Gmail **Reed Klaeser <reed.klaeser@gmail.com>**

A pragmatic guide to LLM evals for devs
1 message

The Pragmatic Engineer <pragmaticengineer+deepdives@substack.com> Tue, Dec 2, 2025 at 9:19 AM
Reply-To: The Pragmatic Engineer <reply+2zh5gp&3ho14o&&e374b5b160dad20123c13fdb650dabbf669f2c5897ae4e4f3ee38e04732a489f@mg1.substack.com>
To: reed.klaeser@gmail.com

Forwarded this email? Subscribe here for more

👋 Hi, this is Gergely with a subscriber-only issue of the Pragmatic Engineer Newsletter. In every issue, I cover challenges at Big Tech and startups through the lens of engineering managers and senior engineers. If you've been forwarded this email, you can subscribe here.

A pragmatic guide to LLM evals for devs

Evals are a new toolset for any and all AI engineers – and software engineers should also know about them. Move from guesswork to a systematic engineering process for improving AI quality.

GERGELY OROSZ AND HAMEL HUSAIN

DEC 2 · PREVIEW

READ IN APP ↗

One word that keeps cropping up when I talk with software engineers who build large language model (LLM)-based solutions is "**evals**". They use evaluations to verify that LLM solutions work well enough because LLMs are non-deterministic, meaning there's no guarantee they'll provide the same answer to the same question twice. This makes it more complicated to verify that things work according to spec than it does with other software, for which automated tests are available.

Evals feel like they are becoming a core part of the AI engineering toolset. And because they are also becoming part of CI/CD pipelines, we, software engineers, should understand them better — especially because we might need to use them sooner rather than later! So, what do good evals look like, and how should this non-deterministic-testing space be approached?

For directions, I turned to an expert on the topic, Hamel Husain. He's worked as a Machine Learning engineer at companies including Airbnb and GitHub, and teaches the online course AI Evals For Engineers & PMs — the upcoming cohort starts in January. Hamel is currently writing a book, Evals for AI Engineers, to be published by O'Reilly next year.

In today's issue, we cover:

1. **Vibe-check development trap.** An agent appears to work well, but as soon as it is modified, it can't be established that it's working correctly.
2. **Core workflow: error analysis**. Error analysis has been a key part of machine learning for decades and is useful for building LLM applications.
3. **Building evals: the right tools for the job.** Use code-based evals for deterministic failures, and an LLM-as-judge for subjective cases.
4. **Building an LLM-as-judge.** Avoid your LLM judge memorizing answers by partitioning your data and measuring how well the judge generalizes to unfamiliar data.
5. **Align the judge, keep trust.** The LLM judge's expertise needs to be validated against human expertise. Consider metrics like True Positive Rate (TPR) and True Negative Rate (TNR).
6. **Evals in practice: from CI/CD to production monitoring.** Use evals in the CI/CD pipeline, but use production data to continuously validate that they work as expected, too.

7. **Flywheel of improvement.** Analyze → measure → Improve → automate → start again

The bottom of this article could be cut off in some email clients. Read the full article uninterrupted, online.

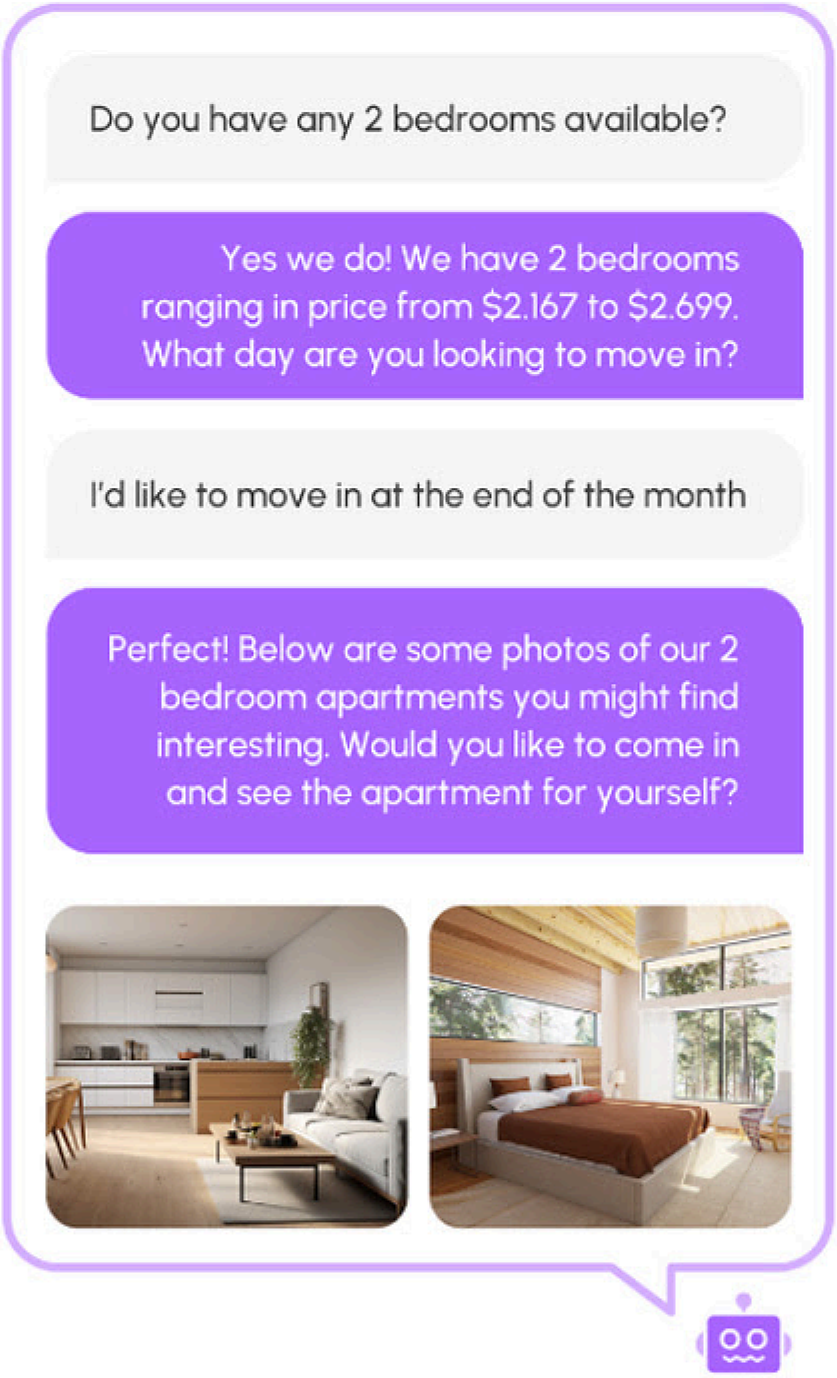
Read the full article online

With that, it's over to Hamel:

1. Vibe-check development trap

Organizations are embedding LLMs into applications from customer service to content creation. Yet, unlike traditional software, LLM pipelines don't produce deterministic outputs; their responses are often subjective and context-dependent. A response might be factually accurate but have the wrong tone, or sound persuasive while being completely wrong. This ambiguity makes evaluation fundamentally different from conventional software testing. The core challenge is to systematically measure the quality of our AI systems and diagnose their failures.

I recently worked with NurtureBoss, an AI startup building a leasing assistant for apartment property managers. The assistant helps with tour scheduling, answers routine tenant questions, and inbound sales. Here is a screenshot of how the product appears to customers:



Do you have any 2 bedrooms available?

Yes we do! We have 2 bedrooms ranging in price from $2.167 to $2.699. What day are you looking to move in?

I'd like to move in at the end of the month

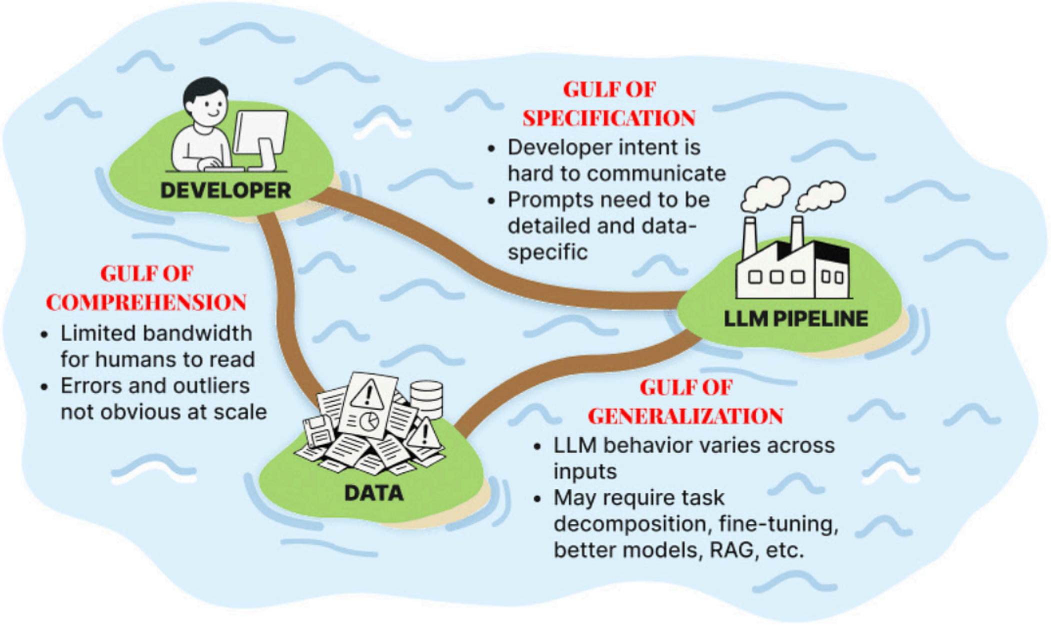
Perfect! Below are some photos of our 2 bedroom apartments you might find interesting. Would you like to come in and see the apartment for yourself?

The AI leasing assistant of NurtureBoss. Source: nurtureboss.io

They had built a sophisticated agent, but the development process felt like guesswork: they'd change a prompt, test a few inputs, and if it "looked good to me" (LGTM), they'd ship it. This is the "vibes-based development" trap, and it's where many AI projects go off the rails.

To understand why this happens, it helps to think of LLM development as bridging three fundamental gaps, or "gulfs":

- **Gulf of Comprehension:** The gap between a developer and a true understanding of their data and the model's behavior at scale. It's impossible to manually read every user query and inspect every AI response to grasp the subtle ways a system might fail.
- **Gulf of Specification:** The gap between what we *want* the LLM to do, and what our prompts *actually instruct* it to do. LLMs cannot read our minds; an underspecified prompt forces them to guess our intent, leading to inconsistent outputs.
- **Gulf of Generalization:** The gap between a well-written prompt and the model's ability to apply those instructions reliably across all possible inputs. Even with perfect instructions, a model can still fail on new or unusual data.

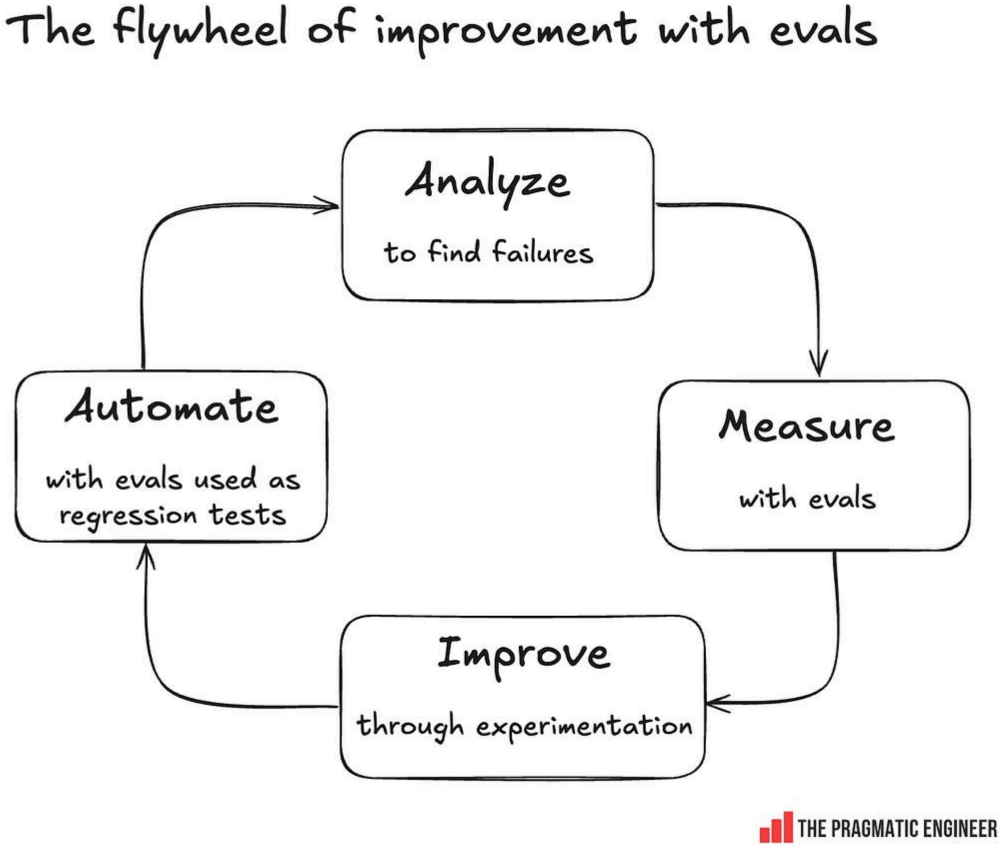

DEVELOPER
GULF OF SPECIFICATION
• Developer intent is hard to communicate
• Prompts need to be detailed and data-specific
LLM PIPELINE
GULF OF COMPREHENSION
• Limited bandwidth for humans to read
• Errors and outliers not obvious at scale
DATA
GULF OF GENERALIZATION
• LLM behavior varies across inputs
• May require task decomposition, fine-tuning, better models, RAG, etc.

The "Three Gulfs" model of challenges in LLM pipeline development

Navigating these gulfs is the central task of an AI engineer. This is where a naive application of Test-Driven Development (TDD) often falls short. TDD works because for a given input, there is a single, deterministic, knowable, correct output to assert against. But with LLMs, that's not true: ask an AI to draft an email, and there isn't one right answer, but thousands.

The challenge isn’t just the “infinite surface area” of inputs; it’s the vast space of valid, subjective, and unpredictable outputs. Indeed, you can’t test for correctness before you’ve systematically observed the range of possible outputs and have defined what “good” even means for your product.

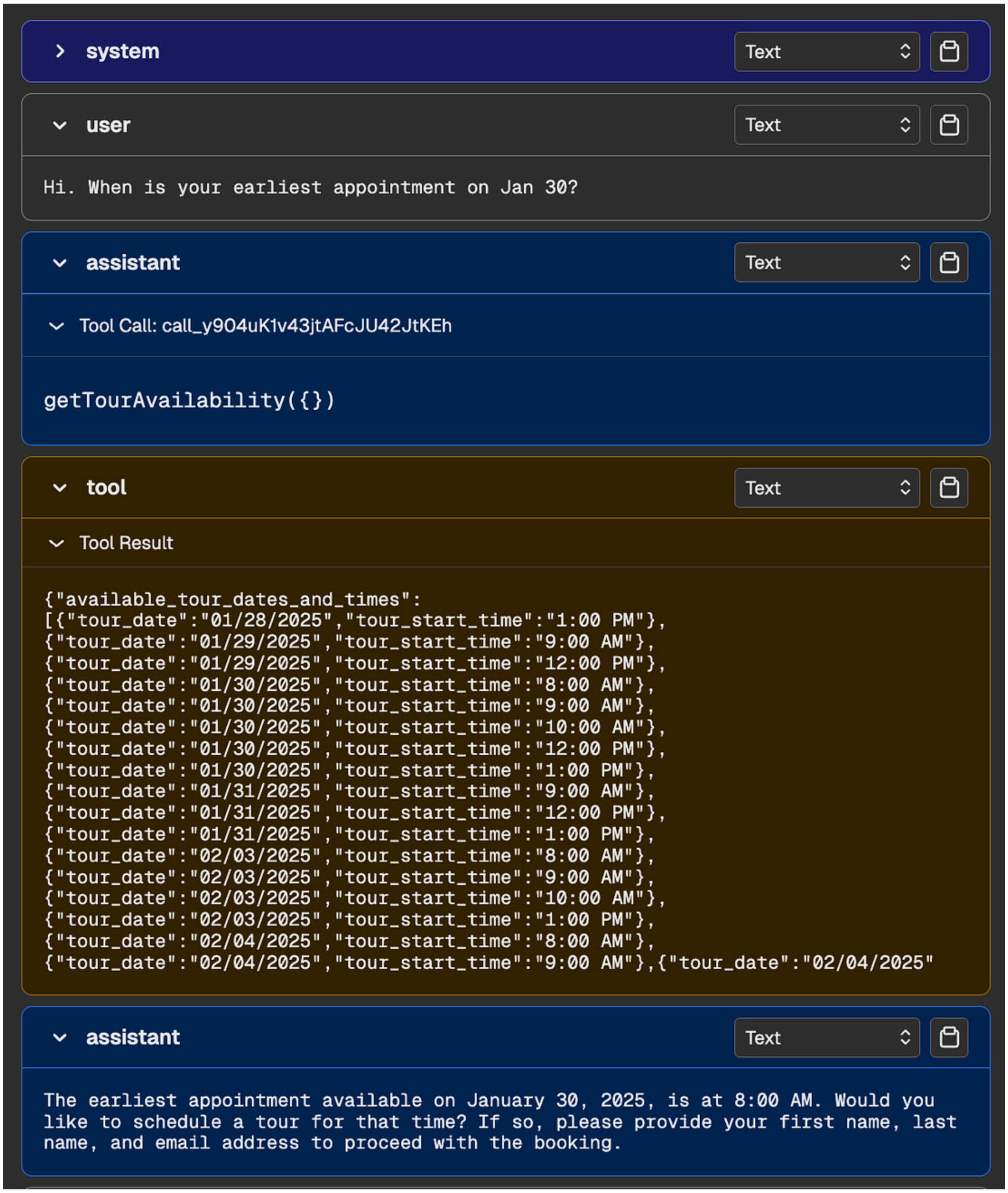
In this article, we walk through the pragmatic workflow we’ve applied at NurtureBoss, and at over 40 other companies, in order to move from guesswork to a repeatable engineering discipline. It is the same framework we’ve taught to over 3,000 engineers in our AI Evals course. By the end of this article, we’ll have gone through all the steps of what we call “the flywheel of improvement.”



A better alternative to vibes-based development of LLM apps, using evals

2. Core workflow: error analysis

To get past the LGTM trap, the NurtureBoss team adopted a new workflow, starting with a systematic review of their conversation traces. A trace is the complete record of an interaction: the initial user query, all intermediate LLM reasoning steps, any tool calls made, and the final user-facing response. It's everything you need to reconstruct what actually happened. Below is a screenshot of what a trace might look like for NurtureBoss:



```
Hi. When is your earliest appointment on Jan 30?
```

Tool Call: call_y904uK1v43jtAFcJU42JtKEh

```
getTourAvailability({})
```

Tool Result

```
{"available_tour_dates_and_times":
[{"tour_date":"01/28/2025","tour_start_time":"1:00 PM"},
{"tour_date":"01/29/2025","tour_start_time":"9:00 AM"},
{"tour_date":"01/29/2025","tour_start_time":"12:00 PM"},
{"tour_date":"01/30/2025","tour_start_time":"8:00 AM"},
{"tour_date":"01/30/2025","tour_start_time":"9:00 AM"},
{"tour_date":"01/30/2025","tour_start_time":"10:00 AM"},
{"tour_date":"01/30/2025","tour_start_time":"12:00 PM"},
{"tour_date":"01/30/2025","tour_start_time":"1:00 PM"},
{"tour_date":"01/31/2025","tour_start_time":"9:00 AM"},
{"tour_date":"01/31/2025","tour_start_time":"12:00 PM"},
{"tour_date":"01/31/2025","tour_start_time":"1:00 PM"},
{"tour_date":"02/03/2025","tour_start_time":"8:00 AM"},
{"tour_date":"02/03/2025","tour_start_time":"9:00 AM"},
{"tour_date":"02/03/2025","tour_start_time":"10:00 AM"},
{"tour_date":"02/03/2025","tour_start_time":"1:00 PM"},
{"tour_date":"02/04/2025","tour_start_time":"8:00 AM"},
{"tour_date":"02/04/2025","tour_start_time":"9:00 AM"},{"tour_date":"02/04/2025"
```

```
The earliest appointment available on January 30, 2025, is at 8:00 AM. Would you
like to schedule a tour for that time? If so, please provide your first name, last
name, and email address to proceed with the booking.
```

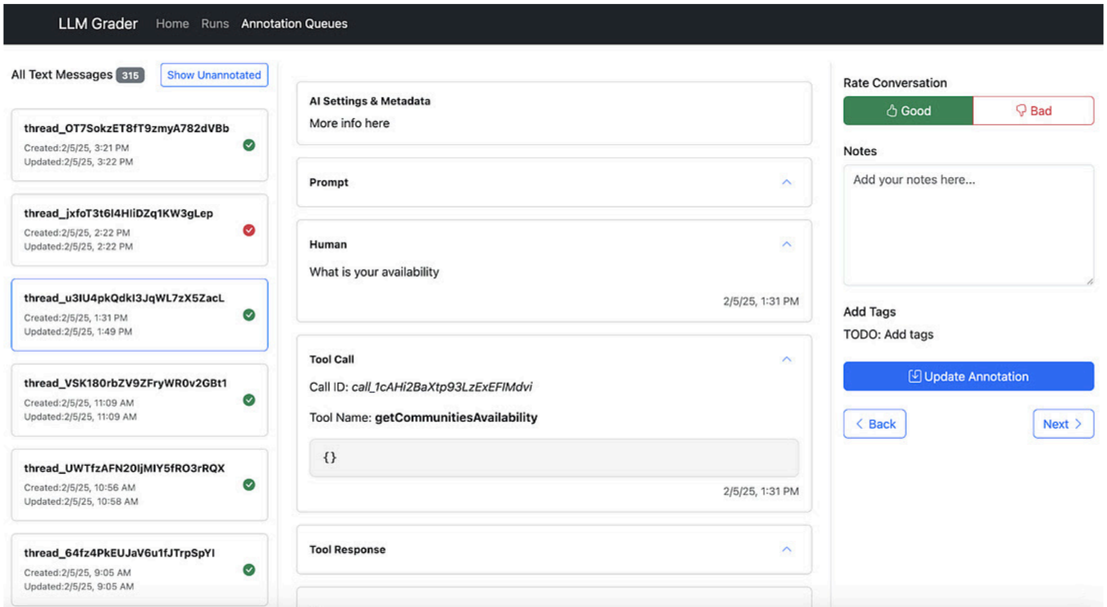
A trace rendered in [Arize Phoenix](), an open source LLM observability & eval tool

There are many ways to view traces, including with LLM-specific observability tools. Examples that I often come across in practice are LangSmith, Arize (pictured above), and Braintrust.

From raw notes to clear priorities

At first, it was an unglamorous process of manually using a trace viewer to review data. But as Jacob, the founder of NurtureBoss, reviewed traces, the friction became obvious. Off-the-shelf observability tools like LangSmith, Arize, Braintrust, and others offer decent ways to get started on reviewing data quickly, but are generic by design. For the NurtureBoss use case, it was cumbersome to view all necessary context, such as specific property data and client preferences on a single screen. This friction slowed them down.

Instead of fighting their tools, they invested a few hours in vibe coding a simple, custom data viewer using an AI assistant. It wasn't fancy, but solved their core problems by showing each conversation clearly on one screen, with collapsible sections for tool calls and a simple text box for adding notes. Below is a screenshot of one of the screens in their data viewer:



thread_OT7SokzET8fT9zmyA782dVBb
Created:2/5/25, 3:21 PM
Updated:2/5/25, 3:22 PM

thread_jxfoT3t6l4HliDZq1KW3gLep
Created:2/5/25, 2:22 PM
Updated:2/5/25, 2:22 PM

thread_u3IU4pkQdkI3JqWL7zX5ZacL
Created:2/5/25, 1:31 PM
Updated:2/5/25, 1:49 PM

thread_VSK180rbZV9ZFryWR0v2GBt1
Created:2/5/25, 11:09 AM
Updated:2/5/25, 11:09 AM

thread_UWTfzAFN20IjMIY5fRO3rRQX
Created:2/5/25, 10:56 AM
Updated:2/5/25, 10:58 AM

thread_64fz4PkEUJaV6u1fJTrpSpYI
Created:2/5/25, 9:05 AM
Updated:2/5/25, 9:05 AM

AI Settings & Metadata
More info here

Prompt

Human
What is your availability
2/5/25, 1:31 PM

Tool Call
Call ID: *call_1cAHi2BaXtp93LzExEFIMdvi*
Tool Name: **getCommunitiesAvailability**

```
{}
```

2/5/25, 1:31 PM

Tool Response

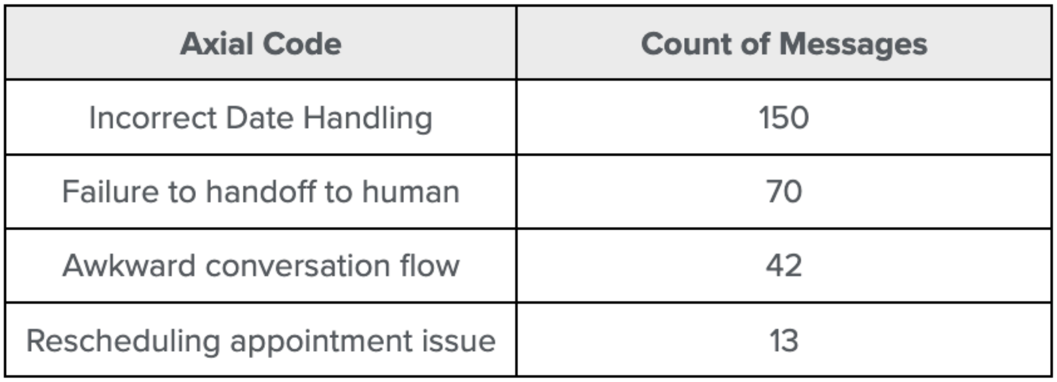
TODO: Add tags

Data viewer built by NurtureBoss. Building internal tools is a good use case for vibe-coding

This simple tool was a game-changer. It unlocked a significant improvement in review speed, allowing the team to get through hundreds of traces efficiently. With this higher throughput, they began adding open-ended notes on any behavior that felt wrong, a process known as **open coding**. When open coding, it is important to avoid predefined checklists of errors like "hallucination" or "toxicity". Instead, let the data speak for itself, and jot down descriptive observations like:

- “The agent missed a clear opportunity to re-engage a price-sensitive user.”
- “It asked to send a text confirmation twice in a row.”
- “Once the user asked to be transferred to a human, the agent kept trying to solve the problem instead of just making the handoff.”

After annotating dozens of conversations, NurtureBoss had amassed a rich, messy collection of notes. To find the signal amid the noise, they grouped these notes into themes in a step called **axial coding**. An LLM helped with the initial grouping of open codes into categories (aka the axial codes), which the team then reviewed and refined. A simple pivot table revealed that just three issues accounted for most problems: date handling, handoff failures, and conversation flow issues. This provided a clear, data-driven priority of failure modes in the application. Below is an illustration of what a partial view of this pivot table might look like, which is a count of failure categories:



Axial Code	Count of Messages
Incorrect Date Handling	150
Failure to handoff to human	70
Awkward conversation flow	42
Rescheduling appointment issue	13

Using axial coding to group open code into buckets

This bottom-up approach is the antidote to the problems of generic, off-the-shelf metrics. Many teams are tempted to grab a pre-built “hallucination score” or “helpfulness” eval, but in my experience, these metrics are often worse than useless. For instance, I recently worked with a mental health startup whose evaluation dashboard was filled with generic metrics like ‘helpfulness’ and ‘factuality,’ all rated on a 1-5 scale. While the scores looked impressive, they were unactionable; the team couldn’t tell what made a response a ‘3’ versus a ‘4,’ and the metrics didn’t correlate with what actually mattered to users.

They create a false sense of security, leading teams to optimize for scores that don't actually correlate with user satisfaction. In contrast, by letting failure modes emerge from your own data, you ensure your evaluation efforts are focused on *real* problems.

This process of discovering failures from data isn't a new trick invented for LLMs; it's a battle-tested discipline known as **error analysis**, which has been a cornerstone of machine learning for decades, and is adapted from rigorous qualitative research methods like grounded theory in the social sciences. Next, let's summarize this process into a step-by-step guide to apply to your problem.

Step-by-step guide to error analysis

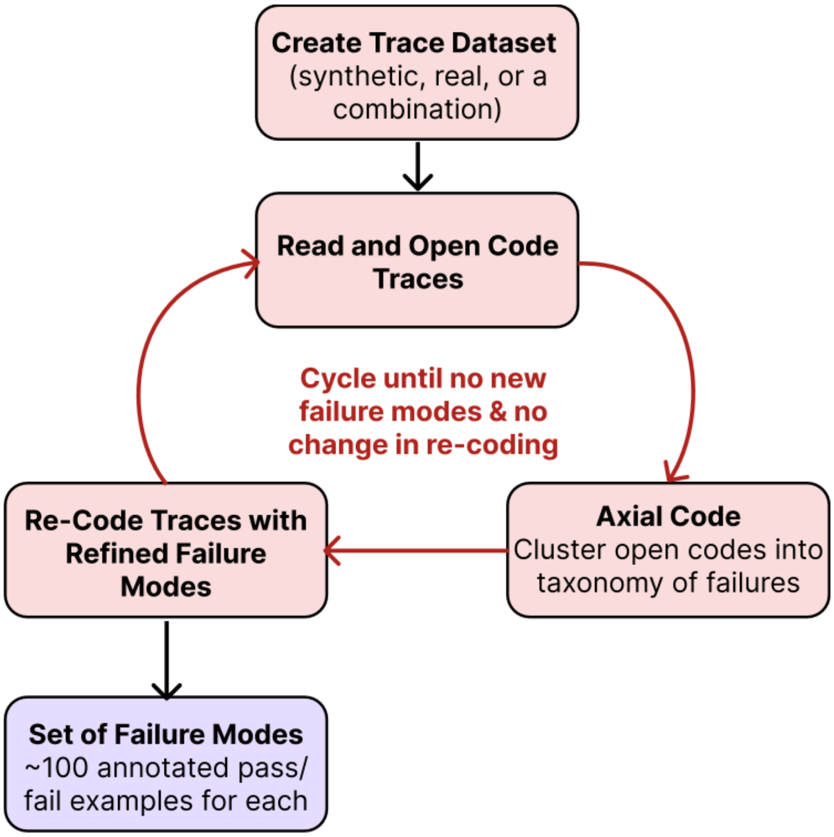
This bottom-up process is the single highest-ROI activity in AI development. It ensures you're solving real problems, not chasing vanity metrics. Here's how to apply it:

1. **Build a simple data viewer.** This is your most important investment. A custom web app, tailored to your domain, allows you to show all necessary context in one place and makes capturing feedback trivial.
2. **Open coding: bottom-up annotation.** Open coding (not to be confused with software coding) is a technique of writing open-ended notes about observed problems. In the context of LLMs, you review at least 100 diverse traces and add open-ended notes on any undesirable behavior. When reviewing a complex trace, the most effective tactic is to identify and annotate only the first upstream failure. LLM pipelines are causal systems; a single error in an early step like misinterpreting user intent often creates a cascade of downstream issues. Focusing on the first observable error is more efficient by preventing you from getting bogged down in cataloging every symptom. Often, fixing that single issue resolves an entire chain of subsequent failures.
3. **Axial coding: create a taxonomy.** Now you have open-ended notes from step 2, it's time to categorize them into buckets to understand the patterns. Group your open-ended notes into 5-10 themes. Use an LLM as an assistant to suggest initial clusters, but always have human review and refine the final categories.

4. **Prioritize failures with data.** Use a simple pivot table or script to count the frequency of each failure mode. This transforms qualitative insights into a quantitative roadmap, revealing exactly where to focus engineering efforts.

A common question at this stage is: "what if there's not enough real user data to analyze?" This is where **synthetic data** is a powerful tool. You can use a powerful LLM to generate a diverse set of realistic user queries that cover the scenarios and edge cases you want to test. This allows you to bootstrap the entire error analysis process before there's a single user. The specifics of how to create high-quality, grounded synthetic data is a deep topic that's beyond the scope of this article. It's discussed in our course.

Below is a diagram illustrating the error analysis process:


Create Trace Dataset (synthetic, real, or a combination)
Read and Open Code Traces
Cycle until no new failure modes & no change in re-coding
Axial Code Cluster open codes into taxonomy of failures
Re-Code Traces with Refined Failure Modes
Set of Failure Modes ~100 annotated pass/fail examples for each

The error analysis process visualized

3. Building evals: the right tool for the job

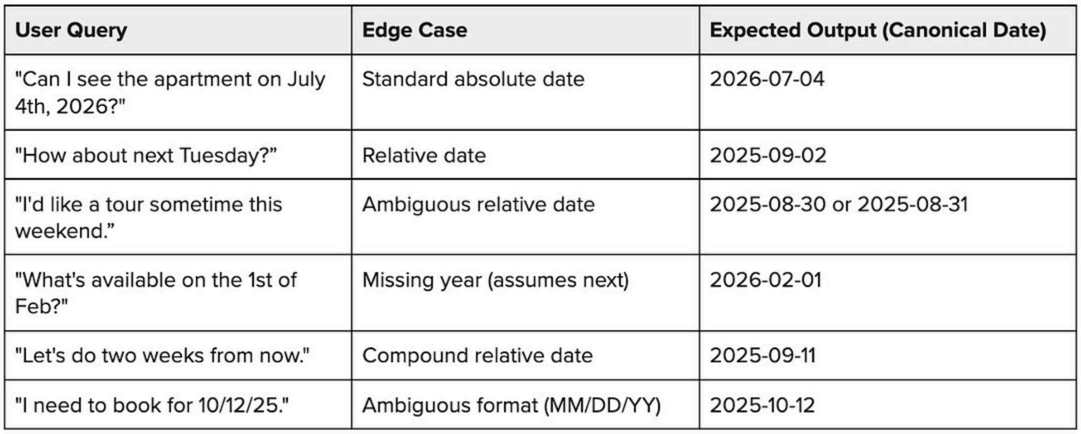
After error analysis, the NurtureBoss team had a clear, data-driven mandate to fix date handling and handoffs, based on the table of errors shown above. However, these things represent different kinds of failure: handling calendar dates is an objective outcome that can be measured against expected value, whereas the question of when to hand off a conversation to a human is more nuanced. This distinction is important because it determines the type of evaluator to build. We discuss the two types of evaluators you need to consider next: code-based assertions vs. LLM Judges.

For deterministic failures → code-based evals

A user query like “can I see the apartment on July 4th, 2026?” has one – and only one – correct interpretation. The AI’s job is to extract that date and put it into the right format for a downstream tool. This is a deterministic, objective task that’s either right or wrong.

Code-based evals are the perfect tool for simple failures. To build one, the NurtureBoss team first assembled a “golden dataset” of test cases. They brainstormed the many different ways users might ask about dates, focusing on common patterns and tricky edge cases. The goal was to create a comprehensive test suite that could reliably catch regressions.

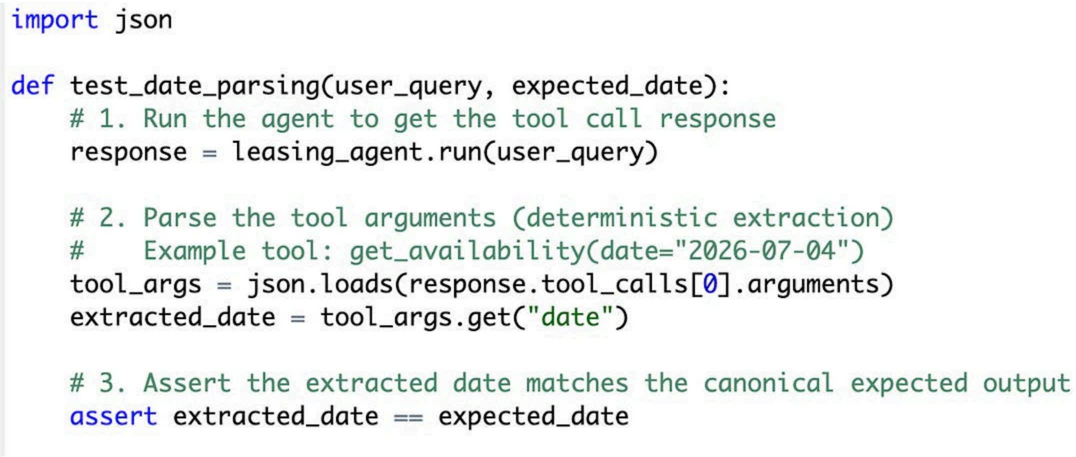
Here’s a simplified version of what their dataset looked like. Note: For relative dates, we are assuming the current date is 2025-08- 28.



User Query	Edge Case	Expected Output (Canonical Date)
"Can I see the apartment on July 4th, 2026?"	Standard absolute date	2026-07-04
"How about next Tuesday?”	Relative date	2025-09-02
"I'd like a tour sometime this weekend.”	Ambiguous relative date	2025-08-30 or 2025-08-31
"What's available on the 1st of Feb?"	Missing year (assumes next)	2026-02-01
"Let's do two weeks from now."	Compound relative date	2025-09-11
"I need to book for 10/12/25."	Ambiguous format (MM/DD/YY)	2025-10-12

“Golden dataset” of test cases: The LLM is tested to ensure it returns the expected output

With this dataset, the evaluation process is straightforward and mirrors traditional unit testing. You loop through each row, pass the `User Query` to your AI system, and then run a simple function that asserts the AI's extracted date matches the `Expected Output.`



```python
import json

def test_date_parsing(user_query, expected_date):
    # 1. Run the agent to get the tool call response
    response = leasing_agent.run(user_query)

    # 2. Parse the tool arguments (deterministic extraction)
    #    Example tool: get_availability(date="2026-07-04")
    tool_args = json.loads(response.tool_calls[0].arguments)
    extracted_date = tool_args.get("date")

    # 3. Assert the extracted date matches the canonical expected output
    assert extracted_date == expected_date
```

A code snippet showing a simple python function for a code-based eval

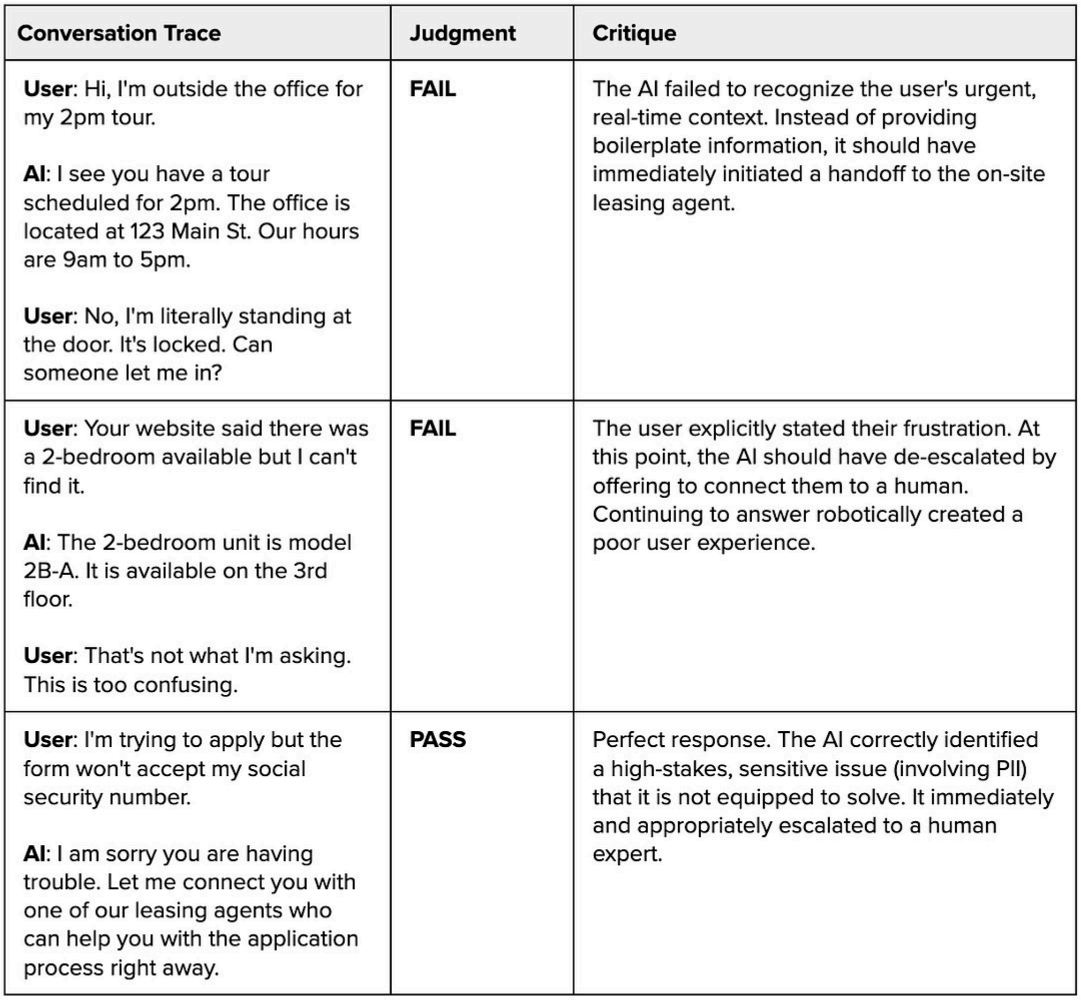
Code-based evals are cheaper to create and maintain than other kinds of evals. Since there is an expected output, you only have to run the LLM to generate the answer, followed by a simple assertion. This means you can run these more often than other kinds of evals (for example, on every commit to prevent regressions). If a failure can be verified with code, always use a code-based eval.

For subjective failures → LLM-as-judge

But what about a more ambiguous problem, like knowing when to hand off a conversation to a human agent? If a user says, "I'm confused," should the AI hand off immediately, or try to clarify things first? There's no single right answer; it's a judgment call based on product philosophy. A code-based test can't evaluate this kind of nuance.

For subjective failures, we created another golden dataset. The domain expert, NurtureBoss's founder, Jacob, reviewed traces where a handoff was potentially needed, and made judgment calls. In each case, he provided a binary PASS/FAIL score and, crucially, a detailed **critique** explaining his reasoning.

Here are a few examples from their dataset:



Conversation Trace	Judgment	Critique
User: Hi, I'm outside the office for my 2pm tour. **AI**: I see you have a tour scheduled for 2pm. The office is located at 123 Main St. Our hours are 9am to 5pm. **User**: No, I'm literally standing at the door. It's locked. Can someone let me in?	**FAIL**	The AI failed to recognize the user's urgent, real-time context. Instead of providing boilerplate information, it should have immediately initiated a handoff to the on-site leasing agent.
User: Your website said there was a 2-bedroom available but I can't find it. **AI**: The 2-bedroom unit is model 2B-A. It is available on the 3rd floor. **User**: That's not what I'm asking. This is too confusing.	**FAIL**	The user explicitly stated their frustration. At this point, the AI should have de-escalated by offering to connect them to a human. Continuing to answer robotically created a poor user experience.
User: I'm trying to apply but the form won't accept my social security number. **AI**: I am sorry you are having trouble. Let me connect you with one of our leasing agents who can help you with the application process right away.	**PASS**	Perfect response. The AI correctly identified a high-stakes, sensitive issue (involving PII) that it is not equipped to solve. It immediately and appropriately escalated to a human expert.

Example dataset for an LLM-as-judge eval.

Using a PASS/FAIL judgment works better than a points rating. You'll notice that in the example above, domain expert Jacob used a simple PASS/FAIL judgment for each trace, not a 1-5 points rating. This was a deliberate choice. I've learned that while it's tempting to use a Likert scale to capture nuance; in practice, it creates more problems than it solves. The distinction between a "3" and a "4" is often subjective and inconsistent across different reviewers, leading to noisy, unreliable data.

In contrast, binary decisions force clarity and compel a domain expert to define a clear line between acceptable and unacceptable, focusing on what truly matters for users' success. It's also far more actionable for engineers: a "fail" is a clear signal to fix a bug, whereas a "3" is an ambiguous signal: a signal to do what, exactly? By starting with pass/fail, you cut through the ambiguity and get a clear, actionable measure of quality, faster.

This dataset of traces, judgments, and critiques does more than just help the team understand the problem. As we'll see in the next section, these hand-labeled examples, especially the detailed critiques, become raw material for building a reliable LLM-as-judge.

4. Building an LLM-as-judge

The hand-labeled dataset of handoff failures is a necessary first step in measuring and solving handoff issues. But another issue is that manual review doesn't scale. At NurtureBoss, the next step was to automate the domain expert's expertise by building an LLM-as-judge: an AI evaluator that could apply his reasoning consistently to thousands of future traces...

Subscribe to The Pragmatic Engineer to unlock the rest.

Become a paying subscriber of The Pragmatic Engineer to get access to this post and other subscriber-only content.

Upgrade to paid

A subscription gets you:

- ✓ Full articles every Tuesday and Thursday
- ✓ Access to resources and templates for engineering managers and engineers
- ✓ Access to the complete archive, see all comments and comment on articles

LIKE COMMENT RESTACK

© 2025 Gergely Orosz
548 Market Street PMB 72296, San Francisco, CA 94104
Unsubscribe

Start writing