbitrary boundaries can separate interconnected pieces of information, making it impossible for the retriever to fetch a complete context. Consider a question like, *"What is the main result in Table 2, and how does it relate to the introduction?"* A fixed-size chunking strategy might place Table 2 in one chunk and the relevant introductory paragraph in another. The retriever might only return one of these, leaving the LLM without the necessary information to form a complete answer. To address this, we need more sophisticated content-aware chunking strategies; for example:

1. **Leveraging Natural Document Structure:** If our documents have inherent boundaries—such as sections in a report, paragraphs in an article, or steps in a recipe—using these to define our chunks is often more effective than imposing an arbitrary length. This ensures that semantically related content stays together.
2. **Contextual Augmentation:** When related information is distributed across a document, we can **augment each chunk with surrounding context**. For the example above, before embedding the chunk containing "Table 2," we could prepend the document's title and the relevant section headings. This enriched chunk is more self-contained and provides the retriever with stronger signals, helping it find content that is both locally relevant and understandable in a broader context.

Ultimately, choosing and optimizing a chunking strategy is an empirical process. It requires us to systematically evaluate different methods—from a simple grid search over fixed-size chunks to more complex content-aware approaches—and measure their impact on retrieval metrics. By analyzing where our retriever fails, we can identify if our chunking strategy is the root cause and adjust it accordingly.

7.5 Evaluating Generation Quality

With a well-performing retrieval system in place, we can then evaluate the LLM's generation using the retrieved context. The ARES framework (Saad-Falcon et al. 2024) offers several dimensions for this:[64]

- **Answer Faithfulness:** Does the LLM output accurately reflect the information present in the retrieved context? This is crucial for avoiding hallucinations and ensuring the LLM grounds its response in the provided evidence. Specifically, we might look for:
  - *Hallucinations*: Information generated in the LLM output that is absent from the source documents.[65]

[64] RAGAS (Es et al. 2024) is another framework for automated RAG evaluation, that predates ARES.

[65] In RAG applications—particularly question answering—answers are expected to stay grounded in the retrieved context. Even if factually correct, additional information from the LLM's world knowledge is often undesirable and treated as a form of hallucination in practice.

  - *Omissions*: Information from the context ignored in the LLM output.
  - *Misinterpretations*: Information from the context represented inaccurately.

- **Answer Relevance:** Is the generated answer not only faithful to the context but also directly relevant to the original input query? An answer can be factually correct based on the context but still fail to address the user's actual question.

The generation failures listed above are precisely what we investigate during the error analysis detailed in Section 3. While these terms define broad categories, effective analysis requires us to uncover the specific, concrete ways *our* particular pipeline manifests these issues. For instance, error analysis should pinpoint not just that hallucinations occur, but specifically *what kind* of information is typically hallucinated or which types of constraints are frequently omitted.

7.6 Common Pitfalls

Evaluating RAG pipelines presents several common failure modes that we must account for to draw meaningful conclusions.

We often make the mistake of relying too heavily on end-to-end metrics. A single correctness score doesn't tell us whether retrieval or generation is to blame. To isolate failures, we need to measure each component independently—using Recall@k for retrieval and faithfulness for generation, for example.

We also tend to overfit to synthetic evaluation datasets. While synthetic data is useful for initial bootstrapping, it often fails to capture the ambiguity and diversity of real user queries. Many generated questions are either too easy or too tightly coupled to specific wording in source documents. Worse, the designated "gold" chunk may not be unique or even optimal. To avoid misleading results, we should regularly validate on real queries from logs or human-curated examples.[66]

Metric selection is another subtle source of error. If the LLM needs to synthesize multiple relevant chunks, Recall@k or NDCG@k is a better fit. Choosing the wrong metric leads us to optimize behaviors that don't actually improve final output.

We often ignore how chunking strategy shapes retrieval evaluation. Chunks that are too small may scatter relevant information across multiple fragments. Chunks that are too large may contain irrelevant noise. We can't assume a fixed chunking scheme—we need to treat it as a tunable part of the pipeline.[67]

We also see teams evaluate generation without checking whether the output is grounded in the retrieved context. Fluency and overlap with

[66] Filtering synthetic data helps, but doesn't fully substitute for real-world test sets. In our experience, synthetic queries tend to skew toward extractive answers and ignore multi-hop or judgment-based questions.

[67] Even with the same retriever, metrics like Recall@k can vary significantly based on chunking alone.

a reference answer aren't enough. A response that sounds correct but contradicts the evidence undermines the entire purpose of using retrieval. To evaluate RAG properly, we need context-sensitive metrics that measure whether the generation aligns with the retrieved content.

7.7 Summary

In this chapter, we covered evaluation strategies for RAG pipelines. We discussed how to construct datasets of queries and gold chunks, how to measure retrieval quality using metrics like Recall@k, MRR, and NDCG, and how to assess generation using context-sensitive metrics such as faithfulness and relevance. We also outlined common pitfalls in evaluating RAG systems, including over-reliance on end-to-end scores, misaligned synthetic datasets, and overlooked design choices like chunking strategy.

So far, we've focused on pipelines with relatively fixed structure: the system retrieves context, then generates a response. In the next chapter, we shift focus to more flexible architectures—those where LLMs operate in loops, construct intermediate reasoning steps, or issue tool calls dynamically. These agentic systems introduce new challenges for evaluation, especially around attribution, planning accuracy, and tool-use correctness.

7.8 Exercises

1. **Designing Synthetic QA Pair Prompts (Conceptual).**
   we have a corpus of movie metadata. Consider this example chunk from a movie entry:

   > "*Inception* is a 2010 science-fiction film written and directed by Christopher Nolan. The film stars Leonardo DiCaprio as Dom Cobb, a professional thief who steals information by infiltrating the subconscious."

   (a) Write a prompt for an LLM that:
   - Extracts a single, self-contained fact (e.g., "The film stars Leonardo DiCaprio as Dom Cobb") from the chunk.
   - Generates a question that can only be answered by that fact (e.g., "Which actor plays Dom Cobb in *Inception*?").
   - Returns exactly one JSON object with keys `"fact"` and `"question"`.

   (b) Modify our prompt so that the generated question reuses terminology from similar movie chunks—such as another entry mentioning "science-fiction film" or "2010"—but only this chunk contains the true answer. Show the revised prompt and explain how it prevents the answer from appearing in distractor chunks.

Solution 7.1

(a) Prompt Draft:

Sample Prompt

```
we are a helpful assistant generating synthetic QA
pairs for retrieval evaluation over a movie database.
Given a chunk of text describing a single movie, do the
following:
(a) Identify one fact that is self-contained (e.g., "The
    film stars Leonardo DiCaprio as Dom Cobb").
(b) Formulate a question that can only be answered by
    that fact.
(c) Return exactly one JSON object with keys "fact" and
    "question"--no extra keys or commentary.
Output format (exactly): { "fact": "...", "question":
"..." }
Chunk: "{movie_chunk_text }"
```

(b) Modified Prompt for Adversarial Questions:

Sample Prompt

we are a helpful assistant generating adversarial synthetic QA pairs for retrieval evaluation over a movie database.
Given:

- A target chunk (Chunk A) describing one movie.
- A set of other movie chunks (Chunks B, C, ...) that mention overlapping terms like "science-fiction film" or "2010" but do *not* contain the specific fact.

Do the following:

(a) Extract a single, self-contained fact from Chunk A (e.g., "The film stars Leonardo DiCaprio as Dom Cobb").

(b) Write a question that uses terminology from Chunks B, C, ... (for example, "science-fiction" or "2010") but can be answered only by the fact in Chunk A.

(c) Return exactly one JSON object with keys "fact" and "question"--no additional keys or text.

Output format (exactly): { "fact": "...", "question": "..." }
Chunk A (target): "{Inception is a 2010 science-fiction film written and directed by Christopher Nolan. The film stars Leonardo DiCaprio as Dom Cobb. }"
Similar chunks (B, C, ...):

- Chunk B: "*Interstellar* is a 2014 science-fiction film directed by Christopher Nolan."
- Chunk C: "*Avatar* is a 2009 science-fiction film directed by James Cameron."

Explanation: By providing "Similar chunks" that contain terms like "science-fiction film" or "Christopher Nolan," we force the LLM to borrow those terms in its question. However, since only Chunk A mentions "Leonardo DiCaprio as Dom Cobb," the question cannot be correctly answered by B or C. For instance, it might produce:

```
{
  "fact": "The film stars Leonardo DiCaprio as Dom Cobb",
  "question": "Which actor plays Dom Cobb in the 2010
  ↪ science-fiction film written and directed by
  ↪ Christopher Nolan?"
}
```

This phrasing uses "2010," "science-fiction," and "Christopher Nolan" from the distractor chunks, but only Chunk A contains

"Leonardo DiCaprio as Dom Cobb," making it adversarial.

2. **Filtering Synthetic Questions (Practical).**
   After generating synthetic questions, we obtain examples such as:

   - "Which actor plays Dom Cobb in *Inception*?"
   - "If I sum the runtimes of *Inception* and *Interstellar*, what is the total?"
   - "When was *The Dark Knight* released in theaters?"
   - "How many Oscars did the movie starring Leonardo DiCaprio as Dom Cobb win?"

   we plan to filter these using an LLM that rates each question's realism on a 1–5 scale (1 = Very Unrealistic, 5 = Very Realistic).

   (a) Create a few-shot prompt that shows at least two realistic movie-related questions (with ratings) and two unrealistic ones, then asks the LLM to rate a new synthetic question like:

   "If I combine the box office earnings of every Christopher Nolan–directed film, what is the sum?"

Solution 7.2

Few-Shot Prompt:

Sample Prompt

```
we are an AI assistant tasked with scoring
movie-related questions for realism on a scale from
1 to 5 (1 = Very Unrealistic, 5 = Very Realistic).
Provide a brief explanation for each rating.
Realistic Examples (Good Queries):
```

- *"Which actor plays Dom Cobb in* Inception*?"*
  Rating: 5
  Explanation: A straightforward fact-based question about a well-known film role.
- *"When was* The Dark Knight *released in theaters?"*
  Rating: 5
  Explanation: Common metadata question about a blockbuster movie release.

```
Unrealistic Examples (Bad Queries):
```

- *"What is the combined IMDb user rating of every Quentin Tarantino film released before 1990?"*
  Rating: 1
  Explanation: Too contrived--unlikely a user would ask for an aggregate rating across multiple films from different eras.
- *"Which movie has the longest sequence of consecutive close-up shots of an actor blinking?"*
  Rating: 2
  Explanation: Overly specific and not representative of typical information needs.

```
Task: For the following synthetic question, assign a
rating (1-5) and provide a brief explanation.
Question to Evaluate: "If I combine the box office
earnings of every Christopher Nolan-directed film, what
is the sum?"
```

3. **Multi-Hop Synthetic Evaluation Design (Advanced Synthesis).** Consider the following two chunks extracted from a Wikipedia-style movie corpus:

   Chunk 1: "*Inception* is a 2010 science-fiction film written and directed by Christopher Nolan."
   Chunk 2: "Leonardo DiCaprio stars as Dom Cobb in *Inception*."

   we want to generate a 2-hop question that first identifies the director by year and then asks which actor stars in the film.

   (a) Describe a procedure to prompt an LLM to produce:

   - An intermediate question for the first hop (e.g., "Who wrote and directed the 2010 science-fiction film *Inception*?"), and

- A final question combining both hops (e.g., "Which actor stars as Dom Cobb in the film written and directed by Christopher Nolan in 2010?").

(b) Explain how we would evaluate retrieval performance by measuring whether the retriever returns $\{\text{Chunk 1}, \text{Chunk 2}\}$ within the top k results for each multi-hop query. Specify the metrics and a diagnostic analysis method.

Solution 7.3

(a) Procedure for Prompting the LLM:

(a) **Prompt for Hop 1:**

Sample Prompt

```
we are a synthetic-QA generator for a movie corpus.
Given a chunk of text, extract one fact and write a
question that asks for that fact.  Return exactly:  {
"fact":  "...", "question":  "..." }
Chunk:  "Inception is a 2010 science-fiction film
written and directed by Christopher Nolan."
```

Expected LLM Output Example:

```
{
  "fact": "Christopher Nolan wrote and directed the
  ↪  2010 science-fiction film Inception",
  "question": "Who wrote and directed the 2010
  ↪  science-fiction film Inception?"
}
```

That yields the intermediate Hop 1 question.

(b) **Prompt for Final (2-Hop) Question:**

Sample Prompt

```
we are a synthetic-QA generator for a movie corpus.
Use two chunks and the intermediate entity from Hop 1
to create a 2-hop question.  Return exactly one JSON
object with keys "fact" and "question".
Chunk 1:  "Inception is a 2010 science-fiction film
written and directed by Christopher Nolan."
Chunk 2:  "Leonardo DiCaprio stars as Dom Cobb in
Inception."
Intermediate entity (from Hop 1 answer):
"Christopher Nolan"
Instructions:
 i. The final question should identify the actor who
    stars as Dom Cobb, given that the film was written
    and directed by Christopher Nolan in 2010.
ii. Ensure that answering requires retrieving both
    chunks in sequence.
```

Expected LLM Output Example:

```
{
  "fact": "Leonardo DiCaprio stars as Dom Cobb in
  ↪  Inception",
  "question": "Which actor stars as Dom Cobb in the
  ↪  film written and directed by Christopher Nolan
  ↪  in 2010?"
}
```

This 2-hop question forces retrieval of Chunk 1 (to find "Christopher Nolan in 2010") and then Chunk 2 (to find "Leonardo DiCaprio").

(b) Retrieval Evaluation Strategy:

(a) **Two-Hop Recall@k:**

- *Definition:* The fraction of 2-hop queries for which both ground-truth chunks $\{\text{Chunk 1}, \text{Chunk 2}\}$ appear within the top k retrieved results.
- *Computation:* For each multi-hop query q, let $\mathcal{R}(q)$ be the set of top-k chunk IDs returned by the retriever. Then:

$$\text{TwoHopRecall@}k \;=\; \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}\big(\{\text{Chunk1}_i, \text{Chunk2}_i\} \subseteq \mathcal{R}(q_i)\big), \tag{4}$$

where N is the total number of 2-hop queries.

(b) **Hop-Specific MRR:**

- *Hop 1 MRR:* Treat the intermediate query ("Who wrote and directed the 2010 science-fiction film *Inception*?") as a standalone retrieval task. Compute:

$$\text{MRR}_1 = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{\text{rank}(\text{Chunk1}_i)}. \tag{5}$$

- *Hop 2 MRR:* Given the intermediate entity "Christopher Nolan" (assumed known), treat "Which actor stars as Dom Cobb in the film written and directed by Christopher Nolan in 2010?" as a standalone retrieval task. Compute:

$$\text{MRR}_2 = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{\text{rank}(\text{Chunk2}_i)}. \tag{6}$$

(c) **Diagnostic Analysis:**

- *Error Attribution:* For each failed 2-hop query, classify the failure as:
  i. Hop 1 Miss: Chunk1 not retrieved at all.
  ii. Hop 2 Miss: Chunk1 retrieved but Chunk2 missing.
  iii. Rank-Out-of-Top-k: Both are retrieved but one is ranked beyond position k.
- *Case Studies:* Sample failed queries and inspect:
  - How similar the query phrasing is to the actual chunk text.
  - Whether vocabulary overlap (e.g., "2010," "Christopher Nolan") was sufficient.
  - If chunk boundaries cause information splitting (e.g., actor's name not present in chunk 2 for some variations).
- *Index Granularity Ablation:* Vary chunk size (e.g., combine multiple paragraphs vs. single-sentence chunks) to see if retrieval performance improves when both facts are co-located.
- *Prompt Sensitivity:* If using an LLM-based reranker for Hop 2, measure retrieval rates when the intermediate entity is perturbed (e.g., "Nolan" vs. "Christopher Nolan") to gauge robustness.

(d) **Summary Reporting:**

- Report TwoHopRecall@k, MRR_1, and MRR_2 on a held-out set of synthetic 2-hop queries.
- Provide a small confusion-style breakdown:

Error Type	Count	Fraction
Hop 1 Miss	n_1	n_1/N
Hop 2 Miss	n_2	n_2/N
Rank-Out-of-Top-k	n_3	n_3/N

- Discuss how a Hop 1 success rate of 90% and a conditional Hop 2 success of 80% implies TwoHopRecall@$k \approx 0.72$ (i.e., 0.9×0.8).

4. **Interpreting RAG Performance with NDCG@k (Math & Analysis).**
Two RAG systems, System A and System B, are evaluated on the same query. For this query, the Ideal Discounted Cumulative Gain at rank 3 (IDCG@3) is 5.5. Both systems retrieve 3 documents.

   - System A retrieves documents with relevance scores (rank 1 to 3): [3, 0, 2]
   - System B retrieves documents with relevance scores (rank 1 to 3): [2, 3, 0]

   (Relevance: 0=irrelevant, 1=somewhat relevant, 2=relevant, 3=highly relevant)

   (a) Calculate $DCG@3$ for both System A and System B. Show our work.

   (b) Calculate $NDCG@3$ for both systems.

   (c) Based on their $NDCG@3$ scores, which system performed better on this query and why? What specific characteristic of $NDCG@k$ does this comparison highlight regarding how RAG systems are evaluated (refer to Section 7)?

Solution 7.4

(a) **Calculating $DCG@3$:**
The formula is

$$DCG@k = \sum_{i=1}^{k} \frac{\text{rel}_i}{\log_2(i+1)}. \tag{7}$$

- **System A:**

$$DCG@3_A = \frac{3}{\log_2(2)} + \frac{0}{\log_2(3)} + \frac{2}{\log_2(4)} = 3 + 0 + 1 = 4.0. \tag{8}$$

- **System B:**
$$DCG@3_B = \frac{2}{\log_2(2)} + \frac{3}{\log_2(3)} + \frac{0}{\log_2(4)} \approx 2 + 1.8927 + 0 = 3.8927. \tag{9}$$

(b) **Calculating $NDCG@3$:**
Since $IDCG@3 = 5.5$,

$$NDCG@3_A = \frac{4.0}{5.5} \approx 0.727, \quad NDCG@3_B = \frac{3.8927}{5.5} \approx 0.707. \tag{10}$$

(c) **Analysis:**
System A's $NDCG@3 \approx 0.727$ is higher than System B's 0.707. This is because System A placed its most relevant document (score 3) first, whereas System B placed its highly relevant document at rank 2. The logarithmic discount factor ($\log_2(i + 1)$) penalizes later ranks, so $NDCG@k$ heavily rewards placing highly relevant items earlier—critical in RAG when context windows are limited (see Section 7).

5. **Diagnosing RAG Failures with ARES Metrics (Analysis).**
Imagine a RAG system designed to answer technical support questions. For a specific query, "How do I reset the foobar widget to factory settings?", evaluation using ARES-style metrics (Section 7) yields:

   - **Context Relevance:** High (The retrieved manual section clearly describes factory resets for several widgets, including the foobar).
   - **Answer Faithfulness:** High (The LLM's generated answer accurately quotes steps from the retrieved foobar widget section).
   - **Answer Relevance:** Low (The LLM's answer provides instructions for resetting the "bazqux" widget, which was also in the retrieved context, instead of the "foobar" widget).

   Based on this pattern of scores, what is the most likely failure mode in the RAG system's generation step for this specific query? Which component (retriever or generator) and which specific aspect of that component's task is most likely implicated? Justify our reasoning.

Solution 7.5

The pattern (High Context Relevance, High Answer Faithfulness, Low Answer Relevance) indicates a failure in the **generator's selection of the correct portion of context**.

- **Retriever is performing well:** High Context Relevance means the correct manual section for "foobar widget" was included.
- **Generator is not hallucinating:** High Answer Faithfulness

shows that the LLM faithfully quoted instructions—just for the wrong widget.

- **Core issue:** Low Answer Relevance implies the generator attended to or prioritized instructions for the "bazqux widget" despite the user query specifically asking about "foobar widget."

Implicated component: *Generator (LLM)*. **Specific aspect:** Its grounding mechanism—an inability to focus on the exact context span relevant to "foobar widget" when multiple widget instructions coexist. The LLM correctly reproduced retrieved content but failed to discriminate between "foobar" and "bazqux" instructions.

8 Specific Architectures and Data Modalities

LLM-powered pipelines in production rarely follow a single-turn prompt-response model. Instead, they span complex, multi-step architectures—from retrieval-augmented generation to agentic planning—and often operate over messy, multimodal inputs like PDFs, long documents, or images. This chapter introduces practical evaluation techniques for these architectures and modalities.

Many production systems follow recurring design patterns. We cover some key architectures: tool calling and agentic systems more broadly. For each, we focus on common failure points (i.e., what to look for in error analysis).

8.1 Tool Calling

Tool calling architectures allow LLMs to interact with external systems (Schick et al. 2023). These interactions can range from calling highly structured tools like specific API endpoints (e.g., `get_calendar_availability`), all the way to using open-ended tools that execute code generated dynamically by the LLM itself (e.g., `exec_code`). Moreover, tools can differ by potential impact (Patil et al. 2024). Read-only tools limit the consequences of errors. Tools that write data (e.g., updating a database, scheduling an event) pose greater risks if they malfunction or are used incorrectly.

The first step, even before evaluation, is to carefully **prompt engineer tool descriptions**. The names, descriptions, and parameter specifications we provide to the LLM heavily influence its ability to select and use tools correctly. So, clear, unambiguous instructions are important.

In the tool calling pattern, the LLM interprets a request, decides which tool to use (from a list of tools that we pre-define in our request to the LLM), generates arguments for it, and then processes the tool's output. This cycle might repeat or conclude with a final response. We detail potential failure modes along each step of the tool calling cycle below:

1. **Tool Selection:** First, we check if the LLM selected the correct tool for the user's task. This includes ensuring that our **tools provided to the LLM are sufficiently distinct**; if we define tools with overlapping functionalities or ambiguous names, the LLM may struggle to differentiate them, leading to incorrect selections. Potential failures include the LLM choosing an inappropriate tool for the stated intent (e.g., trying to use the `send_email` tool when asked Are there showings available this weekend?" instead of the correct `check_calendar tool`). Other failures involve hallucinating a tool that did not exist in our pre-defined list of tools, or failing to use a tool when one is clearly needed (like finding listings but not calling `send_email` when asked to "Find houses

and email Bob").

2. **Argument Generation:** Next, we verify the arguments the LLM generated for the selected tool. We need to ensure they are structurally valid *and* semantically correct. Common failure modes range from schema or type check violations (e.g., providing `"price_max": "700k"` as a string instead of the required integer 700000 for `query_listings`) to missing required arguments (like calling `send_email` without specifying a `recipient_address`). Other failures involve incorrect argument *values*, such as using the wrong date format ("May 10th") for a `check_calendar` tool which requires YYYY-MM-DD.[68] We can detect structural issues using automated schema validation, for example, with Pydantic (Colvin 2017). Identifying incorrect values may require reference data, rule-based checks, or specific custom validators that we can implement.[69]

3. **Execution Success:** We also track whether the tool call itself completed successfully and returned the expected kind of results. These problems can happen even if the generated arguments passed earlier validation checks. Consider a tool call like `query_listings(property_id="98765")`. It might pass schema checks if the ID format is correct. However, if "property98765" doesn't exist in the database, the query will return an empty result—a common silent failure. Moreover, the tool can also give us runtime errors; for example, a call to an external service like `search_web("latest mortgage rates")` could fail due to a temporary network connection issue or an API timeout. When architecting LLM pipelines, it is helpful for us to capture (and log) any error messages or tracebacks with tool failures, so error analysis (Section 3) is easier.

4. **Handling Tool Output:** After the tool executes successfully and returns its result, we assess if the LLM correctly interpreted the result. Failures here are specific to the LLM's processing, assuming the tool worked correctly. One failure mode is the LLM *misinterpreting* specific data points from a successful tool response; for example, `query_listings` returns details for a property including `"sqft": 1500`, but the LLM incorrectly states in its summary that the property is "over 2000 sq ft." Another common failure involves the LLM *ignoring important details* within a successful response; for instance, `query_listings` might return two properties, one with a clear agent note ,"structural issues reported," which the LLM omits when presenting the options.[70] A different failure mode is failing to integrate the tool's output logically into the conversation; for example, `check_calendar` returns available slots `["Saturday 2pm", "Sunday 4pm"]`, but the LLM then asks the user "When are you free?" instead of proposing the specific times discovered by the tool.

[68] Generated arguments must be checked for security vulnerabilities; for instance, passing unsanitized user input directly into a SQL filter argument could risk injection attacks (Pedro et al. 2023).

[69] This semantic validation is analogous to data validation in traditional machine learning (Polyzotis et al. 2019; Breck et al. 2017), where we ensure feature values fall within expected, developer-defined ranges or belong to predefined sets.

[70] Ignoring or omitting important details is very domain-specific and is typically only uncovered during careful error analysis.

Evaluating each stage separately is crucial because failures often cascade; for example, poor argument generation might cause execution errors or lead to misinterpretation even if the LLM handles correct output well. Pinpointing the exact stage of failure—selection, arguments, execution, or output handling—directly informs where interventions like prompt refinement or schema adjustments are needed.

8.2 Agentic Systems

Agentic systems empower LLMs to make sequences of decisions, often involving tool use and iterative reasoning (e.g., following patterns like ReAct (Yao et al. 2023) and tool calling), to accomplish complex goals. The first thing to consider is the **spectrum of agency** we intend for our system. In other words, we need to decide how much autonomy the agent should have.

- At one end of the spectrum, we might instruct the agent to have **more agency**, encouraging it to keep making tool calls, refining its plan, and pursuing sub-goals until it's confident it has fulfilled the user's request or reached a definitive answer. The prompt instruction might say something like: "Keep invoking tools and resolve issues on your own until you are confident in the final answer."
- At the opposite end, for systems with a strong human-in-the-loop component or in safety-critical applications, we might instruct our agent to have **less agency**. This means it should cede control, stop, or explicitly ask for human input if anything is unclear, if it encounters an unexpected situation, or if its confidence in the next step is low.

It is important to define where our product lies on this spectrum. If we don't clearly articulate how much agency our agent is supposed to have, we can't even tell what constitutes an error versus intended behavior. For example, an agent that stops and asks for clarification might be a failure if we designed it for high autonomy, but correct behavior if we designed it for cautious, human-guided operation.

Evaluation of agentic systems, therefore, must assess not only the correctness of individual steps (like tool selection or argument generation, as discussed in Section 8.1) but also the overall strategy and adherence to the designed level of agency. During error analysis (Section 3), we must ask:

- Did the agent make reasonable decisions at each step given the information it had?
- Did its sequence of actions (its "plan") logically lead towards the goal?
- Did it exhibit the appropriate level of initiative or caution based on our design?
- Did it get stuck in loops, give up prematurely, or over-extend beyond its capabilities or mandate?

Answering these questions often requires detailed multi-step trace analysis and debugging, which we discuss further in Section 8.3.

It's worth noting that as agentic systems evolve, we might encounter or implement specific architectural patterns or communication protocols, such as the Model Context Protocol (MCP),[71] designed to standardize interactions between the LLM, tools, and data sources. While such protocols can significantly improve the robustness of tool integration and the clarity of interaction flows by defining structured message formats (often JSON-based) for elements like tool calls and responses, they generally do not change the fundamental questions we ask during evaluation, nor the core processes we follow.[72] For instance, an agent using MCP to call a tool still needs to be evaluated on whether it selected the correct tool, generated valid arguments, and appropriately handled the tool's output. Similarly, the core principles for assessing the agent's decision-making quality and adherence to its designed agency, as outlined above, remain the same.

[71] See `https://modelcontextprotocol.io` for more details on this specific protocol.

[72] A benefit of standardized protocols from an evaluation standpoint is that they often lead to more structured and consistent logging. This can make the detailed trace analysis and debugging, which we discuss further in Section 8.3, more straightforward by providing clearer, more easily parsable data on each step of the agent's interaction with its tools and environment.

8.3 Debugging Multi-Step Pipelines

Debugging pipelines with multiple interdependent steps, such as those involving RAG, tool calling, or agentic reasoning, is hard. An error originating in an early state can easily cascade, causing downstream components to fail or behave unexpectedly. Moreover, the LLM's own reasoning or decision-making might falter mid-process, especially in dynamic agentic pipelines where an LLM decides what step to take, and thus the execution path is not fixed.[73] Efficiently improving agentic pipelines requires us to pinpoint the *root cause* of failures.

The core principle enabling effective debugging is *traceability*. When we record a trace, we should capture as much information as possible at each state of the agent's execution. This includes inputs, outputs, intermediate reasoning or "thoughts" generated by the LLM, any tool calls made along with their arguments and responses (e.g., `call_api(tool='query_listings', args={...})`), and the decisions reached.[74]

Manual trace inspection explains individual failures well. However, it doesn't scale to reveal systemic patterns. We need a systematic method to find the most common or impactful errors across many runs. This identifies where to focus error analysis (Section 3) and debugging for the largest performance improvements. First, we define discrete **states** representing significant stages in the pipeline's potential execution paths. Typically, each distinct tool call corresponds to one or more states (e.g., generating arguments for `query_listings` might be state `GenSQL`, executing it state `ExecSQL`). We can even subdivide complex tools into different states based on usage patterns (e.g., distinguishing between

[73] With agents, we often do not have a mental model of their behavior on user queries until we look at several traces. Understanding the desired spectrum of agency (Section 8.2) is key to interpreting these traces.

[74] Specialized observability platforms designed for LLM applications can be invaluable for capturing and exploring these detailed traces (LangSmith by LangChain, Inc.; Phoenix by Arize AI; Weights & Biases).

`WebSearch_For_Facts` and `WebSearch_For_Opinions` if using a single search tool). Importantly, internal LLM reasoning steps, especially in agentic systems using patterns like ReAct (Yao et al. 2023), should also be defined as distinct states (e.g., `Update_Plan`, `Reflect_on_Result`, `Decide_Next_Action`).

The attribution process starts by collecting a dataset of execution traces, each labeled with its end-to-end outcome (e.g., success or failure). For the traces marked as failures, we need to identify the *first state* in the sequence where something demonstrably went wrong. This identification can be done using automated component-level evaluators (like schema validators or faithfulness checkers) or through human annotation, guided by predefined failure criteria for each state (e.g., generated invalid JSON arguments, hallucinated a reasoning step inconsistent with the goal, chose an inappropriate tool).

Note to readers

Why focus on the first failure? Errors often cascade; a mistake in an early state can lead to seemingly unrelated failures later on. By attributing the failure to the first state where criteria were violated, we avoid incorrectly blaming downstream components that might have acted reasonably given faulty inputs.

Instead of simply counting where failures originate, we often gain more insight by analyzing the *transitions* between states that lead to failures. We present a method aggregating failure data into a **transition failure matrix**. In this matrix, rows represent the "From State" (the last successfully completed state) and columns represent the "In State" (the state *during which* the first failure occurred). The value in cell (i, j) counts the number of times a failure happened *within state* j immediately following a successful completion of state i.

Here is an example for our real estate agent assistant, using the states shown in Figure 10. Analyzing many failed traces yields counts for first failures occurring in the column state after successfully transitioning from the row state:

Figure 10 visualizes such a transition failure matrix for our example agent, using color intensity to highlight the frequency of first failures. We can immediately see hotspots where these initial errors concentrate. The most frequent transition leading to a first failure (12 instances, highest intensity) occurs when entering the `ExecSQL` state immediately after completing the `GenSQL` state. Other notable failure points appear at the `PlanCal` $\rightarrow$ `ExecCal` transition (7 failures) and the `DecideTool` $\rightarrow$ `GenSQL` transition (6 failures).

To identify the overall most frequently failing *state* (where first failures are detected, regardless of the preceding step), we sum the failure

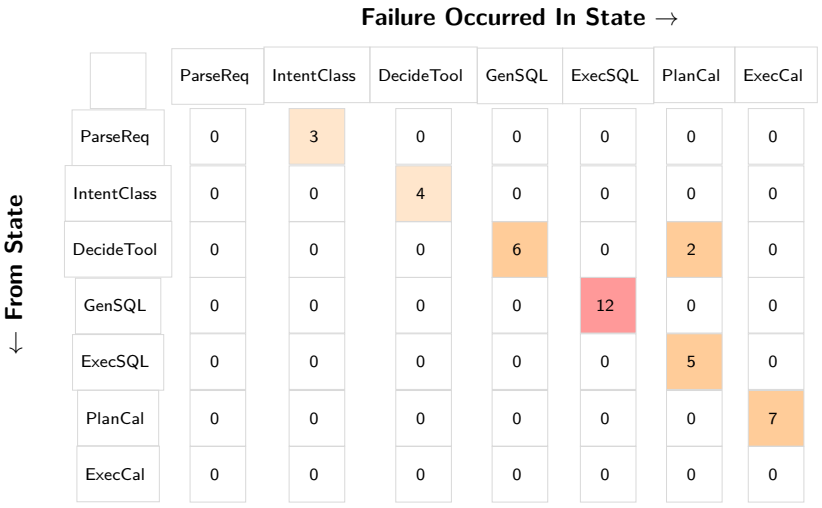


Figure 10: Transition-failure heatmap showing first-failure counts between pipeline states.

counts in each column. Based on Figure 10, `ExecSQL` has the highest total count of initial failures (12), making it the most common point of first failure overall, followed by `PlanCal` (5+2=7), `ExecCal` (7), and `GenSQL` (6). This column-sum perspective helps prioritize states that are broadly problematic.

When investigating a specific hotspot like the `GenSQL` → `ExecSQL` transition, our debugging starts by isolating those 12 specific traces. We analyze what went wrong *during* the `ExecSQL` state, critically examining the context provided by the preceding state when performing error analysis (Section 3).[75] Note that this pinpoints *where* to begin investigating, but doesn't automatically imply the preceding `GenSQL` state is the sole cause; the root issue might lie there, within `ExecSQL` itself (e.g., if our database uses a SQL dialect that we did not mention in the agent's prompt).

Observing the overall pattern of transitions (the distribution of non-zero cells across the matrix) can illuminate the agent's typical logic flow. For example, seeing multiple possible "To States" with failures originating from `DecideTool` highlights a key branching point. Is the agent mostly linear, or does it explore many paths? Indeed, we could create a similar heatmap tracking *all* state transitions (counting successes, not just first failures) simply to map and understand the agent's common operational patterns and identify frequently (or rarely) used pathways.

As we iteratively debug the pipeline using insights from this transition analysis, we expect the failure heatmap (Figure 10) to evolve. Fixes—to prompts, tool schemas, validation logic, or agent reasoning—should reduce the counts in the identified hotspots. Over time, the heatmap should become sparser or show lower overall intensity (fewer bright orange cells). Ideally, it converges towards a stable pattern reflecting the system's

[75] Why track the preceding "From State"? While summing columns identifies the overall most error-prone state, this alone can be misleading for targeted debugging. A state like `ExecSQL` might fail frequently after LLM-generated operations but succeed reliably after simpler preceding states (e.g., the user may have specified a SQL query). Tracking the "From State" allows a finer-grained analysis, helping us pinpoint the specific *transitions* or *contexts* that most often lead to failure, rather than just identifying a state that might be problematic only under certain conditions.

improved robustness, perhaps highlighting only residual, hard-to-fix errors or the inherent difficulty of certain specific transitions.

8.4 Evaluating Specific Input Data Modalities

Evaluating pipelines that process input types beyond simple or short text—like images, long documents, or structured PDFs—requires considering modality-specific failure modes. We briefly outline some error types to look out for, tailored to these common complex data types.

Images. Vision-Language Models (VLMs) process images, sometimes alongside text prompts, to generate textual descriptions, answer questions, or even edit images (Bordes et al. 2024; Webber and Olgiati 2023). They are generally effective at describing visual properties of individual objects—such as color, shape, and material (e.g., identifying a red car or a wooden table) (Li et al. 2022). Common issues to look out for in error analysis are: errors in spatial reasoning (misjudging positions like left/right, above/below, or failing to interpret complex spatial predicates such as "behind the chair to the right of the table") (Chen et al. 2024a), counting objects (Parcalabescu et al. 2022), recognizing text within the image (Nagaonkar et al. 2025), and hallucinating objects or details not actually present (Zhou et al. 2024b).

The core evaluation techniques we outlined earlier apply directly to VLMs. We rely on defining clear, preferably binary, evaluation criteria and detailed rubrics (Section 3) for consistent assessment. These guide human reviewers (Section 4) and can be adapted for LLM-as-judge prompts (Section 5.3). The key implementation difference is the need for visual grounding: human evaluators must reference the image, and any automated judge must itself be a capable VLM able to process visual input.

Evaluating image generation (text-to-image) raises different issues. Generally, we check if the image matches the prompt, its overall quality and coherence, and safety. However, image generation model architectures, e.g., diffusion models (Nakkiran et al. 2025), lend themselves to types of mistakes we would not expect. Unlike text models that generate token-by-token, many image generators start with noise and refine pixels across the entire image, in parallel. Because they don't typically build the image object-by-object, it can be hard for them to enforce "global" constraints like exact counts or distinct identities for multiple objects. For example, asking for "an image of 3 presidents" might produce three copies of the same person or figures that look like strange combinations of different presidents. A more structured prompt like "Obama, Trump, and Biden standing left to right" gives better guidance. Overall, accurately rendering complex compositions of concepts is difficult. Thus, evaluating these aspects—making sure objects are distinct, counts are right, layouts are

correct—often requires careful human review or transformer-based VLMs to serve as judges of correctness.

Long Documents. LLMs encounter difficulties with very long text inputs. While these stem from fixed context window limits, many empirical studies have found that task accuracy degrades as context grows, even if the text can theoretically fit within an LLM's context window (Hsieh et al. 2024; Tay et al. 2021; Liu et al. 2024b). Evaluating pipelines processing long documents depends heavily on the specific task. We consider two common workload types, informally distinguished by how much of the document needs attention:

Constant Output Tasks (sometimes called "Needle-in-a-Haystack"). Here, the LLM needs to find and use a relatively small, constant amount of information from within the long document. Essentially, the scope of information needed doesn't necessarily grow because the document is longer. Such tasks also need not focus solely on keyword retrieval; they might involve finding a few specific facts (e.g., "What were the names and start dates of the two project leads mentioned?"), identifying a concept (e.g., "Find the paragraph describing the main ethical concern"), or performing reasoning based on localized information (e.g., "Determine if the proposed budget exceeds $50k"). The output can still be generative, summarizing or reasoning about the found information.[76] Error analysis should focus on whether performance drops significantly when the needed information is "lost in the middle" of the context window (Liu et al. 2024b).

[76] Hsieh et al. (2024) propose some evals to assess capabilities beyond keyword retrieval, such as multi-hop tracing and aggregation of multiple discovered insights.

Variable-Size Output Tasks. These tasks produce outputs that grow with the document length—for example, extracting *all* names in a transcript, summarizing *every* section of a report, or identifying *each* angry comment in a script. Since the LLM can't process the full document at once, we break it into smaller chunks, process each chunk independently, then aggregate the results (Shankar et al. 2024a).

The challenge is to find the largest chunk size that still lets the model perform the task reliably. This depends not just on the model's context window, but on how hard the per-chunk task is. Simple tasks, like "extract all names mentioned in this chunk," can work well with longer chunks—e.g., 2k–4k tokens. But more complex tasks—like "extract and sort all names alphabetically" or "rewrite all angry comments in a polite tone"—often fail if the chunk is too large. These tasks involve reasoning, transformation, or tracking many internal elements (like tone, sort order, or labels), and the model's accuracy tends to degrade as the number of items increases.

A good rule of thumb is to keep each chunk focused on *3–5 ideas or spans of content* that the LLM needs to reason about or operate over.[77]

[77] There's no fixed threshold for when chunking becomes necessary, but it depends on the task type.

For "constant-output" queries—like retrieving a specific fact from a long document—you can often use large chunks, even 16k–32k tokens. But for "variable-output" queries—like "extract all angry comments from a movie script"—you'll need smaller chunks, often closer to 1k tokens. In these cases, the limiting factor isn't context window size, but reasoning load: if a chunk contains more than 3–5 reasoning units (e.g., angry utterances), the LLM may miss or misclassify them. As a rule of thumb, tune chunk size to match the query type and the number of concepts the LLM needs to process per chunk. Constant output tasks will tolerate longer chunks than variable-size output tasks. Evaluation involves both per-chunk checks (e.g., did the model extract or summarize correctly?) and whole-document checks (e.g., were all relevant items found, were any duplicated, was the final output consistent?). We may also need overlapping chunks or global summaries to help the model preserve context across chunk boundaries.

Evaluating long-document pipelines follows the same basic steps we've used throughout this reader: start with error analysis (Section 3), define metrics, and scale evaluation using methods like LLM-as-Judge (Section 5.3). But for long documents, LLM-as-Judge needs special handling. We usually can't feed the full document, which can be thousands of tokens, into the judge due to context limits, latency, and even accuracy degradation when the judge sees too much at once (Shankar et al. 2024a). Instead, one solution is to give the judge just the relevant portion of the source—for example, the paragraph a summary sentence came from, or the snippet where an entity was supposedly extracted. To support this, judge prompts and rubrics should be especially clear about what "correct" means, since the judge won't see the whole document.

Another approach is chunk-level evaluation. We could break the document into chunks, have the pipeline process each one, and then ask the LLM judge to evaluate the output chunk-by-chunk. These per-chunk judgments are then aggregated. The aggregation depends on the task: for variable-output tasks (like "extract all angry comments"), we usually need every chunk to be correct. For constant-output tasks (like "what is the project code name that this document discusses?"), it's enough if the correct answer shows up in just one chunk.

In short, effective evaluation of long-document pipelines comes down to: (1) identifying the task type (constant or variable output), (2) tuning our chunking strategy to match the task's complexity, and (3) adapting LLM-as-Judge by using constrained snippets of documents as few-shot examples and chunk-wise scoring instead of prompts that reference the full document.

PDF Documents. PDF pipelines begin by extracting text and structure from the file to prepare input for the LLM. This extraction step is often

the biggest source of errors. Common problems include incorrect text (especially from OCR on scanned documents), broken tables, and text that's read in the wrong order due to complex layouts. OCR tools, in particular, tend to struggle with tables (Lin et al. 2025).

Extraction quality depends heavily on the type of PDF. Digitally generated PDFs usually extract cleanly, but scanned documents often produce noisy, error-prone text. Even when extraction succeeds, how we format the result for the LLM matters. For example, presenting a table row-by-row can yield different results than formatting it column-by-column (Liu et al. 2024c, 2025).

To evaluate these pipelines, we need to separate extraction errors from LLM reasoning failures. That means inspecting the actual extracted content the LLM saw, not just the original PDF. Otherwise, we risk blaming the model for mistakes caused by bad input.

8.5 Common Pitfalls

When evaluating complex LLM pipelines, particularly those involving tool calling and agentic systems, several common pitfalls can lead to misleading conclusions or inefficient improvement cycles. Awareness of these can help us design more effective evaluation strategies.

Tool Calling Systems. A frequent oversight in evaluating tool-calling systems is focusing solely on whether a tool executes without errors, as detailed in Section 8.1. This neglects important failures, such as the LLM generating structurally valid but semantically incorrect arguments, or misinterpreting the tool's output even if the call itself succeeded. Another common issue is underestimating the impact of tool descriptions; unclear or ambiguous names, descriptions, or parameter specifications provided to the LLM directly hinder its ability to select and use tools effectively. We must treat tool definitions as a critical, iterable component of the system.

Agentic Systems. For agentic systems, a primary pitfall is the lack of clearly defined expectations regarding the agent's autonomy, as discussed in Section 8.2. Without a well-articulated "spectrum of agency," we cannot reliably distinguish between desired initiative and problematic overreach, or between cautious clarification and undesirable passivity. Furthermore, over-relying on final task outcomes can obscure critical inefficiencies or flawed reasoning in an agent's intermediate steps. While end-to-end success is vital, our analysis must also scrutinize the coherence and efficiency of the agent's plan and decision-making process, often through the trace analysis techniques described in Section 8.3.

General Pipeline Concerns. Across all multi-step LLM pipelines, insufficient traceability remains a pervasive challenge. Without comprehensive logging of inputs, outputs, intermediate LLM "thoughts," and tool interactions at each stage, pinpointing the root cause of failures becomes exceptionally difficult, as highlighted in our discussion on debugging (Section 8.3).[78] Effective debugging and iterative improvement hinge on our ability to reconstruct and analyze the agent's behavior. Relatedly, if we delay or omit the evaluation of individual pipeline components in isolation—for example, the quality of PDF text extraction (Section 8.4) before LLM reasoning, or a specific tool's reliability before integrating it into a complex agent—we may struggle to efficiently identify and address bottlenecks. Both component-level and end-to-end evaluations are necessary for a holistic understanding of pipeline performance.

[78] **Shreya's Note:** Log everything—all artifacts generated as part of the path to answering a query!

8.6 Summary

Evaluating complex LLM applications effectively requires strategies tailored to specific architectures and data types. In this section, we examined key considerations for common patterns. When evaluating tool calling architectures (Section 8.1), we check for failures in tool selection, argument generation (structure, values, safety), execution success, and the LLM's handling of the tool's output. For multi-turn conversations, evaluation expands to assess overall session success, user satisfaction, and consistency maintained across and potentially between conversations. We also detailed a systematic approach to debugging agentic pipelines using state transition failure analysis (Section 8.3).

We also discussed challenges posed by different input modalities, such as images, long documents, and PDFs. Robust evaluation, therefore, integrates these specific techniques with component checks, end-to-end task assessment, and systematic error analysis (Section 3). Choosing the right mix is essential for building reliable systems.

Next, we will turn to incorporating these evaluation practices into continuous integration and development workflows.

8.7 Exercises

1. **Analyzing Faulty Tool Argument Generation (Scenario-based Free Response).**
   An LLM-powered travel assistant is asked: "I want to book a flight for two adults from London to New York, sometime in the first week of June. I prefer an early morning departure." The LLM decides to use a tool: `search_flights(details)`. The arguments generated by the LLM for the 'details' parameter are: `{"departure_city": "London", "arrival_city": "NYC", "date_range": "June 1st - June 7th", "num_passengers": "2", "preferred_time": "any"}`.

The tool's expected schema for 'details' is: `{"departure_city": str, "arrival_city": str, "start_date": date(YYYY-MM-DD), "end_date": date(YYYY-MM-DD), "num_passengers": int, "preferred_departure_window": Optional[Tuple[int, int]]}` (e.g., (6, 10) for 6 AM to 10 AM).

Based on the information in Section 8.1, identify and classify at least three distinct argument generation failures in the LLM's output above. For each failure, explain why it's an error and suggest a specific improvement (to the tool's description provided to the LLM, the LLM's system prompt, or the validation logic) to help prevent or catch it.

Solution 8.1

Here are three distinct argument generation failures:

(a) **Failure 1: Incorrect Data Type and Format for Dates.**

- **Classification:** Schema violation (data type/format) and Semantic incorrectness (combined date range instead of separate start/end).
- **Error:** The LLM provided `"date_range": "June 1st - June 7th"` as a single string. The tool expects two separate date objects: `"start_date": date(YYYY-MM-DD)` and `"end_date": date(YYYY-MM-DD)`.
- **Improvement Suggestion:** The tool description provided to the LLM should be very explicit about the `start_date` and `end_date` parameters, specifying their individual YYYY-MM-DD format. For example: "`start_date`: The first date of the desired travel period in YYYY-MM-DD format. `end_date`: The last date of the desired travel period in YYYY-MM-DD format." Schema validation (e.g., Pydantic) should then enforce these separate fields and their date types.

(b) **Failure 2: Incorrect Data Type for `num_passengers`.**

- **Classification:** Schema violation (data type).
- **Error:** The LLM provided `"num_passengers": "2"` (a string) instead of the expected integer `2`.
- **Improvement Suggestion:** The tool description should specify `num_passengers` as an integer. Schema validation should strictly enforce the integer type for this parameter. The LLM's system prompt could also include a general instruction: "Ensure all numerical values are passed as integers or floats as appropriate, not strings."

(c) **Failure 3: Semantic Misinterpretation/Omission for `preferred_time`.**

- **Classification:** Incorrect argument value (semantic error) / Failure to utilize available information.
- **Error:** The user specified "I prefer an early morning departure." The LLM generated `"preferred_time": "any"`, effectively ignoring this preference. While the schema shows `preferred_departure_window` as optional, the LLM missed an opportunity to use it correctly based on user input. It also used an incorrect parameter name and structure.
- **Improvement Suggestion:** The tool description for `preferred_departure_window` should clearly explain how to represent time windows (e.g., "Optional. A tuple representing the preferred departure window in 24-hour format, e.g., (6, 10) for 6:00 AM to 10:00 AM. Corresponds to early morning (5-9), morning (9-12), etc."). The LLM's system prompt should encourage it to extract all relevant constraints from the user's query. If the user specifies a preference like "early morning," the LLM should be guided (perhaps by few-shot examples or more detailed instructions in its meta-prompt) to translate that into the expected structured format, e.g., `{"preferred_departure_window": [5, 9]}`.

2. **Agentic System Design: Impact of Agency on Behavior & Evaluation (Scenario-based Free Response).**
Consider an agent designed to help users troubleshoot complex software issues. The agent has access to tools like `run_diagnostic_scan()`, `search_knowledge_base(query)`, and `ask_clarifying_question_to_user(question)`. A user reports: "My SuperAnalyzer software crashes whenever I try to load a 'Type Z' project file. It worked fine yesterday."

   Describe how the agent's subsequent actions and our evaluation of those actions might differ if it were designed with:

   (a) High Agency (instructed to "resolve issues independently, making necessary tool calls and inferences until a solution or definitive diagnosis is reached").

   (b) Low Agency (instructed to "prioritize safety and clarity; if uncertain about the next best step or if user input is ambiguous, ask for clarification before proceeding with potentially complex actions").

   For each scenario (a and b), provide a plausible sequence of 2-3 agent actions and explain why those actions (and the evaluation of them)

align with that specific agency design, referring to principles in Section 8.2.

Solution 8.2

This scenario tests how the defined "spectrum of agency" (Section 8.2) influences an agent's behavior and how we, as evaluators, would judge that behavior.

(a) High Agency Agent: (Instructed to "resolve issues independently, making necessary tool calls and inferences until a solution or definitive diagnosis is reached")

- **Plausible Actions:**
  (a) **Agent thinks:** "User reports crash with 'Type Z' files, specific to today. Potential causes: recent software update, corrupted file, system change. Start with diagnostics."
  (b) **Agent calls:** `run_diagnostic_scan(software="SuperAnalyzer", last_known_good_state="yesterday")`
  (c) **Agent (if scan inconclusive) calls:** `search_knowledge_base(query="SuperAnalyzer 'Type Z' file crash after working previously")`
- **Alignment with High Agency & Evaluation:** This sequence demonstrates initiative. The agent forms hypotheses and independently uses tools to gather information (`run_diagnostic_scan`, `search_knowledge_base`) without immediately asking the user for more details.
  - **Evaluation Focus:** Evaluate whether its chosen tools were logical, its inferences reasonable, and whether it pursued a resolution path effectively. Success would be identifying the cause or a viable solution. Asking the user immediately might be marked as suboptimal if the agent could have inferred or discovered the needed information on its own.

(b) Low Agency Agent: (Instructed to "prioritize safety and clarity; if uncertain about the next best step or if user input is ambiguous, ask for clarification before proceeding with potentially complex actions")

- **Plausible Actions:**
  (a) **Agent thinks:** "User reports crash. 'Type Z' file is specific. 'Worked fine yesterday' is key. Need to understand what changed or if this specific file is an issue."

(b) **Agent calls:** `ask_clarifying_question_to_user(question="Thanks for reporting this. To clarify, did anything change on your system or with the SuperAnalyzer software between yesterday and today, like an update? Also, have you tried loading a different 'Type Z' project file, or a different file type, to see if the issue is specific to this one file?")`

(c) **Agent (based on user response) might then suggest:** `search_knowledge_base(query="SuperAnalyzer 'Type Z' file crash specific file")` or run a targeted diagnostic.

- **Alignment with Low Agency & Evaluation:** This sequence prioritizes gathering more information before taking complex actions.
  - **Evaluation Focus:** Evaluate the relevance and helpfulness of the clarifying questions. Success means efficiently gathering necessary information to proceed safely. Running diagnostics without clarification would be a failure for a low-agency agent.

In summary, the same user query leads to different "correct" behaviors based on agency level. Evaluation must be benchmarked against these intended agency instructions.

3. **Evaluating a Complex PDF and Long Document Processing Pipeline (Scenario-based Free Response).**
   We are tasked with building and evaluating an LLM pipeline. Its goal is to answer user questions based on a collection of 500-page scanned PDF research papers. These PDFs contain dense text, complex tables presenting experimental results, and diagrams with captions. The users' questions might require synthesizing information from multiple sections of a paper, interpreting table data, or understanding diagram captions.

   Based on the challenges discussed in Section 8.4 for "Long Documents" and "PDF Documents," outline at least **three distinct major challenges** our evaluation strategy for this system would need to address. For each challenge, propose **one specific evaluation check, metric, or process** we would implement to assess the pipeline's performance regarding that challenge.

Solution 8.3

This complex pipeline faces significant challenges due to its input modalities (scanned PDFs, long documents) and the tasks required (synthesis, table/diagram interpretation). Here are three major challenges for the evaluation strategy and corresponding checks:

(a) **Challenge 1: Accuracy of Information Extraction from Scanned PDFs (especially Tables and Captions).**
As noted in Section 8.4, scanned PDFs require OCR, and OCR tools often struggle with tables and complex layouts. Diagram captions might also be misplaced or misread. Errors in this initial extraction phase will inevitably lead to incorrect answers, even if the LLM's reasoning is sound based on the faulty input.

- **Specific Evaluation Check/Process:** We would implement **Component-Level Evaluation of Extracted Content**. This involves:
  i. Creating a "gold set" of extracted content for a representative sample of PDF pages, including critical tables and diagrams with their captions. This gold set would be manually transcribed or corrected.
  ii. Comparing the pipeline's automated extraction output for these sample pages against the gold set.
  iii. Metrics could include Character Error Rate (CER) for text, Table Structure Accuracy (e.g., F1 score on correctly identified cells and their contents), and Caption Association Accuracy (was the correct caption linked to the correct diagram/table?).

  This isolates extraction errors from downstream LLM processing errors, helping us pinpoint where fixes are needed (e.g., improving OCR, using specialized table extraction models, or better layout analysis).

(b) **Challenge 2: Effective Information Retrieval and Synthesis from Long, Chunked Documents.**
Given 500-page documents, a chunking strategy is inevitable (Section 8.4). Users' questions requiring synthesis from multiple sections mean the system must not only find relevant chunks but also combine information across them. The "lost in the middle" issue can also affect per-chunk processing if chunks are still very large, or the synthesis step if too many chunks are involved.

- **Specific Evaluation Check/Process:** We would design **Multi-Hop Question-Answering Tasks with Traceability Analysis**.
  i. Create test questions that explicitly require information from 2-3 distinct, non-contiguous sections of a sample document.
  ii. For each test question, manually identify the source paragraphs/chunks in the original PDF that contain the necessary information.
  iii. When evaluating the pipeline's answer, we would not only check the answer's correctness but also inspect the pipeline's trace (if available) to see: *(i)* Were all necessary source chunks retrieved (Recall@k for chunks)? *(ii)* Did the LLM's generation step demonstrably use information from all these necessary retrieved chunks? (This might require manual inspection or a sophisticated LLM-as-judge focusing on attribution).

  This helps evaluate both the retrieval over long documents and the LLM's ability to synthesize information across the retrieved chunks for variable-size output tasks (where the answer scales with information spread).

(c) **Challenge 3: LLM's Ability to Interpret and Reason Over Extracted Structured Data (Tables) and Semi-Structured Data (Diagram Captions).**
Simply extracting table text isn't enough; the LLM must understand its structure (rows, columns, headers) to answer questions like "What was the p-value for Group B in Experiment 3?". Similarly, it must correctly associate captions with diagrams and reason about their content. The representation of this extracted data to the LLM is also key (Section 8.4).

- **Specific Evaluation Check/Process:** We would develop **Targeted Question Sets for Data Interpretation and Corresponding Rubrics**.
  i. For a sample of documents, create questions specifically targeting information only found in tables (e.g., requiring lookup, comparison, or simple calculation based on table cells) and questions that require understanding diagram captions in context of the diagram's visual implication (even if the VLM part is separate, the LLM processes the caption).

ii. Evaluate the LLM's answers against these questions using a detailed rubric that assesses not just correctness but also evidence of correct data interpretation (e.g., did it refer to the correct cells/caption concepts?).

iii. As an advanced step, if the intermediate representation of the table/caption fed to the LLM is available, error analysis would involve checking if that representation was conducive to correct interpretation (e.g., was the table linearized effectively for the LLM?).

This directly tests the LLM's reasoning capability on non-prose structured/semi-structured information extracted from the PDFs.

9 Continuous Integration and Deployment

Building an LLM pipeline that performs well initially is a significant achievement. However, maintaining and improving that performance over time, in the face of evolving user needs, data drift, and foundation model changes, requires a *continuous* approach to evaluation. This chapter bridges the gap from developing specific evaluators—the Measure phase (Section 5)—to embedding those measurements into an ongoing engineering lifecycle that drives the Improve phase (Figure 2).

We'll frame this operationalization using familiar concepts from software engineering: **Continuous Integration (CI)** and **Continuous Deployment/Delivery (CD).** However, as we'll explore, applying CI/CD to LLM pipelines is not a simple lift-and-shift from traditional software or even earlier Machine Learning Operations (MLOps). LLM systems present unique challenges: inherent non-determinism, the often subjective nature of "quality," the scarcity of labeled data for highly specific failure modes, and the "black box" nature of frequent foundation model updates by providers. Table 5 describes many of the differences between traditional MLOps and today's LLMOps.

A helpful framing for this new landscape is the split between ***"unit tests for known unknowns"*** (the primary role of CI) and ***"online monitoring for unknown unknowns"*** (the core of CD in the LLM context). While CI helps us guard against regressions on failure modes we've already identified, it's in the dynamic environment of production that many new, unexpected issues will surface. It's simply not feasible to anticipate and catch all, or even most, failure modes before an LLM application is in the wild.

This evolving landscape also suggests the emergence of new, specialized roles within engineering teams—perhaps an "AI-Native" QA Specialist. Such a role would be responsible for continuously tracking performance, designing and maintaining evaluation suites, analyzing unstructured feedback, and coordinating the iterative improvement of the LLM pipeline. This requires a blend of data analysis acumen, a deep understanding of LLM behaviors, and strong product intuition. This chapter aims to provide a foundational guide for engineers, PMs, and those stepping into such emerging evaluation-focused roles.

9.1 CI: Building a Safety Net Against Regressions

Continuous Integration (CI) in software engineering aims to verify that code changes integrate correctly and don't introduce regressions. This goal applies equally to LLM pipelines. Even traditional machine learning has CI practices. However, while ML CI might measure overall accuracy against large test sets, LLM CI focuses narrowly on preventing regressions

Aspect	Traditional MLOps	LLM Pipelines
Data Landscape & Core Logic	Begins with large, labeled datasets; CI/CD focuses on retraining on new data, monitoring feature drift, and validating against substantial test sets. (Sculley et al. 2015; Shankar et al. 2024b; Amershi et al. 2019)	Often begins with very few labeled examples generated during error analysis to characterize narrow failure modes. "Golden" CI datasets are small, curated collections of known good and bad examples. The core logic resides in the prompt and the pre-trained foundation model rather than in application-specific supervised training.
Sources & Nature of Failures	Failures include data drift, concept drift, bugs in feature engineering pipelines, and generally, closing the gap between training and inference environments. (Sculley et al. 2015; Breck et al. 2017; Hohman et al. 2024)	New failure categories arise: extreme sensitivity to prompt phrasing; unexpected regressions when providers update foundation models; misalignment or drift in LLM-as-Judge evaluators; persistent hallucination tendencies; inherent non-determinism affecting reproducibility; and complex failures in multi-step reasoning or agentic behaviors. Debugging involves tracing interdependencies across prompt, retrieval, and generation steps (Epperson et al. 2025; Lauro et al. 2025) rather than inspecting feature importance or model weights.
Performance Measurement (Especially Online)	Uses standard metrics (e.g., accuracy, F1-score, AUC). If online labels (e.g., user clicks) are available, calculating performance is straightforward via established formulas.	Quality is multifaceted and often subjective (e.g., tone, faithfulness, relevance, helpfulness) (Ouyang et al. 2022). Requires defining and implementing custom evaluators—often via complex code-based logic or LLM-as-Judge systems. Offline "golden" datasets tend to be small and may not represent production traffic, so online monitoring with these custom evaluators on live data is essential to gauge true performance and uncover new failure modes.
Non-Determinism & Testing	Model outputs are deterministic given a fixed version, enabling consistent CI tests and straightforward tracking of changes.	Identical inputs can yield different outputs (especially if temperature > 0). CI tests must tolerate variation (e.g., running tests multiple times or accepting a set of valid outputs). In CD, retries can serve as a recovery mechanism for transient online failures.

Table 5: Key aspects contrasting traditional MLOps and LLM-specific CI/CD pipelines.

for known failure modes using curated (smaller) golden datasets (**??**). It does not reliably predict overall production accuracy. Its purpose is only to ensuring stability as the pipeline evolves.

The foundation of LLM CI is a **golden dataset:** a hand-curated set of input examples with corresponding reference outputs. Creating this dataset requires human review and labeling (Section 3), often using collaborative evaluation methods (Section 4) to define correctness. The effort is justified because this test suite runs on every proposed pipeline change. On each proposed change (e.g., prompt tweak, model swap), we run all golden inputs through the pipeline and evaluate the outputs with a suite of automated evaluators (Section 5).[79] A well-constructed golden dataset should include:

- Examples covering the pipeline's core features and capabilities.
- Representative instances of critical failures observed in the past (directly acting as regression tests).
- Challenging edge cases uncovered during error analysis.

[79] This could happen on every commit to a feature branch or before merging to main. When using LLM-as-Judge evaluators, we need to pin the exact model version used for the judge in CI (e.g., `gpt-4o-2024-05-13`) to prevent CI results from fluctuating due to unannounced updates to the judge model itself.

- Examples designed to exercise different components or paths within the pipeline (e.g., specific tool calls, RAG scenarios, different prompt branches).

Size and Scope. A golden dataset typically includes 100+ examples representing core functionality, previously observed bugs, and known edge cases. It is not a random sample of production data. Instead, it's purpose-built to stress-test behaviors we want to preserve.

Reference-Based Evaluation. CI evaluations are reference-based: for each golden input, we check whether the new pipeline output aligns with the gold reference. As we discover new failure modes—e.g., via monitoring or error analysis—we expand the dataset accordingly.

We illustrate these principles with CI checks from a real estate assistant pipeline (Example 2) in Table 6. If any of these checks fail on a corresponding golden example, the CI build should fail, preventing the potentially regressive code change from being merged.[80]

[80] In some codebases and applications, LLM tests can be marked flaky, to re-run a few more times (Fernandez 2024).

Maintaining the golden dataset is an ongoing process. As new failure modes or critical edge cases are discovered (often through online monitoring, as discussed in Section 9.2), they must be added to the golden set. This ensures the CI "safety net" expands over time to cover newly understood risks.

Check	Input	Reference	CI Evals
Tone Match (LLM Judge)	Agent prompt and client persona	Human-verified, persona-aligned output	LLM judge verifies tone matches expected persona behavior
Tool Argument Schema Validation	Query to fetch calendar availability	Tool schema definition	Code-based validator ensures arguments conform to schema
Email Address Existence Check	Query instructing an email to a client	Known client email database	Code check confirms recipient address exists

Table 6: Examples of CI checks for a real estate assistant agent.

People make several common mistakes when interpreting CI results from golden datasets, leading to a false sense of security or misguided development efforts. First, a fundamental error is assuming that accuracy measured offline—whether on the entire CI set or a specific "hold-out" partition (i.e., not seen by the pipeline developer)—accurately reflects true production performance. This assumption is flawed because golden datasets are typically small (often < 100 examples), curated based on known issues rather than representative sampling, and thus offer weak statistical guarantees about behavior on diverse, live traffic. Consequently, the CI pass rate is an unreliable predictor of real-world quality.

Second, a critical error that invalidates the integrity of these checks is

data leakage: directly including any traces from the golden set within the main pipeline's prompts (e.g., as few-shot examples) or the LLM judge's prompts constitutes **overfitting**. This practice artificially inflates performance on those specific known examples, rendering the checks potentially meaningless. Therefore, the CI pass rate—assuming no data leakage has occurred—must be interpreted *only* as a regression indicator confirming stability against known issues, not as a reliable measure of overall generalizability.

9.2 CD & Online Monitoring: Tracking Real-World Performance

While CI prevents regressions on known issues, CD focuses on tracking system behavior in production. This is where we uncover true performance, surface new failure modes, and detect the "unknown unknowns."

Observability: The Foundation of Monitoring. Online evaluation starts with detailed instrumentation. Without comprehensive logging, we can't evaluate or debug effectively. For each production request (or a representative sample), we log:

- The initial input and relevant metadata (e.g., user/session IDs, flags)
- All intermediate LLM calls: prompts, responses, and intermediate reasoning
- All tool calls: names, arguments, responses, and errors
- Retrieved documents in RAG pipelines[81]
- The final output shown to the user
- Any user feedback (thumbs up/down, edits, follow-ups)
- Evaluator outputs (if applicable)

These logs allow us to reconstruct full user interactions. Tools like LangSmith, Arize Phoenix, and W&B Weave, or integrations with OpenTelemetry, help manage and explore this data, organized into traces, spans, and sessions. A **trace** encompasses the entire end-to-end processing of *a single* user request or workflow (e.g., handling one query to the real estate chatbot assistant, from initial parsing to final response). A trace, in turn, is composed of multiple **spans**, where each span represents a discrete computational step or operation within that workflow.[82] Lastly, a **session** typically groups a sequence of related traces over time, providing broader context—for instance, tracking an entire multi-turn conversation between a user and the chatbot as they discuss property requirements and refine search results.

Running Automated Evaluators in Production. With observability in place, we deploy automated evaluators (Section 5) to run on sampled production traces. A typical pipeline uses 5–7 key evaluators (a mix of code-based and LLM-as-Judge), each tracking a known failure mode.

[81] For debugging, it may be beneficial to also return more documents that the retriever *could have* retrieved for the LLM (but did not make it into the LLM prompt).

[82] Examples of spans in an LLM pipeline include individual LLM calls (like generating SQL or summarizing results), function calls to external tools (like executing `query_listings` or calling `send_email`), database interactions, or even specific internal logic blocks.

Due to latency and cost—especially for LLM-as-Judge—we often evaluate a small, stratified sample (e.g., 1–5%) asynchronously after the request completes.

To monitor each failure mode, we compute its corrected success rate $\hat{\theta}$ and a 95% confidence interval using bootstrapping (Section 5.3). This adjusts raw predictions using the evaluator's known TPR and TNR, giving us a statistically grounded estimate of real-world success rates—even without labels.

Example: Online Monitoring

Here is an example workflow for the team building the real estate agent assistant (Example 2):

1. The team samples 1% of production traces daily.
2. They run their "Hallucinated Tool Usage" and "Location Ambiguity" LLM judges on these sampled traces.
3. Using the pre-calculated TPR and TNR for each judge and bootstrapping (from Section 5.3), they compute the 95% confidence interval for the true success rate θ for each of "hallucination" and "ambiguity issues."
4. These θs and their confidence intervals are dashboarded. An alert is triggered if the lower bound of either confidence interval (or only θ) exceeds some acceptable threshold.

Guardrails: Synchronous Online Evaluation. Some evaluators act as **guardrails** and run synchronously during pipeline execution—while the system is actively processing a user request and before producing a final output. These guardrails must be both fast and reliable. They are typically lightweight, code-based checks such as regex filters, schema validation, or checking against a blacklist, and they must maintain a very *low false failure rate* to avoid interrupting valid user experiences.

Guardrails are typically used for critical safety checks (e.g., detecting PII, toxicity, harmful content), validating essential constraints (e.g., checking if generated SQL is syntactically valid before execution), or enforcing strict output formats. When a guardrail fails, the pipeline needs a predefined action to follow:

- *Reject:* Block the problematic output entirely. Return a safe, canned response or an error message to the user.
- Retry: Automatically re-run the LLM pipeline. Given the non-deterministic nature of LLMs, a retry might produce a compliant output. This is useful for transient generation glitches, but should be used judiciously.

- Fallback: Switch to an alternative, potentially simpler or constrained model or logic path to fulfill the user request.

Going back to the real estate agent assistant, a guardrail could run a fast regex check on generated emails before sending. For example, if it detects a pattern resembling a social security number, it triggers "Reject." Another guardrail could checks if generated SQL is syntactically valid, using a parser. If invalid, it triggers "Retry" once. If the retry also fails, it might "Fallback" to asking the agent user to rephrase their request.

Judge Drift: Maintaining LLM-as-Judge Accuracy. LLM-as-Judge evaluators can drift over time. This drift may result from foundation model updates or changes in how the team defines a subjective quality criterion. To maintain alignment, we recommend the following:

- **Pin model versions.** Always specify the exact foundation model version (e.g., `claude-3-opus-20240229`) in both CI and production configurations.
- **Implement a re-alignment loop.** Judge alignment is not a one-time task. Periodically revalidate each judge to ensure it remains accurate.
  - **Frequency:** Sample recent production traces every few weeks—more frequently for subjective criteria.
  - **Labeling:** Obtain new human labels for these traces using the current evaluation rubric.
  - **Metrics:** Recompute the judge's true positive rate (TPR) and true negative rate (TNR) against the fresh labels.
  - **Trigger:** If TPR or TNR falls below an acceptable threshold, update the judge's prompt or few-shot examples (Section 5.3), revalidate it on a hold-out set, and redeploy.
- **Re-evaluate after model changes.** Repeat the process of re-aligning the judge any time the judge's underlying LLM is changed.

9.3 The Continuous Improvement Flywheel

CI and CD are not isolated stages; they are integral components of a continuous improvement flywheel that powers the iterative refinement and sustained quality of our LLM application. This cycle explicitly connects monitoring and evaluation back to the *Improve* phase of our Analyze-Measure-Improve lifecycle. We present a flywheel as follows:

Develop & Analyze (Initial): Start with initial pipeline development (prompts, RAG setup, tool integrations). Conduct thorough Error Analysis (Section 3) on early outputs or bootstrapped data to identify key failure modes and understand the data.

Measure & Build Evals: Translate these qualitative failure modes into quantifiable metrics. Implement and validate a suite of automated evaluators (code-based and LLM-as-Judge), as detailed in Section 5. Use collaborative methods (Section 4) if rubrics are subjective, and build the initial golden dataset.

CI Setup: Integrate the golden dataset and automated evaluators into the CI pipeline to act as regression tests for known issues.

Deploy (CD) with Observability: Ship the LLM pipeline instrumented for comprehensive logging. Deploy automated evaluators to run on sampled production traffic (often asynchronously). Ensure LLM-as-Judge models are pinned.

Monitor Online Performance: Actively track the corrected success rates ($\hat{\theta}$) and confidence intervals for key failure modes using live production data. Dashboard these metrics and set up alerts for significant deviations or threshold breaches. Concurrently, track relevant product metrics (e.g., user satisfaction scores, task completion rates, session length, thumbs up/down).

Identify Drift, New Failures, or Product Issues. Once a pipeline is deployed and monitored in production, the next task is to interpret observed changes. Start by analyzing online evaluation data. A drop in corrected success rates for specific evaluators might indicate drift in the underlying model or retriever behavior, or subtle regressions introduced by new features. Evaluator outputs, especially when sampled over time, can help flag traces that exhibit repeated failure patterns. In parallel, product metrics such as task completion rate or session duration may dip, which can signal a quality issue on the AI side—or reveal a UX problem unrelated to the model. Distinguishing the two is challenging but necessary. To complement this, teams should proactively sample and manually inspect traces, especially those not flagged by automated evaluators. This "failure hunting" helps uncover novel or subtle issues—what might otherwise remain unknown unknowns. Section 10 describes lightweight ways to enable this human-in-the-loop monitoring.

Example: Location Ambiguity Detection in Deployment

Monitoring for the Real Estate Agent Assistant reveals a spike in the "Location Ambiguity" success rate after a new neighborhood-targeting feature is deployed. Product metrics show users in those neighborhoods have shorter sessions. Manual review of flagged traces shows the LLM frequently confuses "West Berkeley" with "Berkeley West" (a non-existent area), leading to poor search results and user frustration.

Re-Analyze (Error Analysis): When a new significant issue or drift is confirmed (via automated evals, product metrics, or human review), conduct targeted Error Analysis (Section 3) on the problematic traces.

Note to readers

When do you re-trigger error analysis, as described in Section 3? Some ideas:

- After substantial product changes (e.g., new agent capabilities like DB querying, different feature goals).
- If user demographics or query distributions shift significantly.
- If online evaluators show persistent new failure patterns or product metrics indicate an AI problem.

The goal here is to understand the root cause of new failures and assess whether the current failure mode taxonomy (and evaluators) remains relevant. For instance, if the "West Berkeley" issue is systemic, "Handling of Directional Location Qualifiers" might become a new, specific failure mode to track.

Update Evaluation Artifacts. Once a new failure pattern has been confirmed, update the relevant evaluation artifacts to ensure that future regressions can be caught. The CI **golden dataset** should be augmented with examples that exhibit this failure, ensuring that any changes that reintroduce it will fail CI. Similarly, existing evaluators should be refined to better capture the updated definitions of correctness, or new ones should be implemented if needed. This may require adjusting rubrics used by LLM-as-Judge systems or extending logic in code-based evaluators. Because judges themselves may drift, it's important to re-align them periodically. For subjective evaluators, revalidating true positive and true negative rates against newly labeled traces every few weeks is a good default. If a judge's TPR/TNR drops below acceptable thresholds, update its prompt or few-shot examples and validate the new version before redeployment. Finally, make sure that CI checks are updated to incorporate

tests for any new failure modes, using the newly curated golden examples.

Improve Pipeline: Implement changes to the LLM pipeline based on insights from error analysis and online monitoring. Refer to Section 11 for practical strategies to improve accuracy and cost.

Re-deploy & Iterate: Ship the improved pipeline and updated evaluation artifacts. Resume monitoring, expecting the targeted success rates to increase. The flywheel continues.

This flywheel emphasizes that evaluation is not a one-time phase but an ongoing, iterative process. Discoveries during online monitoring feed back into error analysis, refinement of evaluation artifacts, and ultimately, pipeline improvements.

9.4 Practical Considerations and Common Pitfalls for Production LLM Evaluation

Successfully operationalizing LLM evaluation requires navigating several practical challenges and avoiding common pitfalls.

Tooling for Observability, Annotation, and Sensemaking: While we should avoid prematurely investing in overly complex "evals platforms" before understanding our actual needs, some foundational tooling is essential:

- **Logging & Trace Viewing:** A system to capture and inspect detailed traces (inputs, intermediate steps, tool calls, outputs) is non-negotiable (Section 9.2).
- **Simple Annotation Interfaces:** For human review and labeling, we need interfaces that are more efficient than spreadsheets. These can often be lightweight, custom-built UIs (see Section 10) that allow for quick tagging of failure modes and annotation. The goal is to support rapid sensemaking from trace data.

Common Pitfalls to Avoid:

- **Stagnant CI Golden Datasets:** Production data drifts, user behaviors change, and new edge cases emerge. If our CI golden dataset is not continuously updated with examples of these new phenomena (derived from online monitoring and error analysis), the CI suite will become progressively less effective at catching relevant regressions.
- **Superficial Curation of Golden Datasets:** A large but unrepresentative golden set gives a false sense of security.[83] We should focus on quality and diversity—include examples that stress test core features, past failures, challenging variations of user queries, and distinct pipeline components.

[83] **Shreya's Note:** I see people have really large datasets that they haven't taken the time to curate well. If you don't know what is in your dataset in CI, any metrics on it will be meaningless.

- **Over-reliance on Automated Monitoring Alone:** Even well-aligned LLM-as-Judge evaluators may miss subtle or novel failure modes that require human intuition or domain expertise. We should regularly conduct "failure hunting" sessions where humans review random samples of production traces, not just those flagged by automated systems.[84] This is critical for staying ahead of drift.
- **Ignoring or Mishandling Conflicting Metrics:** Improving one quality dimension (e.g., conciseness) can inadvertently degrade another (e.g., factual accuracy). We must define our key quality metrics, prioritize them (especially non-negotiable safety guardrails), and establish a strategy for making trade-offs when metrics conflict.
- **Not Connecting AI Metrics to Product Metrics:** When user satisfaction or task completion rates drop, we should investigate whether AI quality is a contributing factor.[85] If an AI issue is confirmed (e.g., excessive verbosity causing user drop-off), we must define a corresponding failure mode and evaluator to track it. This ensures alignment between product health and AI performance.
- **Delayed or Absent Re-evaluation of LLM-as-Judge Alignment:** The true positive/negative rates (TPR/TNR) of LLM judges are not static. We should regularly recalculate them against fresh human labels, particularly for subjective criteria or when the judge model changes. Outdated TPR/TNR values will result in inaccurate online success rate estimates ($\hat{\theta}$).
- **Insufficient Traceability:** If we cannot reconstruct what happened during a pipeline execution (e.g., all LLM calls, tool interactions, retrieved context), debugging becomes guesswork. Comprehensive logging is essential.

[84] Product metrics can be an early signal of emerging failure modes. For example, if user engagement drops, such as a decline in the average number of interactions with the assistant, but automated evaluator metrics remain unchanged, it suggests the presence of undetected issues. This warrants targeted error analysis to uncover and characterize new failure modes.

[85] Conversely, product metrics may decline even when AI quality metrics remain strong. This can occur if the product is poorly designed, misaligned with user workflows, or addressing a problem users don't actually have. Evaluation must distinguish between model failures and broader product fit issues.

9.5 Summary

Operationalizing LLM evaluation requires adapting CI/CD to the specifics of LLM systems. CI prevents regressions by running automated checks—using golden datasets and both code-based and pinned LLM-judge evaluators—against known failure modes. Its main purpose is stability, not measuring real-world performance.

CD measures real performance through online monitoring. This includes running evaluators on sampled production traffic, tracking statistically corrected success rates ($\hat{\theta}$) with confidence intervals, and ensuring evaluator reliability through judge pinning and regular re-validation against human labels.

CI and CD form a continuous improvement loop: online monitoring identifies issues, prompting error analysis, updates to evaluation artifacts, pipeline fixes, and redeployment.

Sustaining high-quality LLM systems requires treating evaluation as

a continuous engineering process. While automation is essential, human oversight remains critical for interpreting results, detecting new failures, and adapting to evolving use cases. The next chapter focuses on building interfaces to support this review.

10 Interfaces for Continuous Human Review and Error Analysis

In the previous section, we explored how Continuous Integration (CI) and Continuous Deployment (CD) practices help maintain and monitor LLM pipelines. We saw how the CI/CD "flywheel" drives ongoing refinement.

However, relying solely on automated evaluators, whether code-based checks or LLM-as-Judge systems, is insufficient. LLM judges themselves can drift or misinterpret criteria over time. Moreover, even human criteria over time (Shankar et al. 2024d). As such, we need to ensure judges remain aligned with our quality standards, as well as discover new failure modes to implement new LLM-as-Judge evaluators for.

The challenge, then, is not whether to involve humans, but *how* to do so effectively. Production systems can generate thousands or millions of traces. Reviewing even a fraction requires significant effort. If the review process is slow, cumbersome, or frustrating, the feedback loop breaks, and the **Improve** phase in Figure 2 stalls.

In this section, we argue that investing in custom-built review interfaces is essential for maximizing the speed, quality, and impact of human feedback. We'll outline key interface principles from HCI that apply to LLM review. We'll cover features that make reviewing faster and more reliable, strategies for selecting the most informative traces, and ways to embed review into the broader engineering workflow. Along the way, we'll walk through case studies and real interface examples that bring these ideas to life.

10.1 The Case for Custom Review Interfaces

When first faced with reviewing LLM outputs, the path of least resistance often involves using existing tools. We might export logs to a spreadsheet or use a generic log viewer. While seemingly simple, these approaches quickly reveal their limitations.[86]

Spreadsheets, for instance, struggle with the *complexity* of LLM traces. A single trace might contain a long input prompt, multiple intermediate LLM reasoning steps, several tool calls with structured arguments and responses, retrieved RAG context, and the final output. Cramming this rich, multi-faceted data into flat rows and columns in a spreadsheet is awkward. Navigation is tedious. Providing structured feedback or typing free-form notes in a cell, if long, is also difficult. Some observability interfaces might handle raw text better but typically lack tailored features for efficient labeling or visualizing LLM-specific structures (e.g., tool calls).

The limitations of these default tools hinder the core goal: *maximizing high-quality feedback per unit of reviewer time*. Slow interactions, poor data presentation, and inadequate feedback mechanisms lead to reviewer

[86] **Hamel's Note:** For many situations, the path of least resistance is building your own tools. This is because AI-assisted coding excels in web applications that display data. If you are using an observability system that allows you to read traces (e.g. Langsmith, Braintrust, Arize, etc.), it is worth looking at your data there first to understand how you could customize the experience before building your own. More on this later.

fatigue and lower-quality annotations.

Moreover, because humans are visual creatures, custom interfaces enable better data exploration and insight by presenting complex trace information in digestible ways. For example, if reviewing outputs of an email assistant, it is easier to review the output if it is presented like an email, not some markdown string in a spreadsheet cell. Or, if the LLM pipeline's task is to tag documents, presenting these tags as visually distinct elements (e.g., color-coded labels resembling UI tags) would be more effective for review than a simple list of categories in a spreadsheet cell. Ultimately, a custom interface improves consistency and efficiency.

Building custom tools might sound daunting, but it doesn't necessarily require extensive frontend development effort. We can leverage modern web frameworks or even use LLMs themselves to help generate basic interfaces—a process sometimes called *"vibe coding."*

Note to readers

The phrase "vibe coding" refers to a rapid prototyping style where developers use LLMs (such as ChatGPT, Claude, or Replit's coding agents) to generate lightweight interfaces from natural language prompts. For example, asking "generate an HTML dashboard and serve it via FastAPI to review text generation outputs with pass/fail buttons and a feedback textbox" can yield a working UI in minutes. This approach lowers the barrier to interface development and allows fast iteration without deep frontend expertise.
We demonstrate this technique in a live video walkthrough,`https://www.youtube.com/watch?v=qH1dZ8JLLdU` where we vibe-code an app to support data review and error analysis (Section 3).

10.2 Principles of Effective Review Interfaces

We draw heavily on foundational principles from Human-Computer Interaction (HCI) to guide our design. Relevant principles, mostly dating back to Jakob Nielsen's web usability heuristics (Nielsen 1994), are outlined in Table 7. In what follows, we translate these principles into specific design recommendations for human-in-the-loop review tools. We describe the key interface elements that every LLM review tool should include, as well as additional features that significantly improve review speed, accuracy, and user satisfaction.

Essential Interface Elements. A well-designed review interface must show the full LLM trace and make it easy to provide feedback. This includes not only the input and output, but also intermediate steps like tool calls, retrievals, or internal reasoning. To manage complexity, interfaces can collapse less important sections by default, but the full trace should always

Principle	Design Implication
Visibility of System Status	Show which trace is being reviewed, how many remain, and feedback status. Provides timely feedback (Nielsen 1994; Norman 2013).
Match Between System and Real World	Use familiar terms from rubrics and evaluation criteria; make mappings intuitive (Nielsen 1994; Norman 2013).
User Control and Freedom	Allow navigation (next/previous), undo, and defer actions. Prevent "lock-in" of any notes (Nielsen 1994; Shneiderman 1997).
Consistency and Standards	Reuse layouts, terminology, and behaviors across traces (Nielsen 1994; Shneiderman 1997).
Error Prevention	Use distinct labeling controls; enforce requirements like rationale for "Fail" to reduce mistakes (Nielsen 1994; Norman 2013).
Recognition Rather Than Recall	Show predefined failure modes and explanations visually to reduce cognitive load (Nielsen 1994; Shneiderman 1997).
Flexibility and Efficiency of Use	Offer shortcuts and accelerators for expert users; keep interface intuitive for novices (Nielsen 1994; Shneiderman 1997).
Aesthetic and Minimalist Design	Emphasize signal over noise. Highlight inputs and outputs clearly (Nielsen 1994; Tufte 2001).

Table 7: Core HCI principles that inform the design of effective human review interfaces for LLM pipelines. These principles help ensure that interfaces remain usable, efficient, and aligned with reviewer needs during continuous evaluation.

remain accessible for accurate judgment.

The interface should also support structured, consistent feedback collection. This often involves tagging known failure modes using binary checkboxes or buttons, based on issues identified during Error Analysis (Section 3) or Collaborative Evaluation (Section 4). Quick overall ratings, such as thumbs up/down or numeric scores, help speed up the process. Open-ended feedback fields allow reviewers to explain their reasoning—critical for surfacing unexpected problems or refining evaluation criteria over time.

Features that Improve Review Speed and Quality. The following enhancements help reduce reviewer fatigue and increase the reliability of collected labels:[87]

- **Inline Annotation:** Allow span-level feedback, such as highlighting problematic sentences or marking faulty dialogue turns.
- **Keyboard Shortcuts (Hotkeys):** Provide hotkeys for actions like submitting feedback, tagging failure modes, and moving between traces to speed up expert workflows.
- **Visual Hierarchy:** Emphasize important trace components (e.g., the final output) using layout, color, or segmentation. For instance, highlight different sentences in generated emails to surface verbosity or flow issues (Example 2). This hierarchy should be domain specific, making it easy to quickly see important parts of the trace that tend to be important, whereas hiding the rest by default.
- **Batch Review & Bulk Actions:** Let users review trace clusters (e.g.,

[87] **Hamel's Note:** There is a direct correlation between speed at which you can look at data and the engineering velocity of your AI app. Building a data annotation app is one of the highest leverage activities you can do. You might be tempted to shop for vendors that have all the "features" we outlined here. However, out-of-the-box ("OOB") interfaces are limited in their ability to fit your data. However, OOB interfaces are often a good starting point and allow you to get an idea of what you want to build.

from the same client persona) and optionally apply the same failure label to multiple examples—useful for repetitive issues, but should be used with care.

- **Progressive Disclosure:** Simplify the interface by showing only basic options (e.g., "Pass," "Fail," "Defer") at first, revealing detailed tagging only when needed.
- **Feedback Templates:** Offer frequently used comments (e.g., "Tone too casual," "Missing key fact") as quick-select or dropdown options, optionally adapted to selected failure tags.
- **Defer Option:** Provide a clear, prominent "Defer" button for uncertain cases—critical for maintaining label accuracy and avoiding low-confidence judgments.
- **Contextual Metadata:** Show relevant metadata such as user segment, pipeline version, and automated evaluation results directly in the interface to aid reviewer decisions.
- **Filters:** Allow filtering on important dimensions relevant to the product or domain. Often, you want to filter by key contextual metadata fields. For example, if your user can interact with your AI through multiple channels (text, dedicated app, website, etc.) you would want that as a filter.
- **Progress Indicators:** It can be useful to show counts of traces reviewed, potentially grouped by key dimensions.

10.3 Case Study: EvalGen Interface and Insight

The design principles above can feel abstract, but real interfaces illuminate how even minimal UI choices impact review quality. One such example is **EvalGen**, a lightweight, early-stage interface for human evaluation of LLM outputs (Shankar et al. 2024d).

EvalGen implemented a simple but structured UI: it displayed one output at a time, allowed binary feedback (e.g., thumbs up/down), supported open-ended comments, and gave reviewers the ability to edit past labels. It also included a lightweight rubric builder—automatically extracting candidate evaluation criteria from the original prompt and letting users select which ones to apply during review. A screenshot of the interface is shown in Figure 11, and a "version 2" of EvalGen is publicly available in ChainForge, an open-source LLM chain builder (Arawjo 2025a).

EvalGen prioritized *which* traces to show based on LLM judge disagreement: when multiple automated evaluators (e.g., separate LLMs or LLM judge prompt variants) disagreed on a verdict, that example was surfaced for human review.

EvalGen's user studies surfaced a deeper insight: even human reviewers experience *criteria drift*. Over time, annotators refined what "correctness" or "polish" meant, revised prior judgments, and added new failure cate-

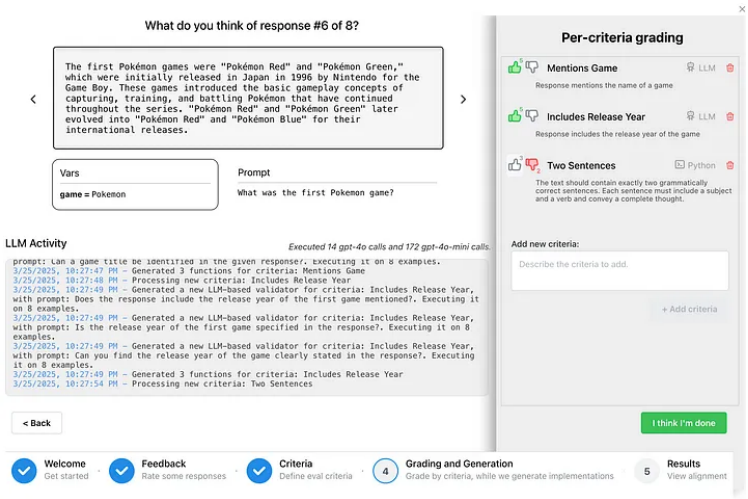


Figure 11: Screenshot of the EvalGen v2 interface, from (Arawjo 2025a). The UI presents one trace at a time, allows binary grading, supports editable labels and open codes, and surfaces high-uncertainty examples based on disagreement between automated LLM judges.

gories that weren't anticipated up front. This finding reshaped the design of EvalGen and points to a general lesson: human review tools must treat evaluation as an *evolving sensemaking process*, not just a static labeling task.

This experience motivates the first ingredient in building effective review systems: **how to select which traces to show**. Even with a well-designed interface, showing the wrong examples leads to wasted effort. We now turn to sampling strategies that help reviewers focus on high-signal data.

10.4 Selecting Traces from the "Firehose" for Human Review

Even with a well-designed interface, reviewing every production trace is impossible. LLM pipelines can produce thousands or millions of outputs, and only a fraction can be reviewed by humans. To make the most of limited human attention, we need smart strategies for choosing which traces to show. In other words, we need to filter the "firehose."

The goal is to prioritize traces that are most likely to uncover quality issues or help improve the system. Several sampling strategies can help:

Random Sampling. The simplest approach is to randomly sample traces from the production stream. This method is unbiased and gives a general sense of system performance, especially if the sample size is large. However, it may miss rare but important failure modes, and if most outputs are already good, it can be inefficient.

Uncertainty Sampling. Uncertainty sampling focuses review on traces where automated evaluators disagree or appear unsure.[88] These traces are often the most informative for human review. We can identify uncertainty

[88] This approach is closely related to active learning in machine learning, where the model selectively queries the most uncertain examples for human labeling to improve learning efficiency (Settles 2009).

in several ways:

- A trace fails some criteria but not others, suggesting borderline quality.
- An LLM judge produces inconsistent results when run multiple times with temperature > 0.
- Different evaluators (e.g., a code-based check vs. an LLM judge) give conflicting judgments.

This last case what EvalGen adopted (Section 10.3), which surfaces traces for human review specifically when automated judges disagree.

Failure-Driven Sampling. In many systems, we already log when something goes wrong—an output fails a guardrail check, triggers a timeout, or receives user-reported feedback. Failure-driven sampling focuses on these traces, using monitoring and logs to pull the most obviously problematic cases for review. This approach is high-precision but lower-recall; it won't catch subtle or unexpected issues, but it ensures that known errors get addressed quickly.

10.5 Navigating Trace Groups and Discovering Patterns

While reviewing individual traces is important, many issues only become visible when we step back and look at groups of examples. To support this, good review interfaces include tools for exploring patterns across similar traces—especially through clustering and search.

Clustering. Clustering lets us group traces that share common features. This can be done using simple metadata or semantic similarity. For example, in our real estate assistant (Example 2), we might cluster traces by client persona (e.g., "investor," "first-time buyer") or by the feature used (e.g., email generation vs. property search). Reviewing a few traces from the "investor" cluster might reveal a recurring issue—such as emails that sound too casual or lack data-driven recommendations.

Semantic clustering goes a step further. Here, we embed the inputs (e.g., user queries) or outputs (e.g., generated emails or SQL queries) and group them based on vector similarity using algorithms like k-means. This might reveal latent categories, such as all queries about "affordable condos near good schools." Reviewing that cluster could uncover a systematic failure, like the model ignoring HOA fees or misunderstanding school quality indicators.

Interfaces can surface these clusters to help reviewers quickly find common patterns, investigate root causes, and identify where fixes are most needed.

Search and Similarity Tools. Clustering is useful for broad exploration. But sometimes, after spotting an interesting trace, we want to dig deeper.

Review tools should support easy ways to find similar examples—by keyword, metadata, or semantic content.[89]

Together, clustering and search allow reviewers to move beyond one-off examples and uncover systematic failures. The most effective workflows blend these group-level tools with targeted sampling (Section 10.4) to guide attention toward the most informative parts of the firehose.

10.6 *Integrating Human Review into the Bigger Engineering Workflow*

A review interface is only valuable if people use it regularly[90], and if the feedback actually leads to changes in the system. Human review should be part of the team's normal engineering workflow, not something done occasionally or after the fact.

To support this, we can set up automated systems that surface the right traces at the right time. For example, we might schedule a daily or weekly job that selects a batch of traces using the sampling strategies from Section 10.4, then notify reviewers via Slack or email when the batch is ready (e.g., "Your daily review set is available"). For high-priority issues—such as safety violations, broken tool calls, or major regressions flagged by online monitoring (Section 9.2)—we can trigger alerts immediately and route them to on-call reviewers.

Review feedback must also flow back into the pipeline. One way is to update evaluation artifacts:

- **Golden Dataset:** Traces labeled by reviewers (especially failed ones) can be added to the CI test set used for regression
- **LLM-as-Judge Evaluators:** Human labels provide a way to monitor whether our automated judges are still accurate. For instance, if humans repeatedly flag outputs as incorrect that the judge rated as "Pass," we may need to revise the judge's prompt, retrain the model, or even swap it out (Section 5.3, Section 9.2).

To make this loop efficient, the review interface should support lightweight actions that translate human insight into engineering outcomes. For instance, a reviewer should be able to click a button to add a trace directly to the CI golden set, or file a bug report with trace details already prefilled. Dashboards can show recent review activity—such as which failure types are appearing most often or where reviewers disagree with LLM judges—so teams can spot trends quickly and prioritize fixes accordingly.

Overall, when human review is embedded into daily routines, and the interface supports easy follow-through, human feedback becomes a natural part of the development process. It helps teams catch issues faster, fix the right problems, and maintain quality as systems evolve.

[89] Keyword search allows reviewers to locate traces with specific terms. Metadata filters (like `pipeline_version`, `user_segment`, or failure tags) help slice the data precisely. Semantic search lets reviewers find conceptually similar traces, even when wording varies. For example, if one email gives unsolicited financial advice, semantic search could help surface others with the same issue.

[90] **Hamel's Note:** If people aren't using your interface regularly, you probably haven't reduced the friction of using it enough. It's critical that you put the work into making it good.

10.7 *Example Walkthrough: Reviewing Real Estate Assistant Emails*

To make everything we've discussed in this section so far concrete, let's walk through how a review interface might evolve for a real estate assistant that drafts client emails (Example 2). We'll see how interface choices affect usability, efficiency, and the quality of feedback.

The most basic interface is: a spreadsheet. We can export logs—including the user's query, client persona, and the generated email—into a tool like Google Sheets. This is easy to start with but quickly becomes painful. Reviewing a single trace often means scrolling across many columns or flipping between sheets. Emails appear as raw text crammed into cells. Feedback is just a free-form comment in a single column—mixing unrelated notes and failure types. Navigation is slow, there are no hotkeys, and the experience is frustrating and error-prone.

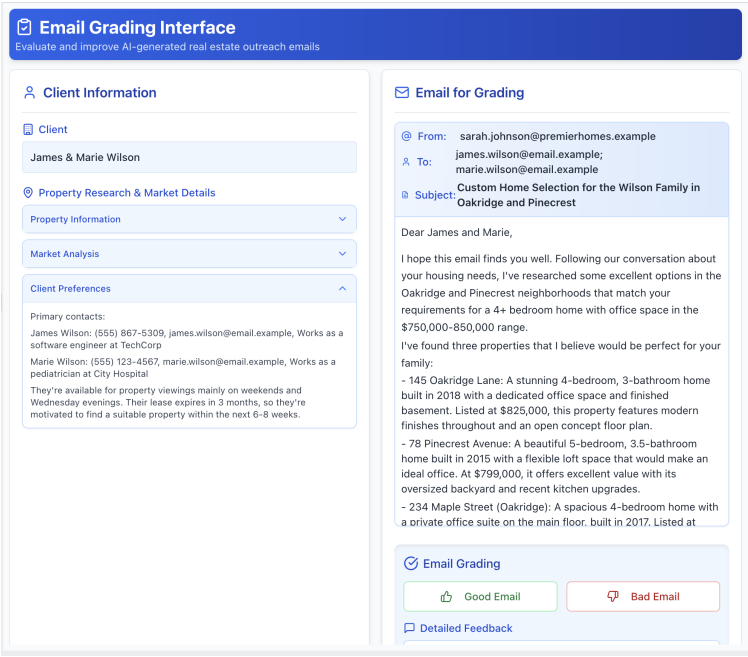
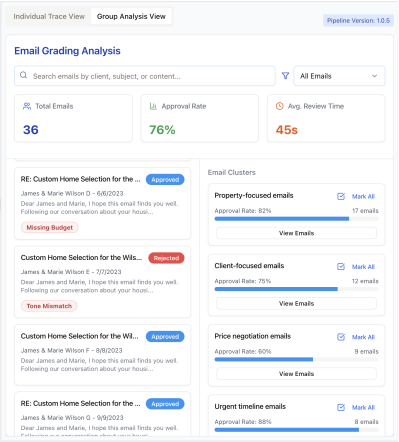


Figure 12: A basic custom interface for reviewing real estate assistant emails. Client info and generated text are shown side by side, with Good Email"/Bad Email" buttons and an optional free-text comment box. Even a simple layout like this outperforms spreadsheets.

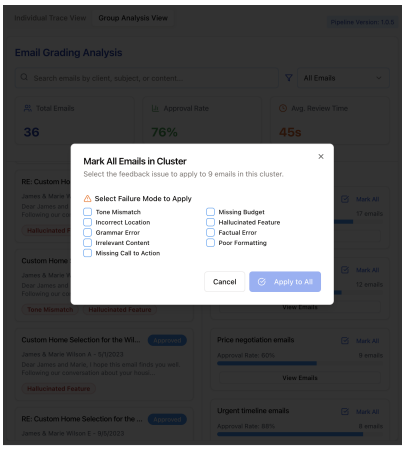
A clear step up is building a lightweight custom UI (Figure 12). This could be done using Streamlit, Gradio, or with the help of an AI coding assistant. The UI shows one trace at a time, with the user input, persona context, and generated email clearly separated. Reviewers can click Pass/-Fail buttons and leave optional comments. Basic navigation buttons (e.g., Next/Previous) keep the workflow moving. Even this minimal interface makes a big difference: better layout, faster decisions, and fewer mistakes.

But a basic UI still leaves out important features. It doesn't show intermediate steps like tool calls or, for a RAG pipeline, external context retrieved for LLM steps. There are no structured failure tags (i.e., axial

codes from Section 3), no way to highlight specific issues in the output, and no support for reviewing multiple related traces together.



(a) Cluster view showing groups of emails, such as property-focused or client-focused examples. Reviewers can drill into a group to see individual traces.



(b) Bulk tagging interface for labeling all traces in a cluster with a shared failure mode (e.g., "Tone too casual").

Figure 13: Group-level review tools. Clustering helps reviewers find patterns, while bulk actions make labeling faster.

A more advanced interface (Figures 13 and 14) brings all of this together. Individual traces are presented with rich, readable layouts: input query, persona context, and the full output, along with any intermediate reasoning or tool usage. Emails are shown with clear visual formatting—highlighting key entities like price, location, or timing—to make errors easier to spot.

Reviewers can select predefined failure tags (like "Tone Mismatch" or "Hallucinated Feature"), write free-form notes (i.e., open codes from Section 3), and use hotkeys to speed up review (e.g., N for Next, D to Defer, T for Tone). Contextual metadata—such as the trace ID, pipeline version, or timestamp—appears directly in the interface to help with triage.

In addition to reviewing individual traces, the interface supports group-level exploration. Clustering can group similar emails, such as those for "investor clients" or "affordable condo searches." Reviewers can examine a few examples from each cluster to spot patterns or apply bulk labels to speed things up. Search tools let them find similar examples—for instance, "all emails that sound too apologetic"—and explore variations across the dataset.

With the right interface, reviewing LLM outputs becomes faster, more consistent, and more useful. Small UI improvements—better layouts, search, tagging, clustering—translate directly into better data and better decisions.[91]

[91] **Shreya's Note:** Some of the highest-leverage features include: hotkeys for faster navigation, structured failure tagging to reduce ambiguity, and clustering to reveal repeated failure modes. These features tend to produce immediate gains in review speed and label quality with minimal implementation effort.

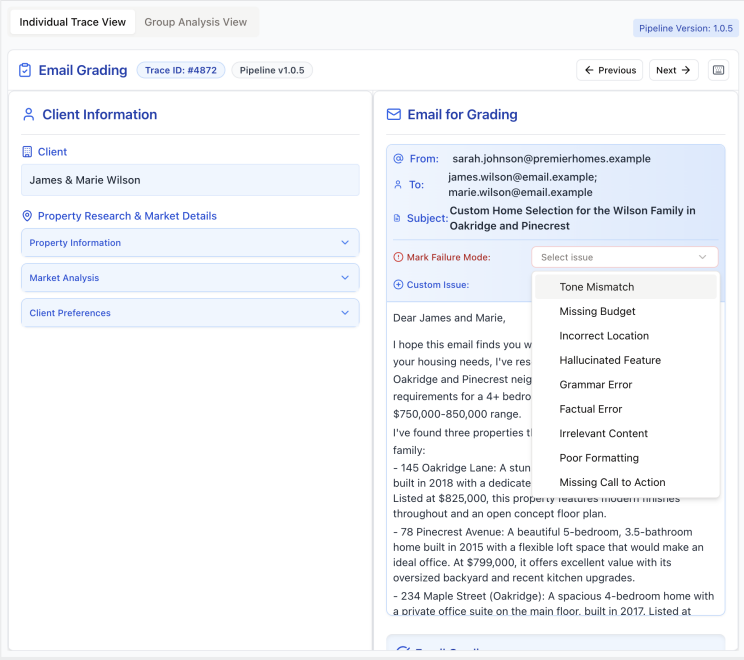


Figure 14: "Advanced" Custom UI for individual trace review of the real estate assistant. This view presents client information, the full email for grading, and structured feedback options, including a predefined list of common failure modes (e.g., 'Tone Mismatch', 'Missing Budget', 'Hallucinated Feature'). Contextual information like Trace ID and Pipeline Version is also visible, aiding detailed analysis and efficient, consistent labeling.

10.8 Case Study: DocWrangler for Prompt Refinement

Interfaces don't just support evaluation—they also shape how we iterate and improve our systems. **DocWrangler** is a purpose-built IDE for LLM-powered document processing pipelines (Shankar et al. 2025). It illustrates many of the interface principles discussed in this section, particularly how layout, interaction design, and lightweight feedback mechanisms can accelerate iterative development.

In DocWrangler, users author prompts to extract, reason about, and make sense of information from unstructured documents. The interface lets them run their prompt over a batch of documents, then review the outputs one by one. Each screen shows the original document, the generated output, and the prompt that produced it—organized side by side to reduce friction when inspecting failures or inconsistencies (Figure 15).

Crucially, the interface encourages *open coding* (Section 3). Users can quickly leave natural-language comments on any example, jotting down observations like "missed the date field," "output order is inconsistent," or "ignored second table." These lightweight notes are faster than formal labeling but still capture valuable signals. Over time, they help users notice recurring problems and refine their mental model of what's going wrong.

Once enough notes are collected, users can invoke the **prompt improvement** feature. This takes the current prompt and the accumulated reviewer comments, and suggests an updated prompt aimed at fixing the issues. The interface presents both versions side by side, allowing quick

inspection and reuse (Figure 16). This closes the loop between feedback and action—a central goal of any effective review interface.

DocWrangler exemplifies several key interface principles: it reduces cognitive load through clear layout, prioritizes efficiency with fast review actions, and supports human sensemaking through freeform notes.[92] While tailored to document extraction, the same ideas apply broadly—interfaces that streamline review and close the feedback loop can dramatically accelerate LLM pipeline development.

[92] DocWrangler is available to try out at docetl.org/playground.

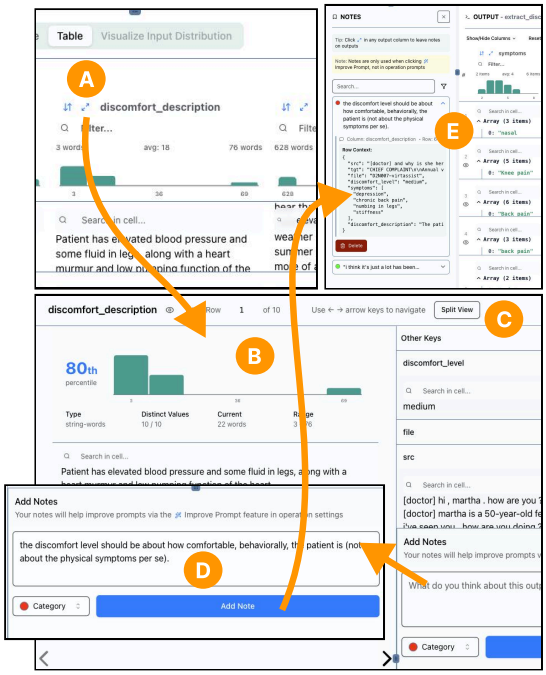


Figure 15: The DocWrangler interface for reviewing document-level outputs. Users can view the document (A), the model's extracted output (B), and the associated prompt (C) side by side. Reviewers can leave open-coded notes (D) to identify issues, and inspect multiple examples in a single session (E). Taken from Figure 4 in (Shankar et al. 2025).

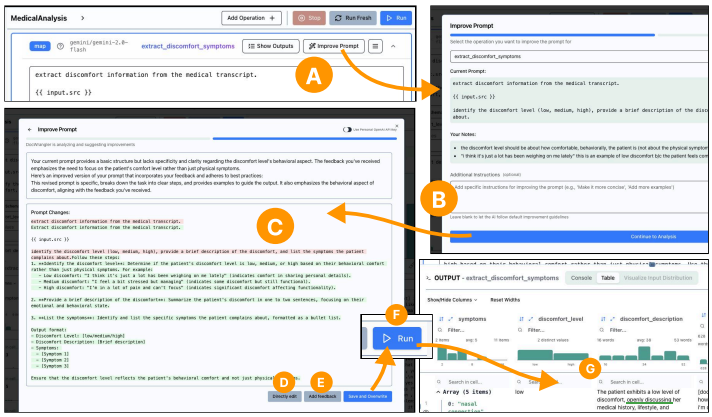


Figure 16: Prompt improvement workflow in DocWrangler. Based on accumulated user feedback, the interface generates a revised version of the original prompt (B), displayed alongside the current prompt (A). The user can inspect both and accept, reject, or further edit the update. Taken from Figure 5 in (Shankar et al. 2025).

10.9 Summary

Automated evaluation helps scale oversight, but it can't replace human judgment. Humans are still essential for catching subtle errors, surfacing unexpected issues, and refining what "quality" really means. Continuous human review is not a nice-to-have—it's critical for maintaining high-quality LLM applications.

But human attention is limited. To make review practical, teams need interfaces that are fast, clear, and tailored to the task. Spreadsheets and generic tools slow reviewers down. Custom interfaces—designed with principles from HCI—can dramatically increase feedback quality by showing full trace context, supporting structured input, and reducing cognitive load.

Since we can't review every trace, sampling strategies like uncertainty-based or failure-driven selection help focus attention on the most valuable examples. And review only matters if it leads to action: interfaces should make it easy to file bugs, update evaluation datasets, and trigger prompt improvements.

Done well, human review becomes a core part of the engineering loop. It helps teams find problems faster, iterate with confidence, and build systems that better match the messy, subjective needs of real users.

11 Improvement

Once we've shipped a working LLM pipeline, the next challenge is making it better. This chapter focuses on what comes after "it runs": improving the quality of results and the efficiency of the system. In the lifecycle we introduced earlier (Figure 2), this is the final stage—Improve—where we apply what we've learned from evaluation and monitoring to make the system more robust and cost-effective over time. We cover two main areas: **accuracy optimization**, or making the model more correct, reliable, or aligned with our needs, and **cost optimization**, or delivering that same quality more efficiently, especially at scale.

We'll start with strategies to increase accuracy, then turn to cost. All of these techniques depend on the earlier phases of the lifecycle—without good evaluators and monitoring, we can't tell whether our changes actually improve anything. But with the right data and a few targeted changes, we can make real progress.

11.1 Accuracy Optimization

Improving accuracy means making the model more likely to produce the correct, helpful, or expected output. It's the first place we usually focus when a pipeline starts failing evaluations or frustrating users. Often, meaningful improvements can come from small changes—tweaking the prompt, adding examples, or restructuring the task. Other times, deeper changes are needed, like breaking the problem into smaller parts or adjusting how we retrieve and feed context to the model.

Below, we walk through concrete strategies to improve accuracy. We start with easy wins that require minimal engineering effort, then move to structural changes that may require refactoring, and finally cover heavier interventions like fine-tuning.

Quick Wins. The fastest way to improve accuracy is to refine the prompt. Prompt refinement includes includes clarifying instructions, adding few-shot examples, and steering the model with short, explicit personas. These are low-effort changes that can have a big impact.

- **Clarify ambiguous wording.** If the model gets confused about phrasing (e.g., "West Berkeley" vs. "Berkeley West"), we update the prompt to be more explicit or include an example that disambiguates.
- **Add a few examples.** We can include 2–3 representative input/output pairs in the prompt. These examples should target failure cases we've observed in evaluation, but be distinct from test cases used in CI.
- **Use role-based guidance.** Giving the model a persona (e.g., "You

are a careful tax advisor...") can help guide tone and reasoning style, especially for open-ended tasks.

- **Ask for step-by-step reasoning.** For tasks involving logic or multiple steps, explicitly asking the model to "think step by step" can improve correctness and completeness.

These changes are fast to try and easy to validate with existing golden test sets.

Structural Changes. When quick fixes don't improve accuracy enough, we can look at how the overall pipeline is structured. Here are some ideas:

- **Break the task into smaller steps.** Instead of a single LLM call that does everything, we can decompose the task. For example:

  1. Extract the user's preferences.
  2. Query a database or tool.
  3. Filter or summarize results.
  4. Generate a final message or answer.

  This lets us use more targeted prompts at each stage and isolate where errors occur.

- **Tune RAG (Retrieval-Augmented Generation) steps.** If we're using retrieved documents, quality often depends more on retrieval than generation. We may need to:

  – Rework the query we send to the retriever
  – Change how we chunk the source documents
  – Retrieve more or fewer chunks
  – Add a lightweight re-ranking step before sending the results to the LLM

- **Fix tool misuse.** If the model keeps calling a tool incorrectly (e.g., wrong argument formats, missing parameters), we can revisit how we describe the tool in the prompt. Simple clarifications—like explaining acceptable values or formats—can reduce invalid calls.

- **Add lightweight checks.** In some cases, it helps to introduce basic validations before returning a result—e.g., regexes to detect malformed outputs, length constraints, or blocklists for unsafe content. These aren't perfect, but they can catch common failure modes.

Heavier Fixes. If we've tried the steps above and still see recurring problems—especially for well-defined, high-value tasks—it might be time to consider model-level changes. One strategy is **fine-tuning.** When we have hundreds of high-quality labeled examples and can't fix the issue with prompting, we can consider fine-tuning a model on those examples. Fine-tuning is slower and more expensive, but can lock in improvements when nothing else works. Another strategy is to **add more human review loops.** For tasks with high stakes or hard-to-detect failure modes, it's often worth building a lightweight interface for human reviewers to check and annotate a small sample of outputs regularly. Annotations can inform fine-tuning datasets. Finally, we can also apply **prompt optimization**: for pipelines where prompts are reused across thousands of requests, even small changes can have a large impact. We can treat the prompt as a tunable artifact: try variations, test them against held-out golden sets, and keep what improves accuracy (Opsahl-Ong et al. 2024). Some teams go further and treat prompt editing as a formal optimization problem, using techniques like grid search over templates or reinforcement learning with evaluation-based rewards.[93]

[93] We recommend *not* turning to prompt optimization or even fine-tuning until you have certainly exhausted all other methods of accuracy improvement.

11.2 Quick Wins and Best Practices for Cost Reduction

Here are some strategies that often yield significant cost savings with relatively low effort.

Tiered Model Selection. Not every task requires the immense power—and associated price tag—of the largest foundation models. We can practice a "tiered" model selection: matching the model's capability to the complexity of the specific task it needs to perform.

First, we identify the different tasks within our pipeline and assess their requirements. Simple, high-volume tasks like basic text classification (e.g., sentiment analysis), straightforward data extraction from structured text, or initial user intent recognition can often be handled effectively by smaller, faster, and significantly cheaper models (e.g., GPT-4o-mini, Gemini 2.0 flash, various smaller open-source or fine-tuned models).[94] We reserve the premium, state-of-the-art models (e.g., GPT-4o, Claude 3.X Sonnet, Gemini Pro) for sub-tasks that genuinely demand their sophisticated reasoning abilities, complex instruction following, or where the stakes for accuracy are exceptionally high.

[94] Note that the exact model names are frequently changing, as newer and more capabile models are released.

For example, consider a customer support chatbot. We might use a cheaper model for the initial step of classifying the user's incoming message into categories like "billing question," "technical issue," or "general inquiry." Only if the query is classified as complex, or requires generating a detailed, personalized explanation, would we route it to a more capable, expensive model for the subsequent steps.

Divide and Conquer with Task Decomposition. Complex end-to-end workflows can often be broken down into a sequence of smaller, more manageable sub-tasks. By decomposing the problem, we gain opportunities to assign different models to different steps, potentially using cheaper models for suitable intermediate stages.

Consider a pipeline designed to answer questions based on lengthy technical manuals using Retrieval-Augmented Generation (RAG). Instead of using a single, powerful LLM for all generation steps, we might decompose it:

1. *Retrieval:* Use a specialized, cost-effective embedding model to find relevant document chunks.
2. *Re-ranking/Filtering:* Employ a fast, inexpensive LLM (or re-ranker embedding model) to quickly score the relevance of the retrieved chunks and filter out less promising ones before sending them to the main generation model.
3. *Synthesis & Generation:* Use a more capable (and expensive) model to synthesize the final answer, but provide it only with the top-ranked, filtered chunks from the previous step. Since there are fewer tokens in the prompt, the cost decreases.

Cutting Tokens in Prompts. Since most LLM providers charge based on the number of processed tokens (both input and output), we can save cost by reducing the number of tokens. First, we consider how to optimize the number of input or prompt tokens. The most obvious strategy is to make sure instructions are concise, and for specialized architectures (e.g., RAG), tune the number of documents retrieved and placed in a prompt to elicit answers from the LLM.

One strategy is to **pre-process inputs into summaries**. Suppose our LLM pipeline operates on documents—e.g., question-answering, creating a knowledge graph, etc. If our pipeline requires multiple distinct LLM calls to analyze or extract information from the same long document, we could add an initial step where a cheaper model summarizes the document (or relevant sections). Subsequent calls then operate on the much shorter summary, drastically reducing input tokens for each downstream call. For instance, when analyzing a lengthy legal contract to extract various clauses and obligations, summarizing the contract once upfront can be much cheaper than feeding the full text to multiple specialized extraction prompts.[95]

We can also consider optimizing the number of output tokens for an LLM call. Again, the obvious strategy is to explicitly instruct the LLM to be concise when appropriate. For tasks where the output should follow some structure, requesting output in a compact format like YAML, rather

[95] Instead of an LLM-powered summarization pre-processing step, we could also ask the LLM to output line numbers or regular expressions to match statements that are obviously not needed in the downstream LLM calls.

than natural language prose or even JSON (which contains many braces or brackets), can reduce the number of generated tokens.

11.3 Leveraging LLM Provider Caching

Many LLM API providers (e.g., OpenAI, Google, Anthropic) use caching mechanisms that can significantly reduce both latency and cost for repeated requests. Modern LLMs are based on the Transformer architecture. Transformers work by computing, at each step, a set of "key" and "value" vectors for every token processed so far. Keys serve as lookup signals to decide which past tokens matter most. Values carry the actual information those tokens contribute. When generating long sequences or handling many similar prompts, recomputing all key and value states from scratch is costly.

LLM inference engines mitigate this redundant computation by storing intermediate key/value (KV) states in a cache. When a new prompt begins with the exact same token sequence (the *prefix*), the engine reuses the cached KV states for that prefix and computes only the states for the new suffix. State reuse reduces both latency and compute overhead.

Caching applies only when the prefix matches *exactly*. A single-token change invalidates the cache beyond that point, and forces full recomputation. Consider a simple task: "*Think of a friendlier way to say: X*" where X is any piece of text, e.g., "hello my name is Shreya" or "hello my first name is Shreya." In the cache-friendly layout, the instruction comes *before* the data (Figure 17). If instead the instruction follows the data, only the repeated words remain in the prefix, as shown in Figure 18.

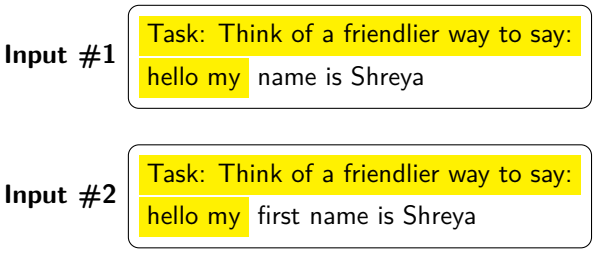


Figure 17: Cache-friendly prompts: both prompts share the highlighted prefix (instruction + "hello my"). So, we can pay a discounted token rate for the highlighted tokens in Input # 2.

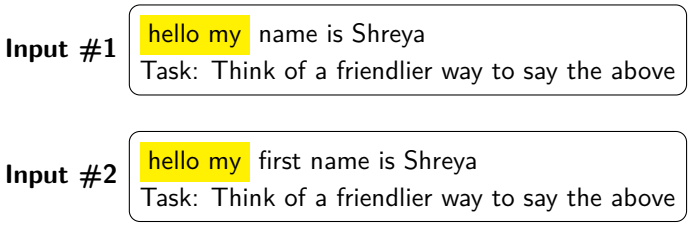


Figure 18: Cache-unfriendly prompts: only the data prefix ("hello my") matches, so minimal reuse. We will pay a discounted token rate for the two highlighted words in Input #2.

Overall, strategically placing static instructions before the changing content maximizes KV cache reuse and improves inference efficiency.

Overall, to take advantage of caching, put constant instructions *before* any user-provided or changing data. Keep formatting and phrasing consistent across requests. Consult the specific LLM provider's documentation to find the associated cost savings, as pricing for cached vs. non-cached tokens can differ a lot between providers.

11.4 Advanced Strategy: Implementing Model Cascades

When we care about both quality and cost, we can leverage a technique called *model cascades*. ***The goal of a model cascade is to match the accuracy of a trusted, expensive model (the "oracle") at a lower cost.*** The cascade is designed to efficiently approximate the oracle's predictions, using the oracle's output as the target standard.[96]

For the purpose of the cascade algorithm, the oracle model is assumed to represent the best available solution we are willing to pay for. Its actual accuracy with respect to an external "ground truth" is not the primary consideration; the system does not try to estimate or compensate for the oracle's own imperfections. The "accuracy guarantee" we aim for in a cascade is always relative to the oracle, not to an external ground truth.[97] For example, if our oracle is 80% accurate on a ground-truth dataset and a cheaper "proxy" model is 70% accurate, the cascade's objective is to recover as much of the oracle's 80% performance as possible, but at a reduced cost. It does not attempt to exceed the 80% benchmark.

A typical model cascade, as depicted in Figure 19, works as follows:

1. A query arrives.
2. The proxy model tries to answer it.
3. We check how confident the proxy is in its answer.
4. If it's confident enough, we return its answer.
5. If it's not, we escalate the query to the oracle and return that result instead.

The challenges are to determine confidence scores for the proxy model's outputs and to set the confidence thresholds carefully. The goal is to set a quality target—for instance, ensuring the cascade's final output matches the oracle's output at least 95% of the time—while maximizing the percentage of queries handled by the cheaper proxy.

Quantifying Proxy Confidence with Log Probabilities. To make a cascade work, we need to estimate how confident the proxy model is in its own answers. If it seems confident, we return its answer. If it seems unsure, we send the query to the more expensive oracle model instead.

[96] Model cascades are especially useful for cutting monitoring costs in pipelines that use LLMs as evaluators (LLM-as-Judge).

[97] **Hamel's Note:** The oracle does not need to be perfect; it only needs to be the best system we currently have. The goal of the cascade is to reproduce the oracle's behavior as cheaply as possible.

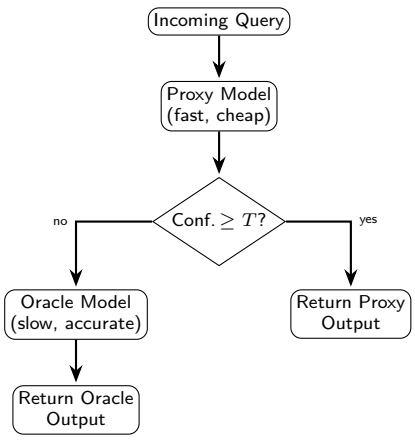


Figure 19: Example dataflow in a two-model cascade.

Many LLM APIs (like OpenAI, Anthropic, or Google) let us request *log probabilities* along with the model's response. A log probability (or logprob) tells us how strongly the model believed in the token it just generated. A higher logprob means the model was more confident in choosing that word.

For simple classification tasks—like predicting `true` or `false`—individual token probabilities can be very reliable. Suppose we ask the model a yes/no question. If the logprob of `true` is much higher than that of `false`, the model is confident. If they're close, it's uncertain. Let's say we ask the model to choose one token from a fixed list (e.g., `true`, `false`, or `maybe`). Here's how we turn logprobs into a confidence score:

1. Request logprobs for all the valid options when we call the model.
2. Convert each logprob to a regular (unnormalized) probability using the exponential function: $P(t) = \exp(\ell(t))$.
3. Normalize the probabilities so they add to 1.
4. Use the normalized probability of the chosen token as our confidence score.

For example, if the model's logprobs are:

$$\ell(\texttt{true}) = -0.10, \quad \ell(\texttt{false}) = -2.30$$

then:

$$P(\texttt{true}) = e^{-0.10} \approx 0.90, \quad P(\texttt{false}) = e^{-2.30} \approx 0.10$$

The normalized probabilities are roughly:

$$\hat{P}(\texttt{true}) = 0.90, \quad \hat{P}(\texttt{false}) = 0.10$$

We'd say the model is 90% confident in its answer of `true`.

For longer, multi-token answers[98]—like full sentences or paragraphs—it's harder to define confidence in a reliable way. There's no perfect rule. In our own work, we've tried using the average log probability per token, which gives us a rough sense of how confident the model was, on average, across its entire output. We've also experimented with summing the total log probability of the whole sequence, which reflects the model's overall certainty but tends to penalize longer responses. These are just heuristics, and their usefulness varies depending on the task. In practice, we've had to validate and calibrate them against actual evaluation data to see if they correlate with correctness in a meaningful way.

[98] This is an active area of research. We do not recommend using any of these methods without significant experimentation.

Building and Tuning the Cascade. Once we can assign confidence scores to proxy responses, we need to decide when to trust them. This means choosing thresholds: how confident does the proxy need to be for us to accept its answer, rather than escalating to the oracle?

Algorithm 2: FindThresholdsForCascade

Input: Dataset $D = \{(x_i, y_i)\}$ of inputs and gold labels from oracle or human; Proxy model M_p; Oracle model M_o; Accuracy target α
Output: Per-class thresholds T_k

Run proxy model M_p on each x_i to obtain prediction k_i and confidence score C_i (e.g., normalized prob of chosen token).
Run oracle model M_o on each x_i to obtain gold label y_i (if not already known).
Group each (C_i, y_i, k_i) by predicted class k.
foreach *class* k **do**
    Sort all proxy outputs with $k_i = k$ by confidence C_i in ascending order.
    foreach *candidate threshold* t *in the sorted list* **do**
        Simulate the cascade: use proxy prediction when $C_i \geq t$, else use oracle's label.
        Compute resulting accuracy A_t on this subset.
        if $A_t \geq \alpha$ **then**
            Set $T_k = t$ and break.
        end
    end
    if *no threshold* t *satisfies the target* **then**
        Set $T_k = \infty$ (i.e., always use oracle for class k).
    end
end
return $\{T_k\}$

Algorithm 2 describes a greedy algorithm for setting these thresholds. For classification tasks, we tune a separate threshold for each predicted class. We sort examples by the proxy's confidence and find the lowest threshold that ensures the cascade's output matches the oracle's output at a desired rate (e.g., α=0.95). If no such threshold exists for a class, we conservatively escalate all predictions of that class to the oracle.

For generative tasks, we don't have discrete class labels. Instead, we can treat all examples as a single group and evaluate the proxy's predictions against the oracle's output using a binary scoring function (e.g., an LLM-as-Judge that asks, "Does the proxy answer convey the same information as the oracle answer?"). We then find a single global threshold that achieves our desired match rate with the oracle. The selection of thresholds and the resulting cost/accuracy trade-offs are always made with respect to matching the oracle, not exceeding it.

While Algorithm 2 is not globally optimal—it tunes thresholds independently per class and does not consider their interactions—it is simple, efficient, and effective in practice. The approach generalizes naturally to multi-stage cascades: assign thresholds to each proxy model independently, then order models by increasing cost. At runtime, the system evaluates each model in sequence, stopping at the first whose confidence exceeds its threshold. More sophisticated strategies may train a learned

router to select among models (Chen et al.), but threshold tuning remains a robust and interpretable baseline.

11.5 Summary

Improving LLM pipelines means two things: making them more accurate and making them more cost-efficient. In this chapter, we covered practical ways to do both.

For accuracy, we started with quick wins like refining prompts and adding examples. We then moved to structural changes like breaking up complex tasks, tuning retrieval steps, and fixing how the model uses tools. When those aren't enough, we discussed heavier options like fine-tuning or setting up human review loops.

For cost, we looked at common ways to save money without sacrificing quality. These included using smaller models for simple tasks, cutting down on token usage, and making prompts more cache-friendly. We also covered model cascades—an advanced way to balance cost and accuracy by routing only hard cases to expensive models.

None of these changes work in isolation. To actually improve our LLM pipelines, we need good evaluation data and monitoring. Every change should be measured. And once it works, it should be integrated into our CI pipeline so that improvements stick.

The Improve phase never ends, as it is part of the evaluation lifecycle. As models, data, and use cases change, so should our pipelines. But with the right tools and a structured approach, we can keep improving without blowing our budget.

References

Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019. ↑123

Anthropic. Create strong empirical evaluations. https://docs.anthropic.com/en/docs/build-with-claude/develop-tests, 2024. Accessed: 2025-05-12. ↑22

Ian Arawjo. Evalgen: Helping developers create LLM evals aligned to their preferences. Medium, May 2025a. URL https://ianarawjo.medium.com/evalgen-helping-developers-create-llm-evals-aligned-to-their-preferences-26757f7e145d. "EvalGen v2". ↑136, ↑137

Ian Arawjo. What AI engineers can learn from qualitative research methods in HCI, January 2025b. URL https://ianarawjo.medium.com/what-ai-engineers-can-learn-from-qualitative-research-methods-in-hci-5b29b9b7465a. Medium blog post. ↑27

Ron Artstein and Massimo Poesio. Inter-coder agreement for computational linguistics. *Computational linguistics*, 34(4):555–596, 2008. ↑33

John Berryman, Shawn Simister, and Hamel Husain. How evals made GitHub Copilot work. https://maven.com/p/da8264/how-evals-made-github-copilot-work, 2025. Accessed: 2025-05-12. ↑27

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021. ↑4

Florian Bordes, Richard Yuanzhe Pang, Anurag Ajay, Alexander C Li, Adrien Bardes, Suzanne Petryk, Oscar Mañas, Zhiqiu Lin, Anas Mahmoud, Bargav Jayaraman, et al. An introduction to vision-language modeling. *arXiv preprint arXiv:2405.17247*, 2024. ↑110

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. The ML test score: A rubric for ML production readiness and technical debt reduction. In *2017 IEEE international conference on big data (big data)*, pages 1123–1132. IEEE, 2017. ↑105, ↑123

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. ↑11

Boyuan Chen, Zhuo Xu, Sean Kirmani, Brain Ichter, Dorsa Sadigh, Leonidas Guibas, and Fei Xia. SpatialVLM: Endowing vision-language models with spatial reasoning capabilities. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14455–14465, 2024a. ↑110

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*. ↑153

Lingjiao Chen, Matei Zaharia, and James Zou. How is ChatGPT's behavior changing over time? *Harvard Data Science Review*, 6(2), 2024b. ↑70

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. ↑68

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing GPT-4 with 90%* ChatGPT quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/. ↑33

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. ↑16

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960. ↑42

Samuel Colvin. Pydantic, 2017. URL https://github.com/pydantic/pydantic. Accessed: 2025-05-03. ↑105

Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang Zhu, and Saleema Amershi. Interactive debugging and steering of multi-agent AI systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2025. ↑123

Shahul Es, Jithin James, Luis Espinosa Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval augmented generation. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, pages 150–158, 2024. ↑91

Tomas Fernandez. Handling flaky tests in llm-powered applications, April 2024. URL https://semaphore.io/blog/llms-flaky-tests. Accessed: 2025-05-04. ↑124

Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971. ↑44

Clémentine Fourrier and The Hugging Face Community. Llm evaluation guidebook, 2024. URL https://github.com/huggingface/evaluation-guidebook). ↑54

Barney Glaser and Anselm Strauss. *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017. ↑26, ↑29

Kristina Gligorić, Tijana Zrnic, Cinoo Lee, Emmanuel J Candès, and Dan Jurafsky. Can unconfident LLM annotations be used for confident conclusions? *arXiv preprint arXiv:2408.15204*, 2024. ↑71

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020. ↑11

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ. ↑4, ↑16

Fred Hohman, Chaoqun Wang, Jinmook Lee, Jochen Görtler, Dominik Moritz, Jeffrey P Bigham, Zhile Ren, Cecile Foret, Qi Shan, and Xiaoyi Zhang. Talaria: Interactively optimizing machine learning models for efficient inference. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2024. ↑123

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy. ↑111

Hamel Husain. A field guide to rapidly improving AI products, March 2025. URL https://hamel.dev/blog/posts/field-guide/. Blog post. ↑22, ↑33

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL. ↑4, ↑68

Kalervo Järvelin and Jaana Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *ACM SIGIR Forum*, volume 51, pages 243–250. ACM New York, NY, USA, 2017. ↑89

Adam Tauman Kalai and Santosh S Vempala. Calibrated language models must hallucinate. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pages 160–171, 2024. ↑7, ↑12

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. 2024. ↑62

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, et al. Prometheus: Inducing fine-grained evaluation capability in language models. In *The Twelfth International Conference on Learning Representations*, 2023. ↑33

Klaus Krippendorff. Computing Krippendorff's alpha-reliability, 2011. ↑44

J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977. ↑43

LangSmith by LangChain, Inc. Langsmith. https://www.langchain.com/langsmith. Accessed: 2025-05-04. ↑107

Quentin Romero Lauro, Shreya Shankar, Sepanta Zeighami, and Aditya Parameswaran. Rag without the lag: Interactive debugging for retrieval-augmented generation pipelines. *arXiv preprint arXiv:2504.13587*, 2025. ↑85, ↑123

Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International conference on machine learning*, pages 12888–12900. PMLR, 2022. ↑110

Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060*, 2024. ↑12

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D Manning, Christopher Re, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue WANG, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Andrew Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=iO4LZibEqW. Featured Certification, Expert Certification, Outstanding Certification. ↑4, ↑16

Yiming Lin, Mawil Hasan, Rohan Kosalge, Alvin Cheung, and Aditya G Parameswaran. Twix: Automatically reconstructing structured data from templatized documents. *arXiv preprint arXiv:2501.06659*, 2025. ↑113

Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, 2025. ↑113

Michael Xieyang Liu, Frederick Liu, Alexander J Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J Cai. " we need structured output": Towards user-centered constraints on large language model output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, pages 1–9, 2024a. ↑16, ↑28

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12, 2024b. ↑89, ↑111

Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing LLM queries in relational workloads. *arXiv preprint arXiv:2403.05821*, 2024c. ↑113

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the*

2022 Conference on Empirical Methods in Natural Language Processing, pages 11048–11064, 2022. ↑61

Janice M Morse. The significance of saturation, 1995. ↑23, ↑28

Sankalp Nagaonkar, Augustya Sharma, Ashish Choithani, and Ashutosh Trivedi. Benchmarking vision-language models on optical character recognition in dynamic video environments. *arXiv preprint arXiv:2502.06445*, 2025. ↑110

Preetum Nakkiran, Arwen Bradley, Hattie Zhou, and Madhu Advani. Step-by-step diffusion: An elementary tutorial. *Found. Trends. Comput. Graph. Vis.*, 17(1):1–75, April 2025. ISSN 1572-2740. DOI: 10.1561/0600000114. URL https://doi.org/10.1561/0600000114. ↑110

Jakob Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 152–158, Boston, MA, 1994. ACM. ↑134, ↑135

Donald A. Norman. *The Design of Everyday Things*. Basic Books, New York, 1988. ↑5

Donald A. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, New York, NY, 2013. ↑135

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. 2021. ↑11

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366, Miami, Florida, USA, November 2024. Association for Computational Linguistics. DOI: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/. ↑147

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022. ↑123

Letitia Parcalabescu, Michele Cafagna, Lilitta Muradjan, Anette Frank, Iacer Calixto, and Albert Gatt. Valse: A task-independent benchmark for vision and language models centered on linguistic phenomena. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8253–8280, 2022. ↑110

Shishir G Patil, Tianjun Zhang, Vivian Fang, Roy Huang, Aaron Hao, Martin Casado, Joseph E Gonzalez, Raluca Ada Popa, Ion Stoica, et al. Goex: Perspectives and designs towards a runtime for autonomous LLM applications. *arXiv preprint arXiv:2404.06921*, 2024. ↑104

Rodrigo Pedro, Daniel Castro, Paulo Carreira, and Nuno Santos. From prompt injections to sql injection attacks: How protected is your llm-integrated web application? *arXiv preprint arXiv:2308.01990*, 2023. ↑105

Phoenix by Arize AI. Phoenix: Open-source LLM tracing & evaluation. https://phoenix.arize.com/. Accessed: 2025-05-04. ↑107

Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. Data validation for machine learning. *Proceedings of machine learning and systems*, 1:334–347, 2019. ↑105

Jing Qian, Hong Wang, Zekun Li, Shiyang Li, and Xifeng Yan. Limitations of language models in arithmetic and symbolic induction. *arXiv preprint arXiv:2208.05051*, 2022. ↑11

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, 2019. ↑84

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024. ↑4

Walter J Rogan and Beth Gladen. Estimating prevalence from the results of a screening test. *American journal of epidemiology*, 107(1):71–76, 1978. ↑64

Jon Saad-Falcon, Omar Khattab, Christopher Potts, and Matei Zaharia. ARES: An automated evaluation framework for retrieval-augmented generation systems. In Kevin Duh, Helena Gomez, and Steven Bethard, editors, *Proceedings of the 2024 Conference of the*

North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pages 338–354, Mexico City, Mexico, June 2024. Association for Computational Linguistics. DOI: 10.18653/v1/2024.naacl-long.20. URL https://aclanthology.org/2024.naacl-long.20/. ↑91

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023. ↑11, ↑104

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=RIu5lyNXjT. ↑12

David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015. ↑71, ↑123

Burr Settles. Active learning literature survey. 2009. ↑137

Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G Parameswaran, and Eugene Wu. Docetl: Agentic query rewriting and evaluation for complex document processing. *arXiv preprint arXiv:2410.12189*, 2024a. ↑111, ↑112

Shreya Shankar, Rolando Garcia, Joseph M Hellerstein, and Aditya G Parameswaran. " we have no idea how models will behave in production until production": How engineers operationalize machine learning. *Proceedings of the ACM on Human-Computer Interaction*, 8(CSCW1): 1–34, 2024b. ↑71, ↑123

Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, JD Zamfirescu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G Parameswaran, and Eugene Wu. SPADE: Synthesizing data quality assertions for large language model pipelines. *Proceedings of the VLDB Endowment*, 17(12):4173–4186, 2024c. ↑16, ↑33

Shreya Shankar, JD Zamfirescu-Pereira, Björn Hartmann, Aditya Parameswaran, and Ian Arawjo. Who validates the validators? aligning llm-assisted evaluation of LLM outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–14, 2024d. ↑6, ↑7, ↑70, ↑133, ↑136

Shreya Shankar, Bhavya Chopra, Mawil Hasan, Stephen Lee, Björn Hartmann, Joseph M Hellerstein, Aditya G Parameswaran, and Eugene Wu. Steering semantic data processing with docwrangler. *arXiv preprint arXiv:2504.14764*, 2025. ↑5, ↑6, ↑142, ↑143

Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Longman Publishing Co., Inc., USA, 3rd edition, 1997. ISBN 0201694972. ↑135

Anselm Strauss, Juliet Corbin, et al. *Basics of qualitative research*, volume 15. sage Newbury Park, CA, 1990. ↑27, ↑29

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena : A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=qVyeW-grC2k. ↑111

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 2 edition, 2001. ↑135

Reya Vir, Shreya Shankar, Harrison Chase, William Hinthorn, and Aditya Parameswaran. PROMPTEVALS: A dataset of assertions and guardrails for custom production large language model pipelines. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4225–4245, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. URL https://aclanthology.org/2025.naacl-long.213/. ↑28

Emma Ward and Steven Feldstein. Evaluating large language models. *Center for Security and Emerging Technology*, July 2024. URL https://cset.georgetown.edu/article/evaluating-large-language-models/. ↑4

Emily Webber and Andrea Olgiati. *Pretrain Vision and Large Language Models in Python: End-to-end techniques for building and deploying foundation models on AWS*. Packt Publishing Ltd, 2023. ↑110

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022. ↑11

Weights & Biases. Weave: A toolkit for developing generative AI applications. https://github.com/wandb/weave. Accessed: 2025-05-04. ↑107

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *International Conference on Machine Learning*, pages 11080–11090. PMLR, 2021. ↑11

Ziyou Yan. Evaluation & hallucination detection for abstractive summaries. *eugeneyan.com*, Sep 2023. URL https://eugeneyan.com/writing/abstractive/. ↑15

Ziyou Yan. Evaluating the effectiveness of llm-evaluators (aka llm-as-judge). *eugeneyan.com*, Aug 2024. URL https://eugeneyan.com/writing/llm-evaluators/. ↑33

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X. ↑106, ↑108

Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound AI systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024. ↑4

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023. ↑33, ↑57, ↑58

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Joshua M. Susskind, Samy Bengio, and Preetum Nakkiran. What algorithms can transformers learn? a study in length generalization. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=AssIuHnmHX. ↑11

Yiyang Zhou, Chenhang Cui, Jaehong Yoon, Linjun Zhang, Zhun Deng, Chelsea Finn, Mohit Bansal, and Huaxiu Yao. Analyzing and mitigating object hallucination in large vision-language models. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=oZDJKTlOUe. ↑110