Chapter 10. AI Engineering Architecture and User Feedback

So far, this book has covered a wide range of techniques to adapt foundation models to specific applications. This chapter will discuss how to bring these techniques together to build successful products.

Given the wide range of AI engineering techniques and tools available, selecting the right ones can feel overwhelming. To simplify this process, this chapter takes a gradual approach. It starts with the simplest architecture for a foundation model application, highlights the challenges of that architecture, and gradually adds components to address them.

We can spend eternity reasoning about how to build a successful application, but the only way to find out if an application actually achieves its goal is to put it in front of users. User feedback has always been invaluable for guiding product development, but for AI applications, user feedback has an even more crucial role as a data source for improving models. The conversational interface makes it easier for users to give feedback but harder for developers to extract signals. This chapter will discuss different types of conversational AI feedback and how to design a system to collect the right feedback without hurting user experience.

AI Engineering Architecture

A full-fledged AI architecture can be complex. This section follows the process that a team might follow in production, starting with the simplest architecture and progressively adding more components. Despite the diversity of AI applications, they share many common components. The architecture proposed here has been validated at multiple companies to be general for a wide range of applications, but certain applications might deviate.

In its simplest form, your application receives a query and sends it to the model. The model generates a response, which is returned to the user, as shown in Figure 10-1. There is no context augmentation, no guardrails, and no optimization. The *Model API* box refers to both third-party APIs (e.g., OpenAI, Google, Anthropic) and self-hosted models. Building an inference server for self-hosted models is discussed in Chapter 9.

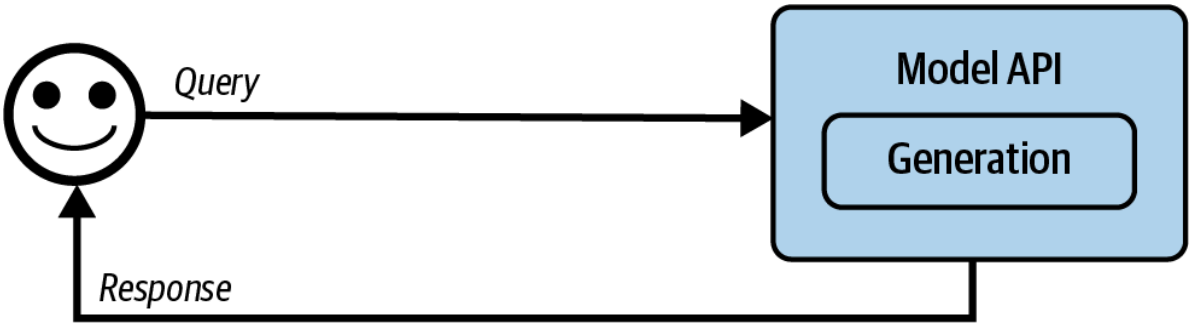

Figure 10-1. The simplest architecture for running an AI application.

From this simple architecture, you can add more components as needs arise. The process might look as follows:

1. Enhance context input into a model by giving the model access to external data sources and tools for information gathering.
2. Put in guardrails to protect your system and your users.
3. Add model router and gateway to support complex pipelines and add more security.
4. Optimize for latency and costs with caching.
5. Add complex logic and write actions to maximize your system's capabilities.

This chapter follows the progression I commonly see in production. However, everyone's needs are different. You should follow the order that makes the most sense for your application.

Monitoring and observability, which are integral to any application for quality control and performance improvement, will be discussed at the end of this process. Orchestration, chaining all these components together, will be discussed after that.

Step 1. Enhance Context

The initial expansion of a platform usually involves adding mechanisms to allow the system to construct the relevant context needed by the model to answer each query. As discussed in Chapter 6, context can be constructed through various retrieval mechanisms, including text retrieval, image retrieval, and tabular data retrieval. Context can also be augmented using tools that allow the model to automatically gather information through APIs such as web search, news, weather, events, etc.

Context construction is like feature engineering for foundation models. It gives the model the necessary information to produce an output. Due to its central role in a system's output quality, context construction is almost universally supported by model API providers. For example, providers like OpenAI, Claude, and Gemini allow users to upload files and allow their models to use tools.

However, just like models differ in their capabilities, these providers differ in their context construction support. For example, they might have limitations on what types of documents and how many you can upload. A specialized RAG solution might let you upload as many documents as your vector database can accommodate, but a generic model API might let you upload only a small number of documents. Different frameworks also differ in their retrieval algorithms and other retrieval configurations, like chunk sizes. Similarly, for tool use, solutions also differ in the types of tools they support and the modes of execution, such as whether they support parallel function execution or long-running jobs.

With context construction, the architecture now looks like Figure 10-2.

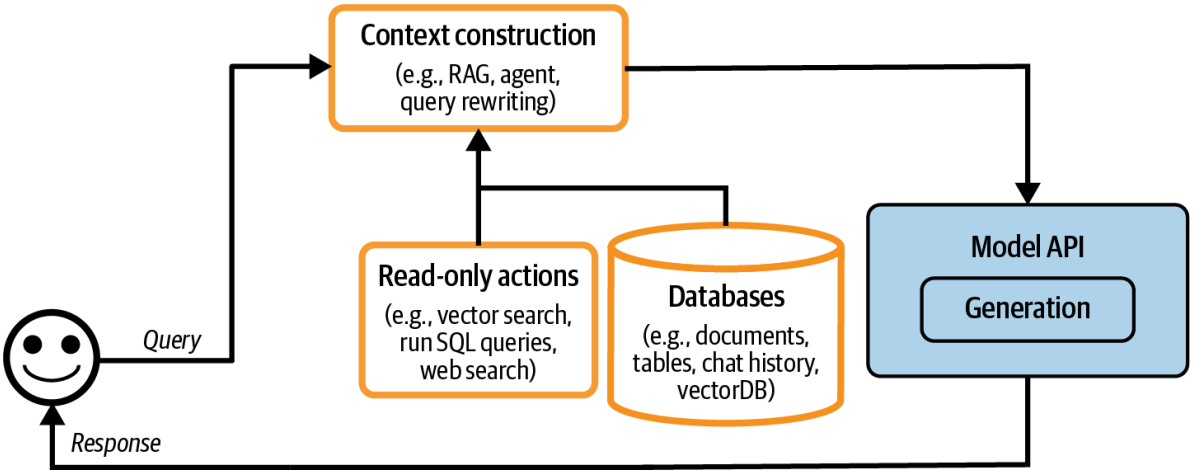

Figure 10-2. A platform architecture with context construction.

Step 2. Put in Guardrails

Guardrails help mitigate risks and protect you and your users. They should be placed whenever there are exposures to risks. In general, they can be categorized into guardrails around inputs and outputs.

Input guardrails

Input guardrails typically protect against two types of risks: leaking private information to external APIs and executing bad prompts that compromise your system. Chapter 5 discusses many different ways attackers can exploit an application through prompt hacks and how to defend your application against them. While you can mitigate risks, they can never be fully eliminated, due to the inherent nature of how models generate responses as well as unavoidable human failures.

Leaking private information to external APIs is a risk specific to using external model APIs when you need to send your data outside your organization. This might happen for many reasons, including the following:

- An employee copies the company's secret or a user's private information into a prompt and sends it to a third-party API.[1]
- An application developer puts the company's internal policies and data into the application's system prompt.
- A tool retrieves private information from an internal database and adds it to the context.

There's no airtight way to eliminate potential leaks when using third-party APIs. However, you can mitigate them with guardrails. You can use one of the many available tools that automatically detect sensitive data. What sensitive data to detect is specified by you. Common sensitive data classes are the following:

- Personal information (ID numbers, phone numbers, bank accounts)
- Human faces
- Specific keywords and phrases associated with the company's intellectual property or privileged information

Many sensitive data detection tools use AI to identify potentially sensitive information, such as determining if a string resembles a valid home address. If a query is found to contain sensitive information, you have two options: block the entire query or remove the sensitive information from it. For instance, you can mask a user's phone number with the placeholder [PHONE NUMBER]. If the generated response contains this placeholder, use a PII reverse dictionary that maps this placeholder to the original information so that you can unmask it, as shown in Figure 10-3.

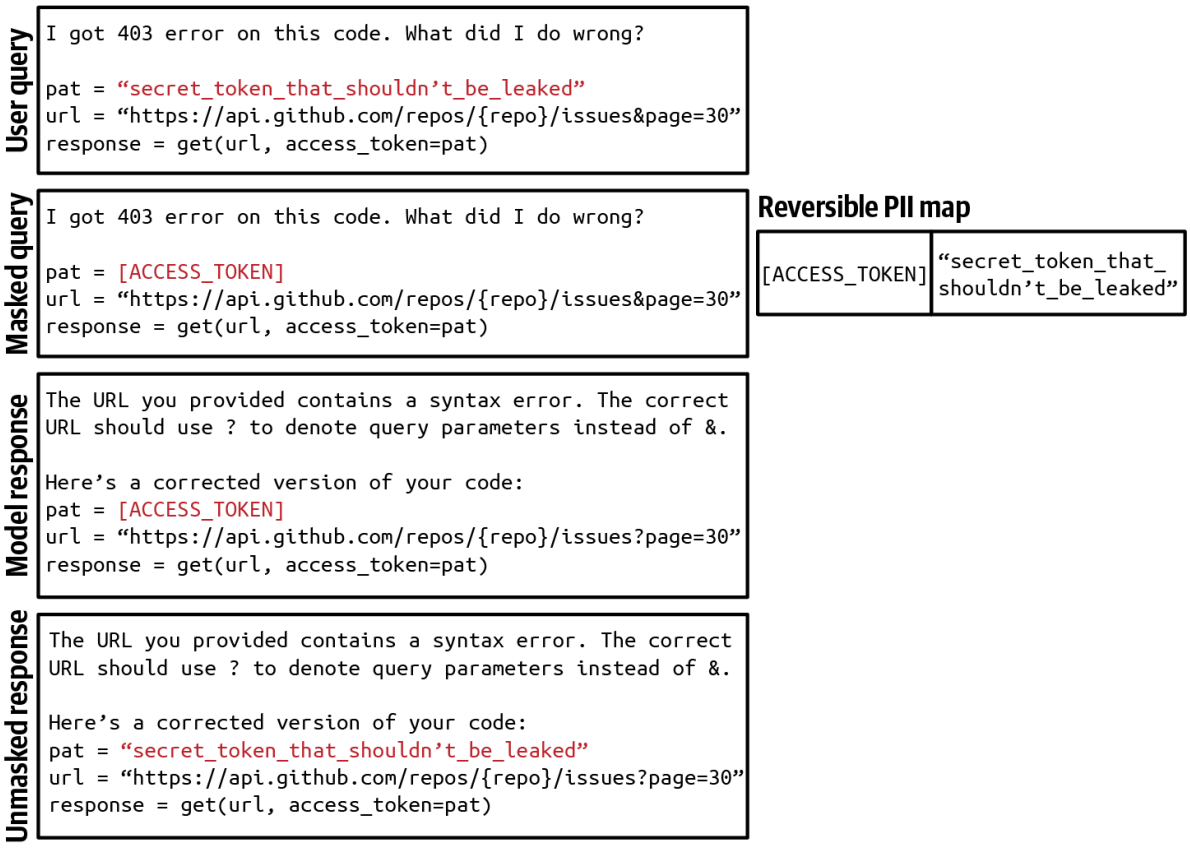

Figure 10-3. An example of masking and unmasking PII information using a reverse PII map to avoid sending it to external APIs.

Output guardrails

A model can fail in many different ways. Output guardrails have two main functions:

- Catch output failures
- Specify the policy to handle different failure modes

To catch outputs that fail to meet your standards, you need to understand what failures look like. The easiest failure to detect is when a model returns an empty response when it shouldn't.[2] Failures look different for different applications. Here are some common failures in the two main categories: quality and security. Quality failures are discussed in Chapter 4, and security failures are discussed in Chapter 5. I'll quickly mention a few of these failures as a recap:

- Quality
  - Malformatted responses that don't follow the expected output format. For example, the application expects JSON, and the model generates invalid JSON.
  - Factually inconsistent responses hallucinated by the model.
  - Generally bad responses. For example, you ask the model to write an essay, and that essay is just bad.
- Security
  - Toxic responses that contain racist content, sexual content, or illegal activities.
  - Responses that contain private and sensitive information.
  - Responses that trigger remote tool and code execution.
  - Brand-risk responses that mischaracterize your company or your competitors.

Recall from Chapter 5 that for security measurements, it's important to track not only the security failures but also the false refusal rate. It's possible to have systems that are too secure, e.g., one that blocks even legitimate requests, interrupting user workloads and causing user frustration.

Many failures can be mitigated by simple retry logic. AI models are probabilistic, which means that if you try a query again, you might get a different

response. For example, if the response is empty, try again X times or until you get a nonempty response. Similarly, if the response is malformatted, try again until the response is correctly formatted.

This retry policy, however, can incur extra latency and cost. Each retry means another round of API calls. If the retry is carried out after failure, the user-perceived latency will double. To reduce latency, you can make calls in parallel. For example, for each query, instead of waiting for the first query to fail before retrying, you send this query to the model twice at the same time, get back two responses, and pick the better one. This increases the number of redundant API calls while keeping latency manageable.

It's also common to fall back on humans for tricky requests. For example, you can transfer the queries that contain specific phrases to human operators. Some teams use a specialized model to decide when to transfer a conversation to humans. One team, for instance, transfers a conversation to human operators when their sentiment analysis model detects anger in users' messages. Another team transfers a conversation after a certain number of turns to prevent users from getting stuck in a loop.

Guardrail implementation

Guardrails come with trade-offs. One is the *reliability versus latency trade-off*. While acknowledging the importance of guardrails, some teams told me that latency is more important. The teams decided not to implement guardrails because they can significantly increase the application's latency.[3]

Output guardrails might not work well in the stream completion mode. By default, the whole response is generated before being shown to the user, which can take a long time. In the stream completion mode, new tokens are streamed to the user as they are generated, reducing the time the user has to wait to see the response. The downside is that it's hard to evaluate partial responses, so unsafe responses might be streamed to users before the system guardrails can determine that they should be blocked.

How many guardrails you need to implement also depends on whether you self-host your models or use third-party APIs. While you can implement

guardrails on top of both, third-party APIs can reduce the guardrails you need to implement since API providers typically provide many guardrails out of the box for you. At the same time, self-hosting means that you don't need to send requests externally, which reduces the need for many types of input guardrails.

Given the many different places where an application might fail, guardrails can be implemented at many different levels. Model providers give their models guardrails to make their models better and more secure. However, model providers have to balance safety and flexibility. Restrictions might make a model safer but can also make it less usable for specific use cases.

Guardrails can also implemented by application developers. Many techniques are discussed in "Defenses Against Prompt Attacks". Guardrail solutions that you can use out of the box include Meta's Purple Llama, NVIDIA's NeMo Guardrails, Azure's PyRIT, Azure's AI content filters, the Perspective API, and OpenAI's content moderation API. Due to the overlap of risks in inputs and outputs, a guardrail solution will likely provide protection for both inputs and outputs. Some model gateways also provide guardrail functionalities, as discussed in the next section.

With guardrails, the architecture looks like Figure 10-4. I put scorers under model APIs since scorers are often AI-powered, even if scorers are typically smaller and faster than generative models. However, scorers can also be placed in the output guardrails box.

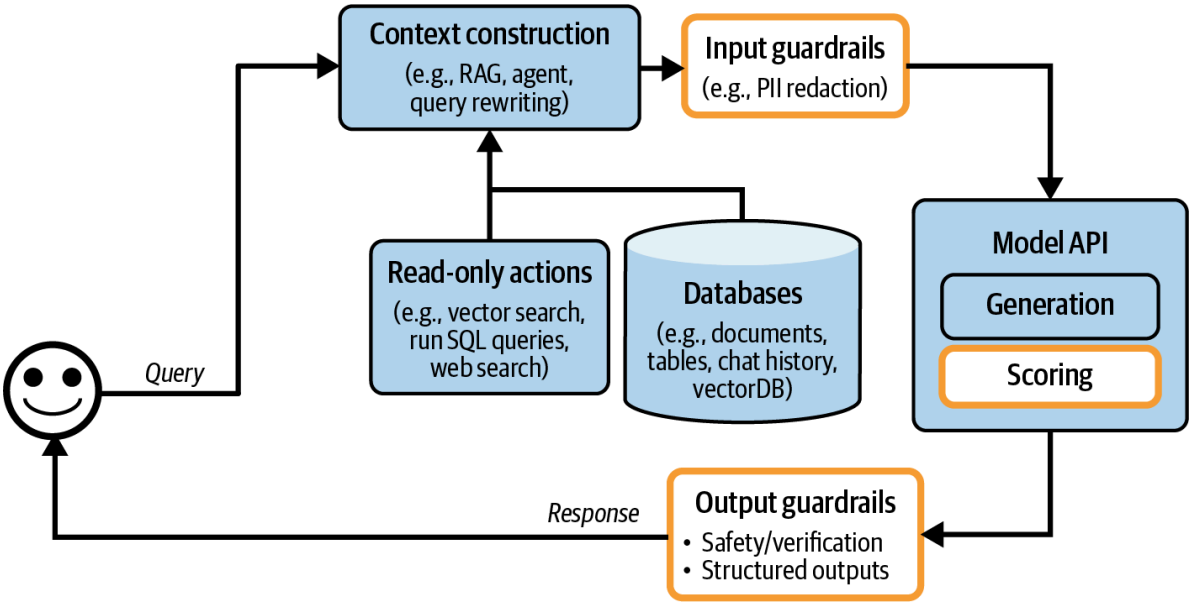

Figure 10-4. Application architecture with the addition of input and output guardrails.

Step 3. Add Model Router and Gateway

As applications grow to involve more models, routers and gateways emerge to help you manage the complexity and costs of serving multiple models.

Router

Instead of using one model for all queries, you can have different solutions for different types of queries. This approach has several benefits. First, it allows specialized models, which can potentially perform better than a general-purpose model for specific queries. For example, you can have one model specialized in technical troubleshooting and another specialized in billing. Second, this can help you save costs. Instead of using one expensive model for all queries, you can route simpler queries to cheaper models.

A router typically consists of *an intent classifier* that predicts what the user is trying to do. Based on the predicted intent, the query is routed to the appropriate solution. As an example, consider different intentions relevant to a customer support chatbot:

- If the user wants to reset the password, route them to the FAQ page about recovering the password.
- If the request is to correct a billing mistake, route it to a human operator.
- If the request is about troubleshooting a technical issue, route it to a chatbot specialized in troubleshooting.

An intent classifier can prevent your system from engaging in out-of-scope conversations. If the query is deemed inappropriate, the chatbot can politely decline to respond using one of the stock responses without wasting an API call. For example, if the user asks who you would vote for in the upcoming election, a chatbot can respond with: "As a chatbot, I don't have the ability to vote. If you have questions about our products, I'd be happy to help."

An intent classifier can help the system detect ambiguous queries and ask for clarification. For example, in response to the query "Freezing", the system might ask, "Do you want to freeze your account or are you talking about the weather?" or simply ask, "I'm sorry. Can you elaborate?"

Other routers can aid the model in deciding what to do next. For example, for an agent capable of multiple actions, a router can take the form of a *next-action predictor*: should the model use a code interpreter or a search API next? For a model with a memory system, a router can predict which part of the memory hierarchy the model should pull information from. Imagine that a user attaches a document that mentions Melbourne to the current conversation. Later on, the user asks: "What's the cutest animal in Melbourne?" The model needs to decide whether to rely on the information in the attached document or to search the internet for this query.

Intent classifiers and next-action predictors can be implemented on top of foundation models. Many teams adapt smaller language models like GPT-2, BERT, and Llama 7B as their intent classifiers. Many teams opt to train even smaller classifiers from scratch. Routers should be fast and cheap so that they can use multiples of them without incurring significant extra latency and cost.

When routing queries to models with varying context limits, the query's context might need to be adjusted accordingly. Consider a 1,000-token query that is slated for a model with a 4K context limit. The system then takes an action, e.g., a web search, that brings back 8,000-token context. You can either truncate the query's context to fit the originally intended model or route the query to a model with a larger context limit.

Because routing is usually done by models, I put routing inside the Model API box in Figure 10-5. Like scorers, routers are typically smaller than models

used for generation.

Grouping routers together with other models makes models easier to manage. However, it's important to note that routing often happens *before* retrieval. For example, before retrieval, a router can help determine if a query is in-scope and, if yes, if it needs retrieval. Routing can happen after retrieval, too, such as determining if a query should be routed to a human operator. However, routing - retrieval - generation - scoring is a much more common AI application pattern.

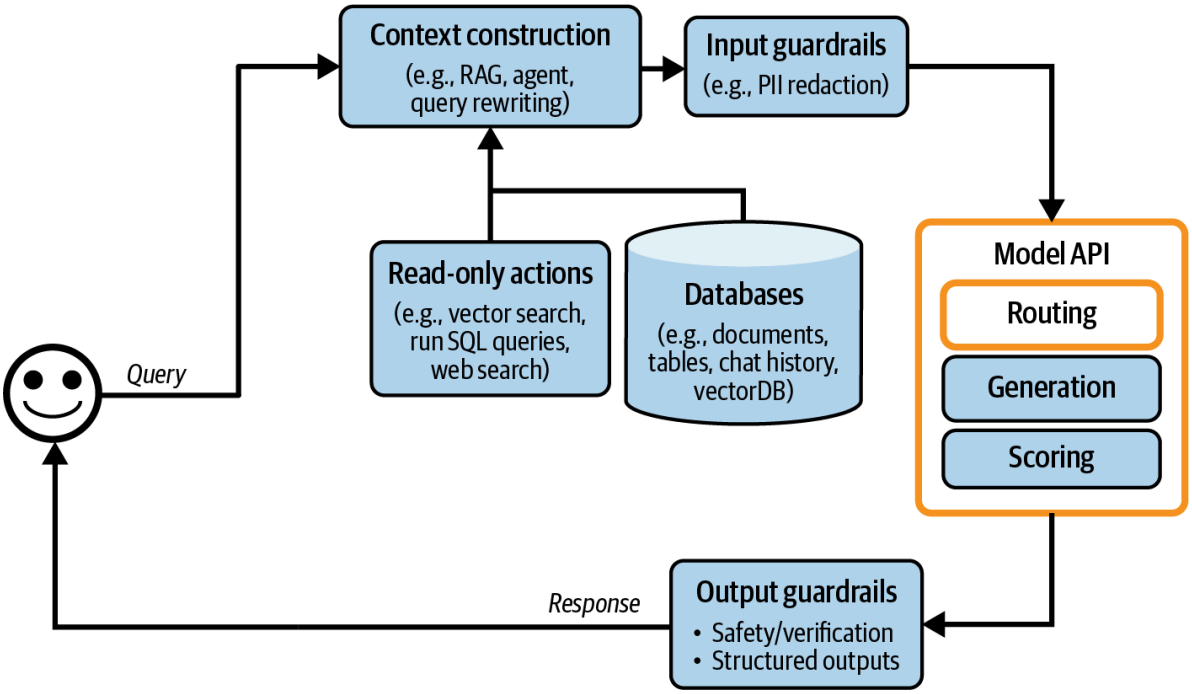

Figure 10-5. Routing helps the system use the optimal solution for each query.

Gateway

A model gateway is an intermediate layer that allows your organization to interface with different models in a unified and secure manner. The most basic functionality of a model gateway is to provide a unified interface to different models, including self-hosted models and models behind commercial APIs. A model gateway makes it easier to maintain your code. If a model API changes, you only need to update the gateway instead of updating all applications that depend on this API. Figure 10-6 shows a high-level visualization of a model gateway.

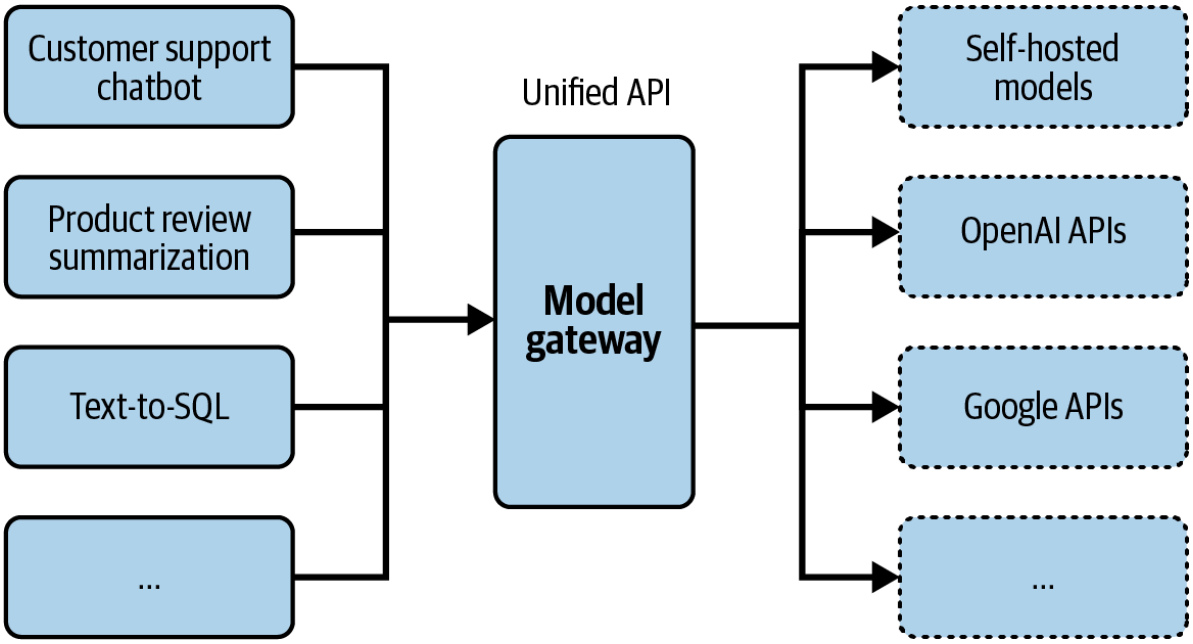

Figure 10-6. A model gateway provides a unified interface to work with different models.

In its simplest form, a model gateway is a unified wrapper. The following code example gives you an idea of how a model gateway might be implemented. It's not meant to be functional, as it doesn't contain any error checking or optimization:

```
import google.generativeai as genai
import openai

def openai_model(input_data, model_name, max_tokens):
    openai.api_key = os.environ["OPENAI_API_KEY"]
    response = openai.Completion.create(
        engine=model_name,
        prompt=input_data,
        max_tokens=max_tokens
    )
    return {"response": response.choices[0].text.strip()}

def gemini_model(input_data, model_name, max_tokens):
    genai.configure(api_key=os.environ["GOOGLE_API_KEY"])
    model = genai.GenerativeModel(model_name=model_name)
    response = model.generate_content(input_data, max_tokens=max_tokens)
    return {"response": response["choices"][0]["message"]["content"]}

@app.route('/model', methods=['POST'])
def model_gateway():
```

```
    data = request.get_json()
    model_type = data.get("model_type")
        model_name = data.get("model_name")
        input_data = data.get("input_data")
        max_tokens = data.get("max_tokens")

        if model_type == "openai":
            result = openai_model(input_data, model_name, max_tokens)
        elif model_type == "gemini":
            result = gemini_model(input_data, model_name, max_tokens)
        return jsonify(result)
```

A model gateway provides *access control and cost management*. Instead of giving everyone who wants access to the OpenAI API your organizational tokens, which can be easily leaked, you give people access only to the model gateway, creating a centralized and controlled point of access. The gateway can also implement fine-grained access controls, specifying which user or application should have access to which model. Moreover, the gateway can monitor and limit the usage of API calls, preventing abuse and managing costs effectively.

A model gateway can also be used to implement fallback policies to overcome rate limits or API failures (the latter is unfortunately common). When the primary API is unavailable, the gateway can route requests to alternative models, retry after a short wait, or handle failures gracefully in other ways. This ensures that your application can operate smoothly without interruptions.

Since requests and responses are already flowing through the gateway, it's a good place to implement other functionalities, such as load balancing, logging, and analytics. Some gateways even provide caching and guardrails.

Given that gateways are relatively straightforward to implement, there are many off-the-shelf gateways. Examples include Portkey's AI Gateway, MLflow AI Gateway, Wealthsimple's LLM Gateway, TrueFoundry, Kong, and Cloudflare.

In our architecture, the gateway now replaces the model API box, as shown in Figure 10-7.

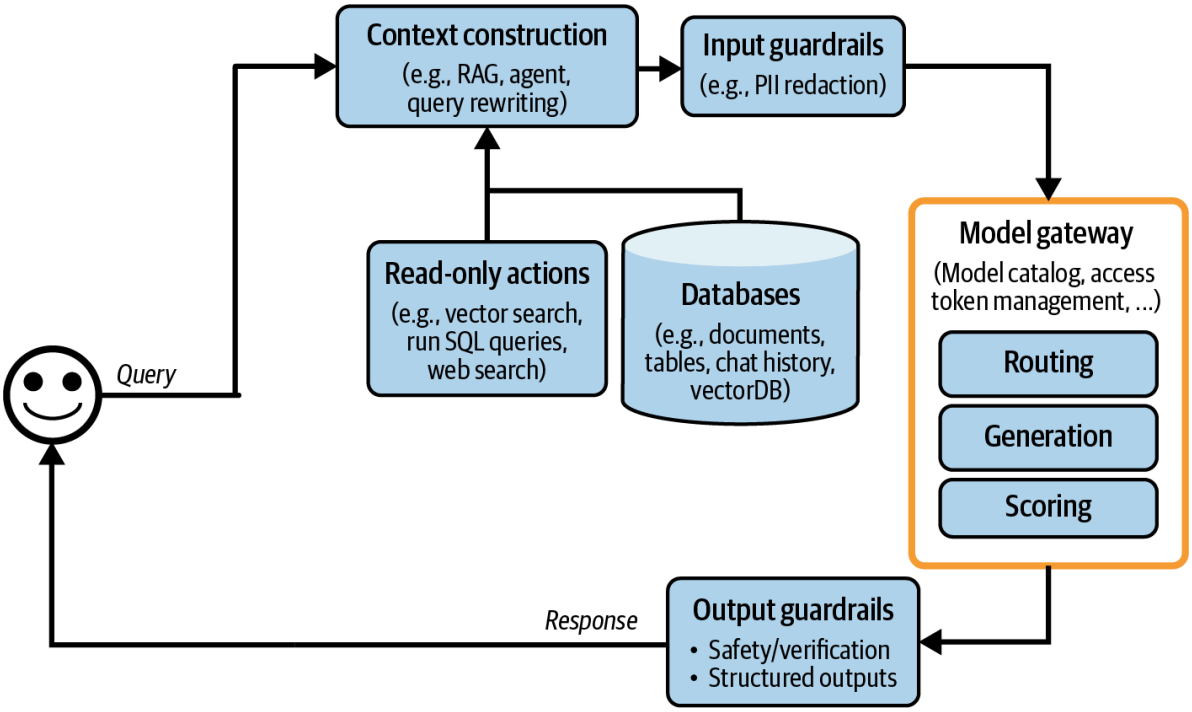

Figure 10-7. The architecture with the added routing and gateway modules.

NOTE

A similar abstraction layer, such as a tool gateway, can also be useful for accessing a wide range of tools. It's not discussed in this book since it's not a common pattern as of this writing.

Step 4. Reduce Latency with Caches

Caching has long been integral to software applications to reduce latency and cost. Many ideas from software caching can be used for AI applications. Inference caching techniques, including KV caching and prompt caching, are discussed in Chapter 9. This section focuses on system caching. Because caching is an old technology with a large amount of existing literature, this book will cover it only in broad strokes. In general, there are two major system caching mechanisms: exact caching and semantic caching.

Exact caching

With exact caching, cached items are used only when these exact items are requested. For example, if a user asks a model to summarize a product, the system checks the cache to see if a summary of this exact product exists. If

yes, fetch this summary. If not, summarize the product and cache the summary.

Exact caching is also used for embedding-based retrieval to avoid redundant vector search. If an incoming query is already in the vector search cache, fetch the cached result. If not, perform a vector search for this query and cache the result.

Caching is especially appealing for queries that involve multiple steps (e.g., chain-of-thought) and/or time-consuming actions (e.g., retrieval, SQL execution, or web search).

An exact cache can be implemented using in-memory storage for fast retrieval. However, since in-memory storage is limited, a cache can also be implemented using databases like PostgreSQL, Redis, or tiered storage to balance speed and storage capacity. Having an eviction policy is crucial to manage the cache size and maintain performance. Common eviction policies include Least Recently Used (LRU), Least Frequently Used (LFU), and first in, first out (FIFO).

How long to keep a query in the cache depends on how likely this query is to be called again. User-specific queries, such as "What's the status of my recent order?", are less likely to be reused by other users and, therefore, shouldn't be cached. Similarly, it makes less sense to cache time-sensitive queries such as "How's the weather?" Many teams train a classifier to predict whether a query should be cached.

WARNING

Caching, when not properly handled, can cause data leaks. Imagine you work for an ecommerce site, and user X asks a seemingly generic question such as: "What is the return policy for electronics products?" Because your return policy depends on the user's membership, the system first retrieves user X's information and then generates a response containing X's information. Mistaking this query for a generic question, the system caches the answer. Later, when user Y asks the same question, the cached result is returned, revealing X's information to Y.

Semantic caching

Unlike in exact caching, cached items are used even if they are only semantically similar, not identical, to the incoming query. Imagine one user asks, "What's the capital of Vietnam?" and the model answers, "Hanoi". Later, another user asks, "What's the capital *city* of Vietnam?", which is semantically the same question but with slightly different wording. With semantic caching, the system can reuse the answer from the first query instead of computing the new query from scratch. Reusing similar queries increases the cache's hit rate and potentially reduces cost. However, semantic caching can reduce your model's performance.

Semantic caching works only if you have a reliable way of determining if two queries are similar. One common approach is to use semantic similarity, as discussed in Chapter 3. As a refresh, semantic similarity works as follows:

1. For each query, generate its embedding using an embedding model.
2. Use vector search to find the cached embedding with the highest similar score to the current query embedding. Let's say this similarity score is *X*.
3. If *X* is higher than a certain similarity threshold, the cached query is considered similar, and the cached results are returned. If not, process this current query and cache it together with its embedding and results.

This approach requires a vector database to store the embeddings of cached queries.

Compared to other caching techniques, semantic caching's value is more dubious because many of its components are prone to failure. Its success relies on high-quality embeddings, functional vector search, and a reliable similarity metric. Setting the right similarity threshold can also be tricky, requiring a lot of trial and error. If the system mistakes the incoming query for one similar to another query, the returned response, fetched from the cache, will be incorrect.

In addition, semantic cache can be time-consuming and compute-intensive, as it involves a vector search. The speed and cost of this vector search depend on the size of your cached embeddings.

Semantic cache might still be worthwhile if the cache hit rate is high, meaning that a good portion of queries can be effectively answered by leveraging the cached results. However, before incorporating the complexities of a semantic cache, make sure to evaluate the associated efficiency, cost, and performance risks.

With the added cache systems, the platform looks like Figure 10-8. A KV cache and prompt cache are typically implemented by model API providers, so they aren't shown in this image. To visualize them, I'd put them in the Model API box. There's a new arrow to add generated responses to the cache.

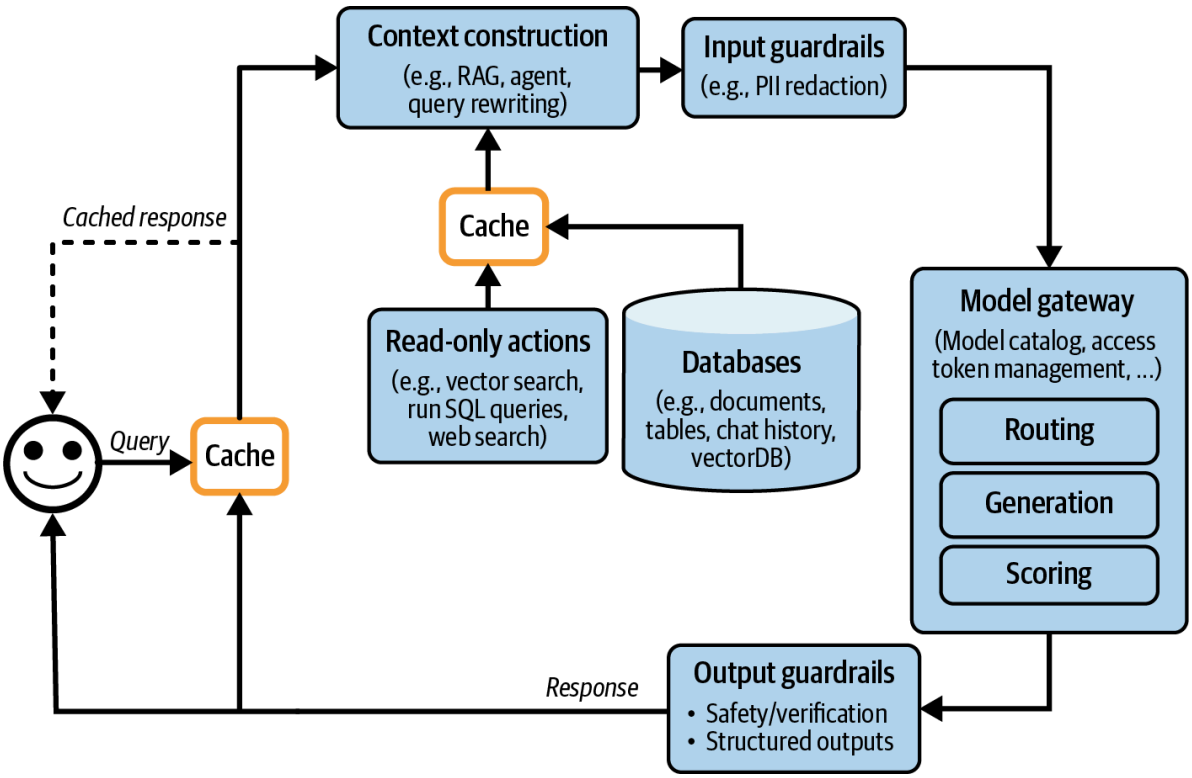

Figure 10-8. An AI application architecture with the added caches.

Step 5. Add Agent Patterns

The applications discussed so far are still fairly simple. Each query follows a sequential flow. However, as discussed in Chapter 6, an application flow can be more complex with loops, parallel execution, and conditional branching. Agentic patterns, discussed in Chapter 6, can help you build complex applications. For example, after the system generates an output, it might determine that it hasn't accomplished the task and that it needs to perform another retrieval to gather more information. The original response, together with

the newly retrieved context, is passed into the same model or a different one. This creates a loop, as shown in Figure 10-9.

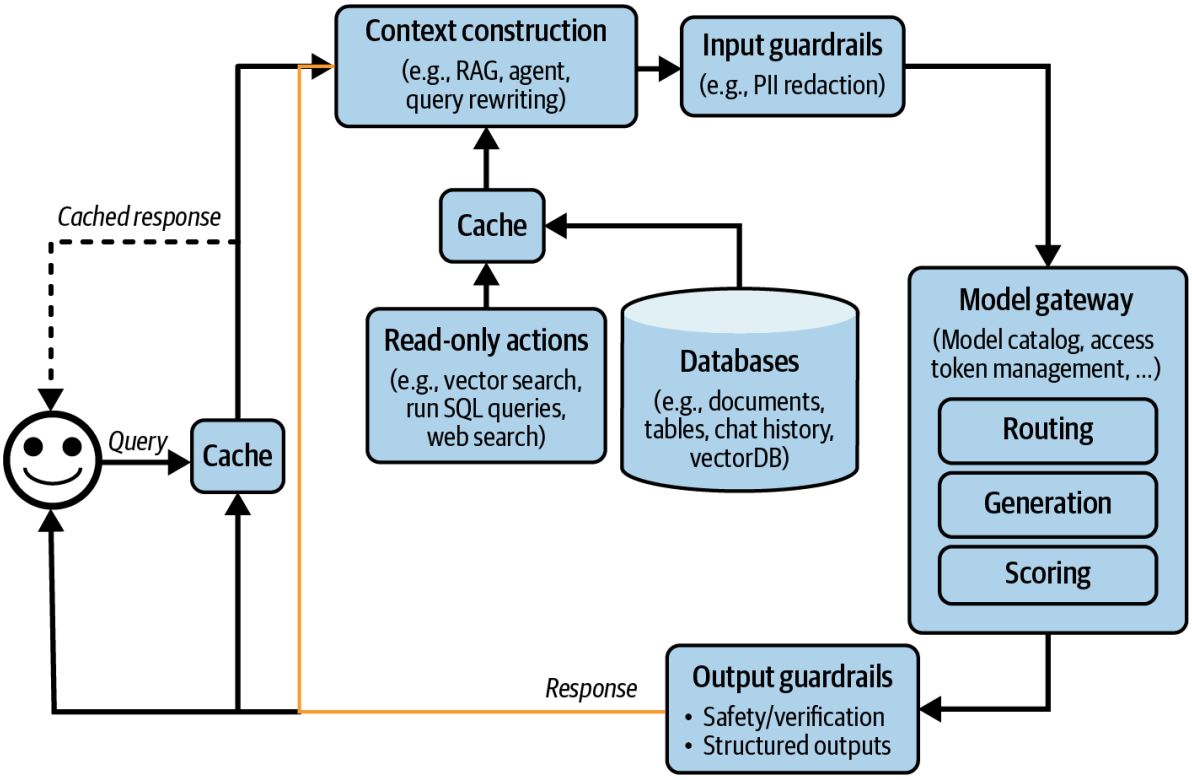

Figure 10-9. The yellow arrow allows the generated response to be fed back into the system, allowing more complex application patterns.

A model's outputs also can be used to invoke write actions, such as composing an email, placing an order, or initializing a bank transfer. Write actions allow a system to make changes to its environment directly. As discussed in Chapter 6, write actions can make a system vastly more capable but also expose it to significantly more risks. Giving a model access to write actions should be done with the utmost care. With added write actions, the architecture looks like Figure 10-10.

If you've followed all the steps so far, your architecture has likely grown quite complex. While complex systems can solve more tasks, they also introduce more failure modes, making them harder to debug due to the many potential points of failure. The next section will cover best practices for improving system observability.

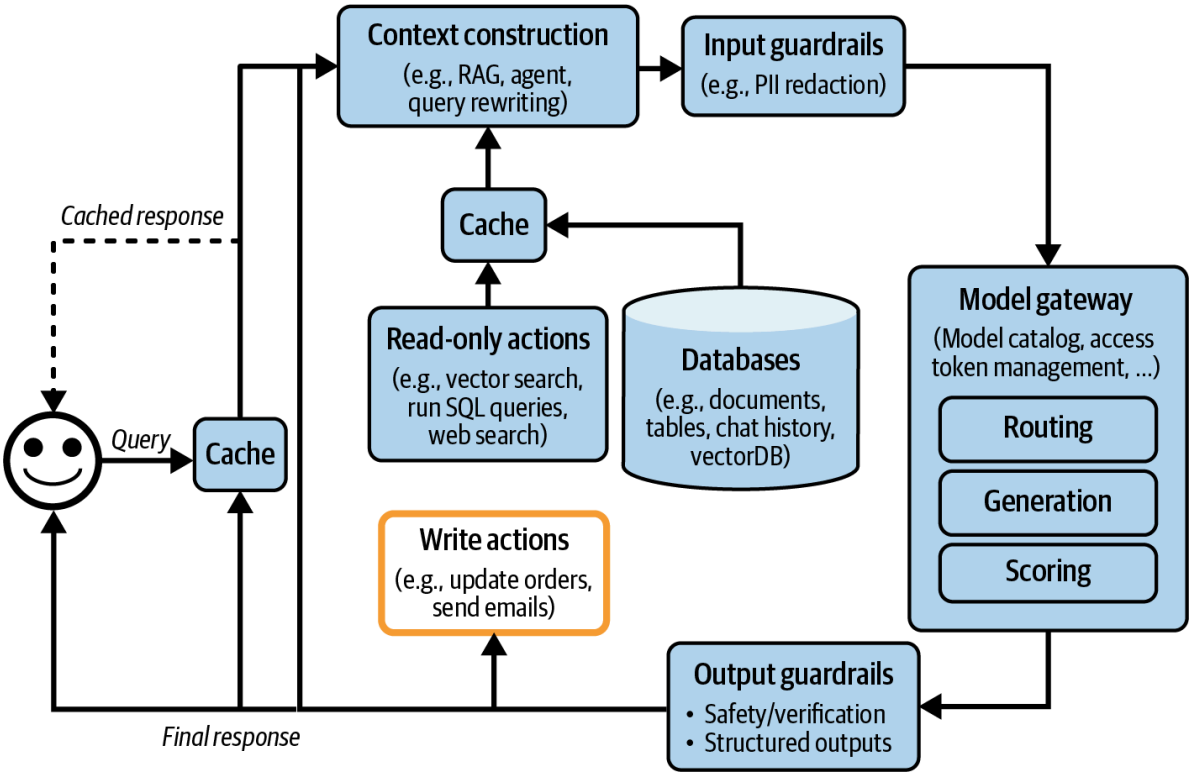

Figure 10-10. An application architecture that enables the system to perform write actions.

Monitoring and Observability

Even though I put observability in its own section, observability should be integral to the design of a product, rather than an afterthought. The more complex a product, the more crucial observability is.

Observability is a universal practice across all software engineering disciplines. It's a big industry with established best practices and many ready-to-use proprietary and open source solutions.[4] To avoid reinventing the wheel, I'll focus on what's unique to applications built on top of foundation models. The book's GitHub repository contains resources for those who want to learn more about observability.[5]

The goal of monitoring is the same as the goal of evaluation: to mitigate risks and discover opportunities. Risks that monitoring should help you mitigate include application failures, security attacks, and drifts. Monitoring can help discover opportunities for application improvement and cost savings. Monitoring can also help keep you accountable by giving visibility into your system's performance.

Three metrics can help evaluate the quality of your system's observability, derived from the DevOps community:

- MTTD (mean time to detection): When something bad happens, how long does it take to detect it?
- MTTR (mean time to response): After detection, how long does it take to be resolved?
- CFR (change failure rate): The percentage of changes or deployments that result in failures requiring fixes or rollbacks. If you don't know your CFR, it's time to redesign your platform to make it more observable.

Having a high CFR doesn't necessarily indicate a bad monitoring system. However, you should rethink your evaluation pipeline so that bad changes are caught before being deployed. Evaluation and monitoring need to work closely together. Evaluation metrics should translate well to monitoring metrics, meaning that a model that does well during evaluation should also do well during monitoring. Issues detected during monitoring should be fed to the evaluation pipeline.

MONITORING VERSUS OBSERVABILITY

Since the mid-2010s, the industry has embraced the term "observability" instead of "monitoring." Monitoring makes no assumption about the relationship between the internal state of a system and its outputs. You monitor the external outputs of the system to figure out when something goes wrong inside the system—there's no guarantee that the external outputs will help you figure out what goes wrong.

Observability, on the other hand, makes an assumption stronger than traditional monitoring: that a system's internal states can be inferred from knowledge of its external outputs. When something goes wrong with an observable system, we should be able to figure out what went wrong by looking at the system's logs and metrics without having to ship new code to the system. Observability is about instrumenting your system in a way that ensures that sufficient information about a system's runtime is collected and analyzed so that when something goes wrong, it can help you figure out what goes wrong.

In this book, I'll use the term "monitoring" to refer to the act of tracking a system's information and "observability" to refer to the whole process of instrumentating, tracking, and debugging the system.

Metrics

When discussing monitoring, most people think of metrics. However, metrics themselves aren't the goal. Frankly, most companies don't care what your application's output relevancy score is unless it serves a purpose. The purpose of a metric is to tell you when something is wrong and to identify opportunities for improvement.

Before listing what metrics to track, it's important to understand what failure modes you want to catch and design your metrics around these failures. For example, if you don't want your application to hallucinate, design metrics that help you detect hallucinations. One relevant metric might be whether an application's output can be inferred from the context. If you don't want your application to burn through your API credit, track metrics related to API

costs, such as the number of input and output tokens per request or your cache's cost and your cache's hit rate.

Because foundation models can generate open-ended outputs, there are many ways things can go wrong. Metrics design requires analytical thinking, statistical knowledge, and, often, creativity. Which metrics you should track are highly application-specific.

This book has covered many different types of model quality metrics (Chapters 4–6, and later in this chapter) and many different ways to compute them (Chapters 3 and 5). Here, I'll do a quick recap.

The easiest types of failures to track are format failures because they are easy to notice and verify. For example, if you expect JSON outputs, track how often the model outputs invalid JSON and, among these invalid JSON outputs, how many can be easily fixed (missing a closing bracket is easy to fix, but missing expected keys is harder).

For open-ended generations, consider monitoring factual consistency and relevant generation quality metrics such as conciseness, creativity, or positivity. Many of these metrics can be computed using AI judges.

If safety is an issue, you can track toxicity-related metrics and detect private and sensitive information in both inputs and outputs. Track how often your guardrails get triggered and how often your system refuses to answer. Detect abnormal queries to your system, too, since they might reveal interesting edge cases or prompt attacks.

Model quality can also be inferred through user natural language feedback and conversational signals. For example, some easy metrics you can track include the following:

- How often do users stop a generation halfway?
- What's the average number of turns per conversation?
- What's the average number of tokens per input? Are users using your application for more complex tasks, or are they learning to be more concise with their prompts?

- What's the average number of tokens per output? Are some models more verbose than others? Are certain types of queries more likely to result in lengthy answers?
- What's the model's output token distribution? How has it changed over time? Is the model getting more or less diverse?

Length-related metrics are also important for tracking latency and costs, as longer contexts and responses typically increase latency and incur higher costs.

Each component in an application pipeline has its own metrics. For example, in a RAG application, the retrieval quality is often evaluated using context relevance and context precision. A vector database can be evaluated by how much storage it needs to index the data and how long it takes to query the data.

Given that you'll likely have multiple metrics, it's useful to measure how these metrics correlate to each other and, especially, to your business north star metrics, which can be DAU (daily active user), session duration (the length of time a user spends actively engaged with the application), or subscriptions. Metrics that are strongly correlated to your north star might give you ideas on how to improve your north star. Metrics that are not at all correlated might also give you ideas on what not to optimize for.

Tracking latency is essential for understanding the user experience. Common latency metrics, as discussed in Chapter 9, include:

- Time to first token (TTFT): the time it takes for the first token to be generated.
- Time per output token (TPOT): the time it takes to generate each output token.
- Total latency: the total time required to complete a response.

Track all these metrics per user to see how your system scales with more users.

You'll also want to track costs. Cost-related metrics are the number of queries and the volume of input and output tokens, such as tokens per second (TPS).

If you use an API with rate limits, tracking the number of requests per second is important to ensure you stay within your allocated limits and avoid potential service interruptions.

When calculating metrics, you can choose between spot checks and exhaustive checks. Spot checks involve sampling a subset of data to quickly identify issues, while exhaustive checks evaluate every request for a comprehensive performance view. The choice depends on your system's requirements and available resources, with a combination of both providing a balanced monitoring strategy.

When computing metrics, ensure they can be broken down by relevant axes, such as users, releases, prompt/chain versions, prompt/chain types, and time. This granularity helps in understanding performance variations and identifying specific issues.

Logs and traces

Metrics are typically aggregated. They condense information from events that occur in your system over time. They help you understand, at a glance, how your system is doing. However, there are many questions that metrics can't help you answer. For example, after seeing a spike in a specific activity, you might wonder: "Has this happened before?" Logs can help you answer this question.

If metrics are numerical measurements representing attributes and events, logs are an append-only record of events. In production, a debugging process might look like this:

1. Metrics tell you something went wrong five minutes ago, but they don't tell you what happened.
2. You look at the logs of events that took place around five minutes ago to figure out what happened.
3. Correlate the errors in the logs to the metrics to make sure that you've identified the right issue.

For fast detection, metrics need to be computed quickly. For fast response, logs need to be readily available and accessible. If your logs are 15 minutes

delayed, you will have to wait for the logs to arrive to track down an issue that happened 5 minutes ago.

Because you don't know exactly what logs you'll need to look at in the future, the general rule for logging is to log everything. Log all the configurations, including the model API endpoint, model name, sampling settings (temperature, top-p, top-k, stopping condition, etc.), and the prompt template.

Log the user query, the final prompt sent to the model, the output, and the intermediate outputs. Log if it calls any tool. Log the tool outputs. Log when a component starts, ends, when something crashes, etc. When recording a piece of log, make sure to give it tags and IDs that can help you know where this log comes from in the system.

Logging everything means that the amount of logs you have can grow very quickly. Many tools for automated log analysis and log anomaly detection are powered by AI.

While it's impossible to process logs manually, it's useful to manually inspect your production data daily to get a sense of how users are using your application. Shankar et al., (2024) found that the developers' perceptions of what constitutes good and bad outputs change as they interact with more data, allowing them to both rewrite their prompts to increase the chance of good responses and update their evaluation pipeline to catch bad responses.

If logs are a series of disjointed events, traces are reconstructed by linking related events together to form a complete timeline of a transaction or process, showing how each step connects from start to finish. In short, a trace is the detailed recording of a request's execution path through various system components and services. In an AI application, tracing reveals the entire process from when a user sends a query to when the final response is returned, including the actions the system takes, the documents retrieved, and the final prompt sent to the model. It should also show how much time each step takes and its associated cost, if measurable. Figure 10-11 is a visualization of a request's trace in LangSmith.

Ideally, you should be able to trace each query's transformation step-by-step through the system. If a query fails, you should be able to pinpoint the exact

step where it went wrong: whether it was incorrectly processed, the retrieved context was irrelevant, or the model generated a wrong response.

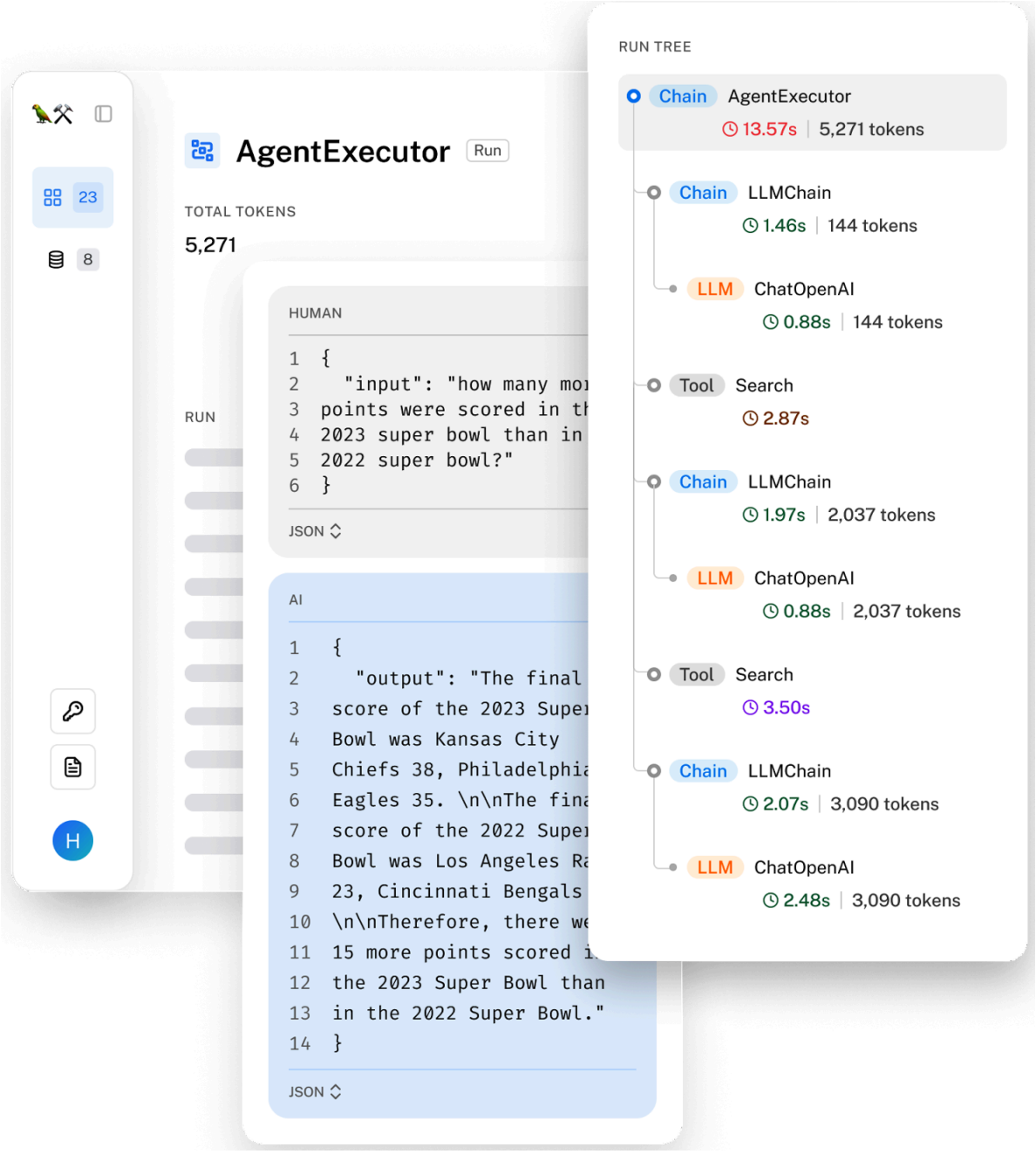

Figure 10-11. A request trace visualized by LangSmith.

Drift detection

The more parts a system has, the more things that can change. In an AI application these can be:

System prompt changes

There are many reasons why your application's system prompt might change without your knowing. The system prompt could've been built

on top of a prompt template, and that prompt template was updated. A coworker could've found a typo and fixed it. A simple logic should be sufficient to catch when your application's system prompt changes.

User behavior changes

Over time, users adapt their behaviors to the technology. For example, people have already figured out how to frame their queries to get better results on Google Search or how to make their articles rank higher on search results. People living in areas with self-driving cars have already figured out how to bully self-driving cars into giving them the right of way (Liu et al., 2020). It's likely that your users will change their behaviors to get better results out of your application. For example, your users might learn to write instructions to make the responses more concise. This might cause a gradual drop in response length over time. If you look only at metrics, it might not be obvious what caused this gradual drop. You need investigations to understand the root cause.

Underlying model changes

When using a model through an API, it's possible that the API remains unchanged while the underlying model is updated. As mentioned in Chapter 4, model providers might not always disclose these updates, leaving it to you to detect any changes. Different versions of the same API can have a significant impact on performance. For instance, Chen et al. (2023) observed notable differences in benchmark scores between the March 2023 and June 2023 versions of GPT-4 and GPT-3.5. Likewise, Voiceflow reported a 10% performance drop when switching from the older GPT-3.5-turbo-0301 to the newer GPT-3.5-turbo-1106.

AI Pipeline Orchestration

An AI application can get fairly complex, consisting of multiple models, retrieving data from many databases, and having access to a wide range of tools. An orchestrator helps you specify how these different components work together to create an end-to-end pipeline. It ensures that data flows seamlessly between components. At a high level, an orchestrator operates in two steps, components definition and chaining:

Components definition

You need to tell the orchestrator what components your system uses, including different models, external data sources for retrieval, and tools that your system can use. A model gateway can make it easier to add a model.[6] You can also tell the orchestrator if you use any tools for evaluation and monitoring.

Chaining

Chaining is basically function composition: it combines different functions (components) together. In chaining (pipelining), you tell the orchestrator the steps your system takes from receiving the user query until completing the task. Here's an example of the steps:

1. Process the raw query.
2. Retrieve the relevant data based on the processed query.
3. Combine the original query and the retrieved data to create a prompt in the format expected by the model.
4. The model generates a response based on the prompt.
5. Evaluate the response.
6. If the response is considered good, return it to the user. If not, route the query to a human operator.

The orchestrator is responsible for passing data between components. It should provide toolings that help ensure that the output from the current step is in the format expected by the next step. Ideally, it should notify you when this data flow is disrupted due to errors such as component failures or data mismatch failures.

WARNING

An AI pipeline orchestrator is different from a general workflow orchestrator, like Airflow or Metaflow.

When designing the pipeline for an application with strict latency requirements, try to do as much in parallel as possible. For example, if you have a routing component (deciding where to send a query) and a PII removal component, both can be done at the same time.

There are many AI orchestration tools, including LangChain, LlamaIndex, Flowise, Langflow, and Haystack. Because retrieval and tool use are common application patterns, many RAG and agent frameworks are also orchestration tools.

While it's tempting to jump straight to an orchestration tool when starting a project, *you might want to start building your application without one first.* Any external tool brings additional complexity. An orchestrator can abstract away critical details of how your system works, making it hard to understand and debug your system.

As you advance to the later stages of your application development process, you might decide that an orchestrator can make your life easier. Here are three aspects to keep in mind when evaluating orchestrators:

Integration and extensibility

Evaluate whether the orchestrator supports the components you're already using or might adopt in the future. For example, if you want to use a Llama model, check if the orchestrator supports that. Given how many models, databases, and frameworks there are, it's impossible for an orchestrator to support everything. Therefore, you'll also need to consider an orchestrator's extensibility. If it doesn't support a specific component, how hard is it to change that?

Support for complex pipelines

As your applications grow in complexity, you might need to manage intricate pipelines involving multiple steps and conditional logic. An orchestrator that supports advanced features like branching, parallel processing, and error handling will help you manage these complexities efficiently.

Ease of use, performance, and scalability

Consider the user-friendliness of the orchestrator. Look for intuitive APIs, comprehensive documentation, and strong community support, as these can significantly reduce the learning curve for you and your team. Avoid orchestrators that initiate hidden API calls or introduce latency to your applications. Additionally, ensure that the orchestrator can scale effectively as the number of applications, developers, and traffic grows.

User Feedback

User feedback has always played a critical role in software applications in two key ways: evaluating the application's performance and informing its development. However, in AI applications, user feedback takes on an even more significant role. User feedback is proprietary data, and data is a competitive advantage. A well-designed user feedback system is necessary to create the data flywheel discussed in Chapter 8.[7]

User feedback can be used not only to personalize models for individual users but also to train future iterations of the models. As data becomes increasingly scarce, proprietary data is more valuable than ever. A product that launches quickly and attracts users early can gather data to continually improve models, making it difficult for competitors to catch up.

It's important to remember that user feedback is user data. Leveraging user feedback requires the same cautions needed when leveraging any data. User privacy should be respected. Users have the right to know how their data is being used.

Extracting Conversational Feedback

Traditionally, feedback can be *explicit or implicit*. Explicit feedback is information users provide in response to explicit requests for feedback in the application, such as thumbs up/thumbs down, upvote/downvote, star rating, or a yes/no answer to the question "Did we solve your problem?" Explicit feedback is fairly standard across applications—there are only so many ways you

can ask a person if they like something. Therefore, explicit feedback is better understood.

Implicit feedback is information inferred from user actions. For example, if someone buys a product recommended to them, it means it was a good recommendation. What can be considered implicit feedback depends on what actions a user can do within each application and is, therefore, highly application-dependent. Foundation models enable a new world of applications and, with them, many genres of implicit feedback.

The conversational interface that many AI applications use makes it easier for users to give feedback. Users can encourage good behaviors and correct errors the same way they would give feedback in daily dialogues. The language that a user uses to give directions to AI can convey feedback about both *the application's performance* and *the user's preference*.

As an example, imagine you're using an AI assistant to help you plan your trip to Australia. You ask the AI to find a hotel for three nights in Sydney. It responds with three recommendations as follows:

```
Here are three hotel suggestions in Sydney for a 3-night stay:

1. Historic Rocks Boutique Hotel (The Rocks)
-Price: ~$400/night
-Neighborhood: Charming streets and close to iconic sights.

2. Stylish Surry Hills House Hotel (Surry Hills)
-Price: ~$200/night
-Neighborhood: Trendy, with vibrant cafes and art galleries.

3. Chill Bondi Beachside Hotel (Bondi Beach)
-Price: ~$300/night
Neighborhood: Beachside, ideal for relaxation and city exploration.
```

How you respond to these three recommendations reveals your preference. For example, if you respond with "Yes book me the one close to galleries", you show an interest in art. On the other hand, the response "Is there nothing under $200?" reveals a price-conscious preference and suggests that the assistant doesn't quite get you yet.

User feedback, extracted from conversations, can be used for evaluation, development, and personalization:

- Evaluation: derive metrics to monitor the application
- Development: train the future models or guide their development
- Personalization: personalize the application to each user

Implicit conversational feedback can be inferred from both the content of user messages and their patterns of communication. Because feedback is blended into daily conversations, it's also challenging to extract. While intuition about conversational cues can help you devise an initial set of signals to look for, rigorous data analysis and user studies are necessary to understand.

While conversational feedback has enjoyed greater attention thanks to the popularity of conversational bots, it had been an active research area for several years before ChatGPT came out. The reinforcement learning community has been trying to get RL algorithms to learn from natural language feedback since the late 2010s, many of them with promising results; see Fu et al. (2019); Goyal et al. (2019); Zhou and Small (2020); and Sumers et al. (2020)). Natural language feedback is also of great interest for early conversational AI applications such as Amazon Alexa (Ponnusamy et al., 2019; Park et al., 2020), Spotify's voice control feature (Xiao et al., 2021), and Yahoo! Voice (Hashimoto and Sassano, 2018).

Natural language feedback

Feedback extracted from the content of messages is called natural language feedback. Here are a couple of natural language feedback signals that tell you how a conversation is going. It's useful to track these signals in production to monitor your application's performance.

Early termination

If a user terminates a response early, e.g., stopping a response generation halfway, exiting the app (for web and mobile apps), telling the model to stop (for voice assistants), or simply leaving the agent hanging (e.g., not responding to the agent with which option you want it to go ahead with), it's likely that the conversation isn't going well.

Error correction

If a user starts their follow-up with "No, ..." or "I meant, ...", the model's response is likely off the mark.

To correct errors, users might try to rephrase their requests. Figure 10-12 shows an example of a user's attempt to correct the model's misunderstanding. Rephrase attempts can be detected using heuristics or ML models.

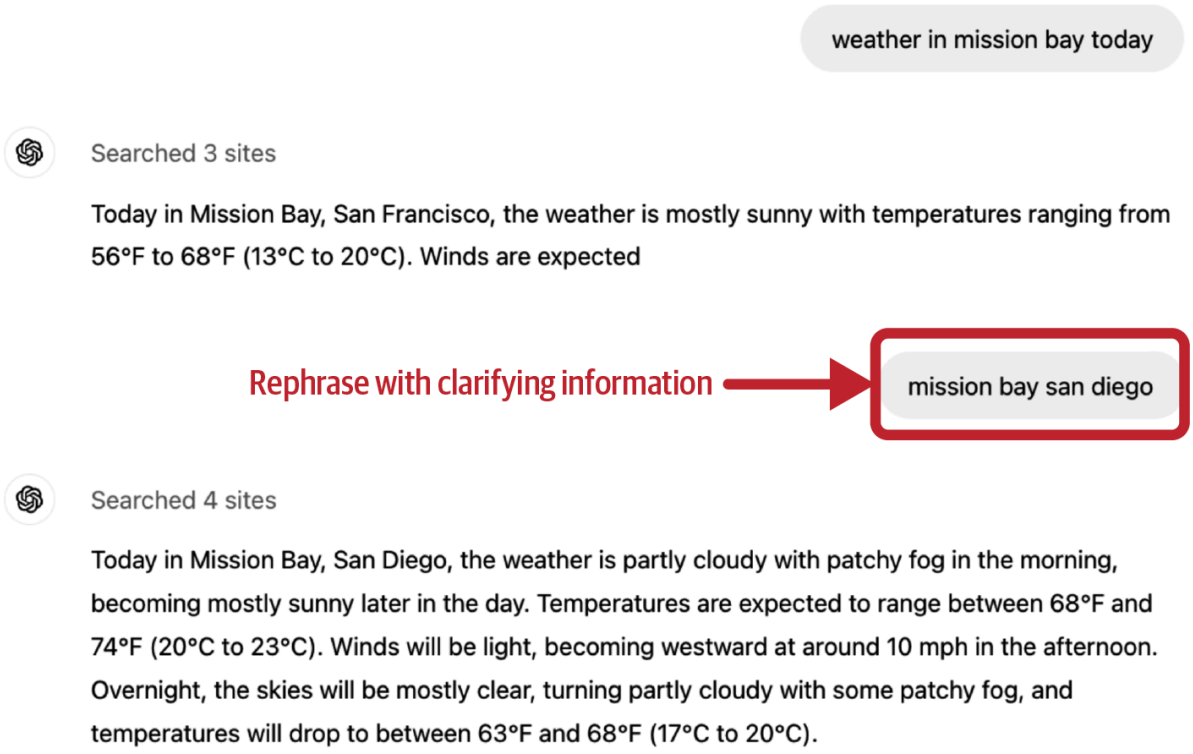

Figure 10-12. Because the user both terminates the generation early and rephrases the question, it can be inferred that the model misunderstood the intent of the original request.

Users can also point out specific things the model should've done differently. For example, if a user asks the model to summarize a story and the model confuses a character, this user can give feedback such as: "Bill is the suspect, not the victim." The model should be able to take this feedback and revise the summary.

This kind of action-correcting feedback is especially common for agentic use cases where users might nudge the agent toward more optional actions. For example, if a user assigns the agent the task of doing market analysis about company XYZ, this user might give feedback such as "You should also check XYZ GitHub page" or "Check the CEO's X profile".

Sometimes, users might want the model to correct itself by asking for explicit confirmation, such as “Are you sure?”, “Check again”, or “Show me the sources”. This doesn’t necessarily mean that the model gives wrong answers. However, it might mean that your model’s answers lack the details the user is looking for. It can also indicate general distrust in your model.

Some applications let users edit the model’s responses directly. For example, if a user asks the model to generate code, and the user corrects the generated code, it’s a very strong signal that the code that got edited isn’t quite right.

User edits also serve as a valuable source of preference data. Recall that preference data, typically in the format of (query, winning response, losing response), can be used to align a model to human preference. Each user edit makes up a preference example, with the original generated response being the losing response and the edited response being the winning response.

Complaints

Often, users just complain about your application’s outputs without trying to correct them. For example, they might complain that an answer is wrong, irrelevant, toxic, lengthy, lacking detail, or just bad. Table 10-1 shows eight groups of natural language feedback resulting from automatic clustering the FITS (Feedback for Interactive Talk & Search) dataset (Xu et al., 2022).

Table 10-1. Feedback types derived from automatic clustering the FITS dataset (Xu et al., 2022). Results from Yuan et al. (2023).

Group	Feedback type	Num.	%
1	Clarify their demand again.	3702	26.54%
2	Complain that the bot (1) does not answer the question or (2) gives irrelevant information or (3) asks the user to find out the answer on their own.	2260	16.20%
3	Point out specific search results that can answer the question.	2255	16.17%
4	Suggest that the bot should use the search results.	2130	15.27%
5	State that the answer is (1) factually incorrect, or (2) not grounded in the search results.	1572	11.27%
6	Point out that the bot's answer is not specific/accurate/complete/detailed.	1309	9.39%
7	Point out that the bot is not confident in its answers and always begins its responses with "I am not sure" or "I don't know".	582	4.17%
8	Complain about repetition/rudeness in bot responses.	137	0.99%

Understanding how the bot fails the user is crucial in making it better. For example, if you know that the user doesn't like verbose answers, you can change the bot's prompt to make it more concise. If the user is unhappy because the answer lacks details, you can prompt the bot to be more specific.

Sentiment

Complaints can also be general expressions of negative sentiments (frustration, disappointment, ridicule, etc.) without explaining the reason why, such

as “Uggh”. This might sound dystopian, but analysis of a user’s sentiments throughout conversations with a bot might give you insights into how the bot is doing. Some call centers track users’ voices throughout the calls. If a user gets increasingly loud, something is wrong. Conversely, if someone starts a conversation angry but ends happily, the conversation might have resolved their issue.

Natural language feedback can also be inferred from the model’s responses. One important signal is the model’s *refusal rate*. If a model says things like “Sorry, I don’t know that one” or “As a language model, I can’t do …”, the user is probably unhappy.

Other conversational feedback

Other types of conversational feedback can be derived from user actions instead of messages.

Regeneration

Many applications let users generate another response, sometimes with a different model. If a user chooses regeneration, it might be because they’re not satisfied with the first response. However, it might also be that the first response is adequate, but the user wants options to compare. This is especially common with creative requests like image or story generation.

Regeneration signals might also be stronger for applications with usage-based billing than those with subscriptions. With usage-based billing, users are less likely to regenerate and spend extra money out of idle curiosity.

Personally, I often choose regeneration for complex requests to ensure the model’s responses are consistent. If two responses give contradicting answers, I can’t trust either.

After regeneration, some applications might explicitly ask to compare the new response with the previous one, as shown in Figure 10-13. This better or worse data, again, can be used for preference finetuning.

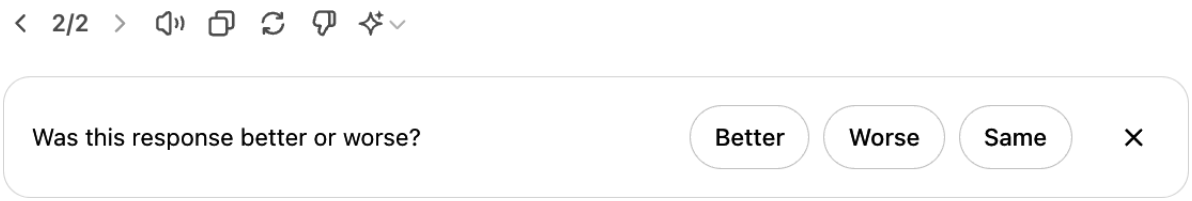

Figure 10-13. ChatGPT asks for comparative feedback when a user regenerates another response.

Conversation organization

The actions a user takes to organize their conversations—such as delete, rename, share, and bookmark—can also be signals. Deleting a conversation is a pretty strong signal that the conversation is bad, unless it's an embarrassing conversation and the user wants to remove its trace. Renaming a conversation suggests that the conversation is good, but the auto-generated title is bad.

Conversation length

Another commonly tracked signal is *the number of turns per conversation*. Whether this is a positive or negative signal depends on the application. For AI companions, a long conversation might indicate that the user enjoys the conversation. However, for chatbots geared toward productivity like customer support, a long conversation might indicate that the bot is inefficient in helping users resolve their issues.

Dialogue diversity

Conversation length can also be interpreted together with *dialogue diversity*, which can be measured by the distinct token or topic count. For example, if the conversation is long but the bot keeps repeating a few lines, the user might be stuck in a loop.

Explicit feedback is easier to interpret, but it demands extra effort from users. Since many users may not be willing to put in this additional work, explicit feedback can be sparse, especially in applications with smaller user bases. Explicit feedback also suffers from response biases. For example, unhappy users might be more likely to complain, causing the feedback to appear more negative than it is.

Implicit feedback is more abundant—what can be considered implicit feedback is limited only by your imagination—but it's noisier. Interpreting implicit signals can be challenging. For example, sharing a conversation can either be a negative or a positive signal. For example, one friend of mine mostly shares conversations when the model has made some glaring mistakes, and another friend mostly shares useful conversations with their coworkers. *It's important to study your users to understand why they do each action*.

Adding more signals can help clarify the intent. For example, if the user rephrases their question after sharing a link, it might indicate that the conversation didn't meet their expectations. Extracting, interpreting, and leveraging implicit responses from conversations is a small but growing area of research.[8]

Feedback Design

If you were unsure of what feedback to collect, I hope that the last section gave you some ideas.

This section discusses when and how to collect this valuable feedback.

When to collect feedback

Feedback can and should be collected throughout the user journey. Users should have the option to give feedback, especially to report errors, whenever this need arises. The feedback collection option, however, should be nonintrusive. It shouldn't interfere with the user workflow. Here are a few places where user feedback might be particularly valuable.

In the beginning

When a user has just signed up, user feedback can help calibrate the application for the user. For example, a face ID app first must scan your face to work. A voice assistant might ask you to read a sentence out loud to recognize your voice for wake words (words that activate a voice assistant, like "Hey Google"). A language learning app might ask you a few questions to gauge your skill level. For some applications, such as face ID, calibration is necessary. For other applications, however, initial feedback should be optional, as

it creates friction for users to try out your product. If a user doesn't specify their preference, you can fall back to a neutral option and calibrate over time.

When something bad happens

When the model hallucinates a response, blocks a legitimate request, generates a compromising image, or takes too long to respond, users should be able to notify you of these failures. You can give users the option to downvote a response, regenerate with the same model, or change to another model. Users might just give conversational feedback like "You're wrong", "Too cliche", or "I want something shorter".

Ideally, when your product makes mistakes, users should still be able to accomplish their tasks. For example, if the model wrongly categorizes a product, users can edit the category. Let users collaborate with the AI. If that doesn't work, let them collaborate with humans. Many customer support bots offer to transfer users to human agents if the conversation drags on or if users seem frustrated.

An example of human–AI collaboration is the *inpainting* functionality for image generation.[9] If a generated image isn't exactly what the user needs, they can select a region of the image and describe with a prompt how to make it better. Figure 10-14 shows an example of inpainting with DALL-E (OpenAI, 2021). This feature allows users to get better results while giving developers high-quality feedback.

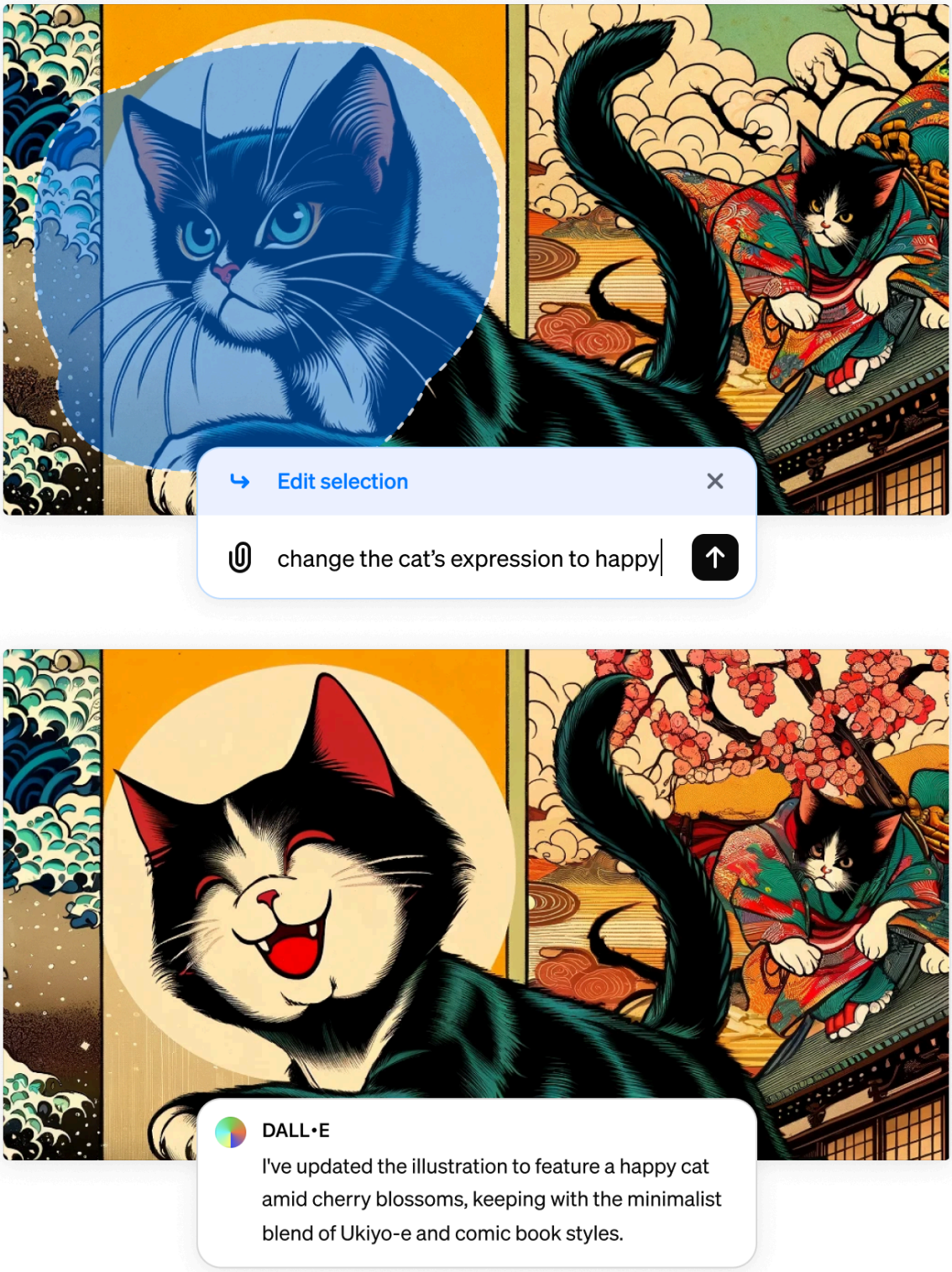

Figure 10-14. An example of how inpainting works in DALL-E. Image by OpenAI.

When the model has low confidence

When a model is uncertain about an action, you can ask the user for feedback to increase its confidence. For example, given a request to summarize a paper, if the model is uncertain whether the user would prefer a short, high-level summary or a detailed section-by-section summary, the model can output both summaries side by side, assuming that generating two summaries

doesn't increase the latency for the user. The user can choose which one they prefer. Comparative signals like this can be used for preference finetuning. An example of comparative evaluation in production is shown in Figure 10-15.

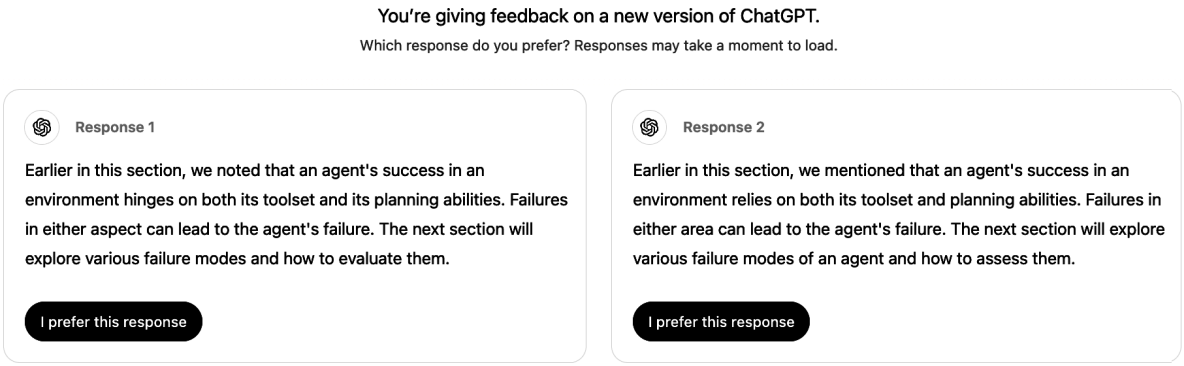

Figure 10-15. Side-by-side comparison of two ChatGPT responses.

Showing two full responses for the user to choose means asking that user for explicit feedback. Users might not have time to read two full responses or care enough to give thoughtful feedback. This can result in noisy votes. Some applications, like Google Gemini, show only the beginning of each response, as shown in Figure 10-16. Users can click to expand the response they want to read. It's unclear, however, whether showing full or partial responses side by side gives more reliable feedback.[10]

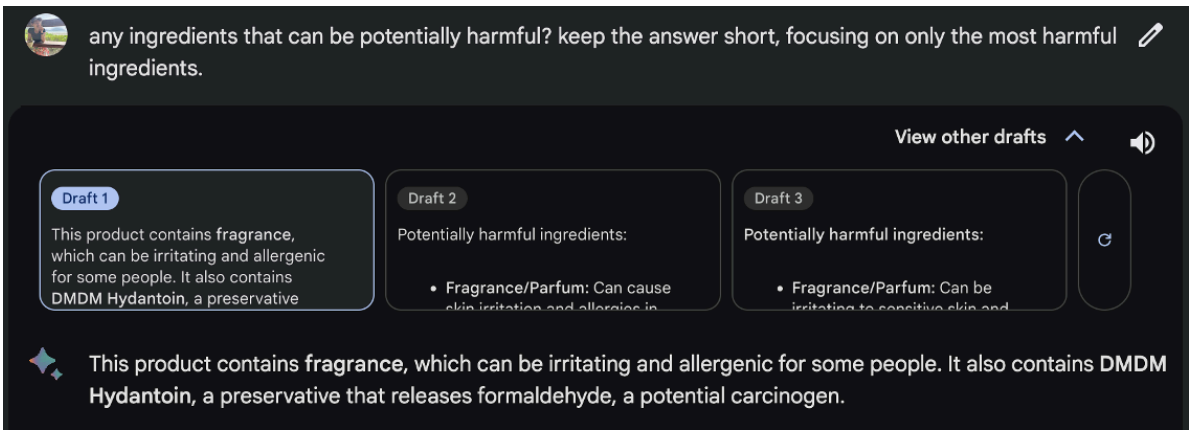

Figure 10-16. Google Gemini shows partial responses side by side for comparative feedback. Users have to click on the response they want to read more about, which gives feedback about which response they find more promising.

Another example is a photo organization application that automatically tags your photos, so that it can respond to queries like "Show me all the photos of X". When unsure if two people are the same, it can ask you for feedback, as Google Photos does in Figure 10-17.

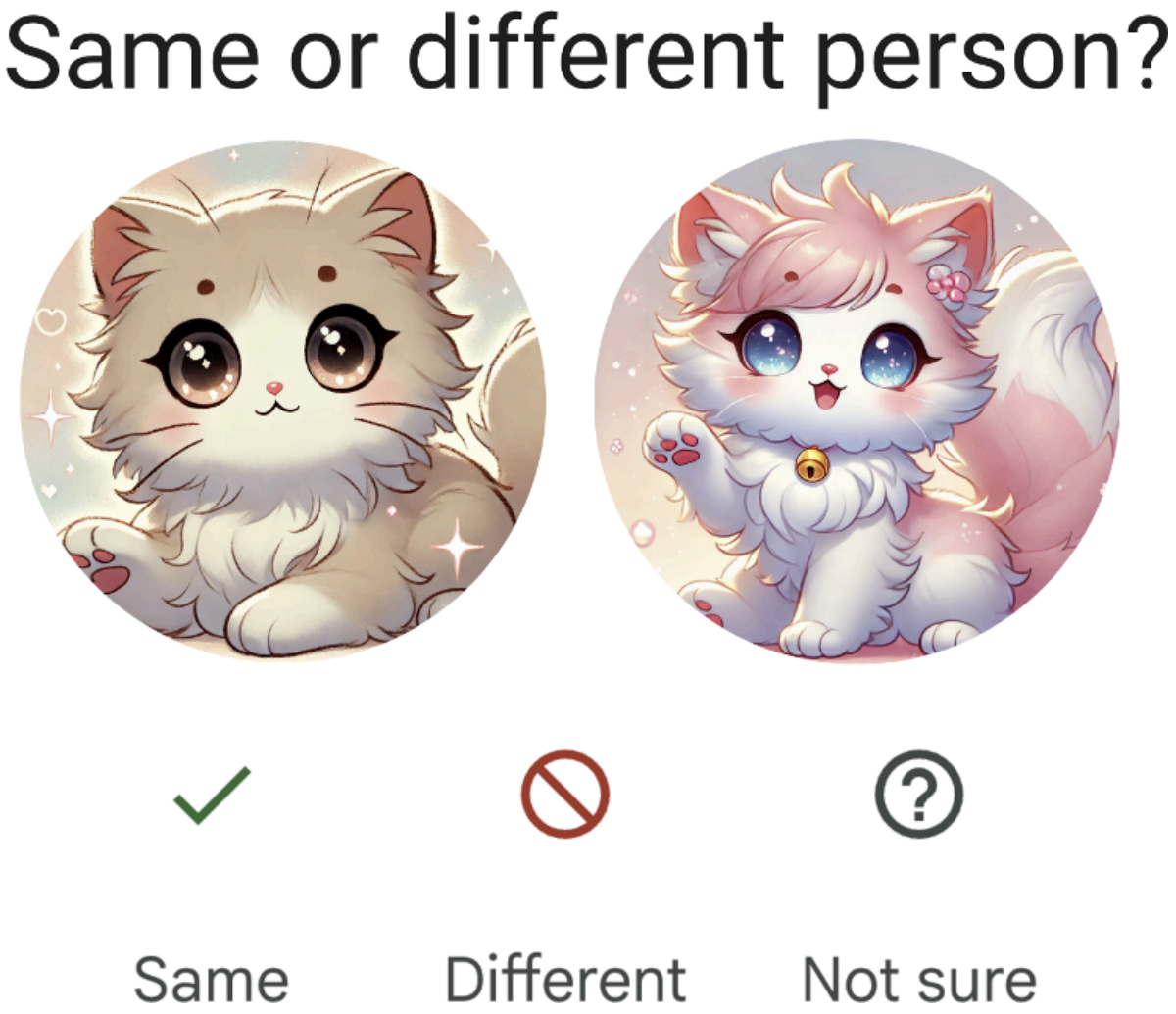

Figure 10-17. Google Photos asks for user feedback when unsure. The two cat images were generated by ChatGPT.

You might wonder: how about feedback when something good happens? Actions that users can take to express their satisfaction include thumbs up, favoriting, or sharing. However, Apple's human interface guideline warns against asking for both positive and negative feedback. Your application should produce good results by default. Asking for feedback on good results might give users the impression that good results are exceptions. Ultimately, if users are happy, they continue using your application.

However, many people I've talked to believe users should have the option to give feedback when they encounter something amazing. A product manager for a popular AI-powered product mentioned that their team needs positive feedback because it reveals the features users love enough to give enthusiastic feedback about. This allows the team to concentrate on refining a small set of high-impact features rather than spreading resources across many with minimal added value.

Some avoid asking for positive feedback out of concern it may clutter the interface or annoy users. However, this risk can be managed by limiting the frequency of feedback requests. For example, if you have a large user base, showing the request to only 1% of users at a time could help gather sufficient feedback without disrupting the experience for most users. Keep in mind that the smaller the percentage of users asked, the greater the risk of feedback biases. Still, with a large enough pool, the feedback can provide meaningful product insights.

How to collect feedback

Feedback should seamlessly integrate into the user's workflow. It should be easy for users to provide feedback without extra work. Feedback collection shouldn't disrupt user experience and should be easy to ignore. There should be incentives for users to give good feedback.

One example often cited as good feedback design is from the image generator app Midjourney. For each prompt, Midjourney generates a set of (four) images and gives the user the following options, as shown in Figure 10-18:

1. Generate an unscaled version of any of these images.
2. Generate variations for any of these images.
3. Regenerate.

All these options give Midjourney different signals. Options 1 and 2 tell Midjourney which of the four photos is considered by the user to be the most promising. Option 1 gives the strongest positive signal about the chosen photo. Option 2 gives a weaker positive signal. Option 3 signals that none of the photos is good enough. However, users might choose to regenerate even if the existing photos are good just to see what else is possible.

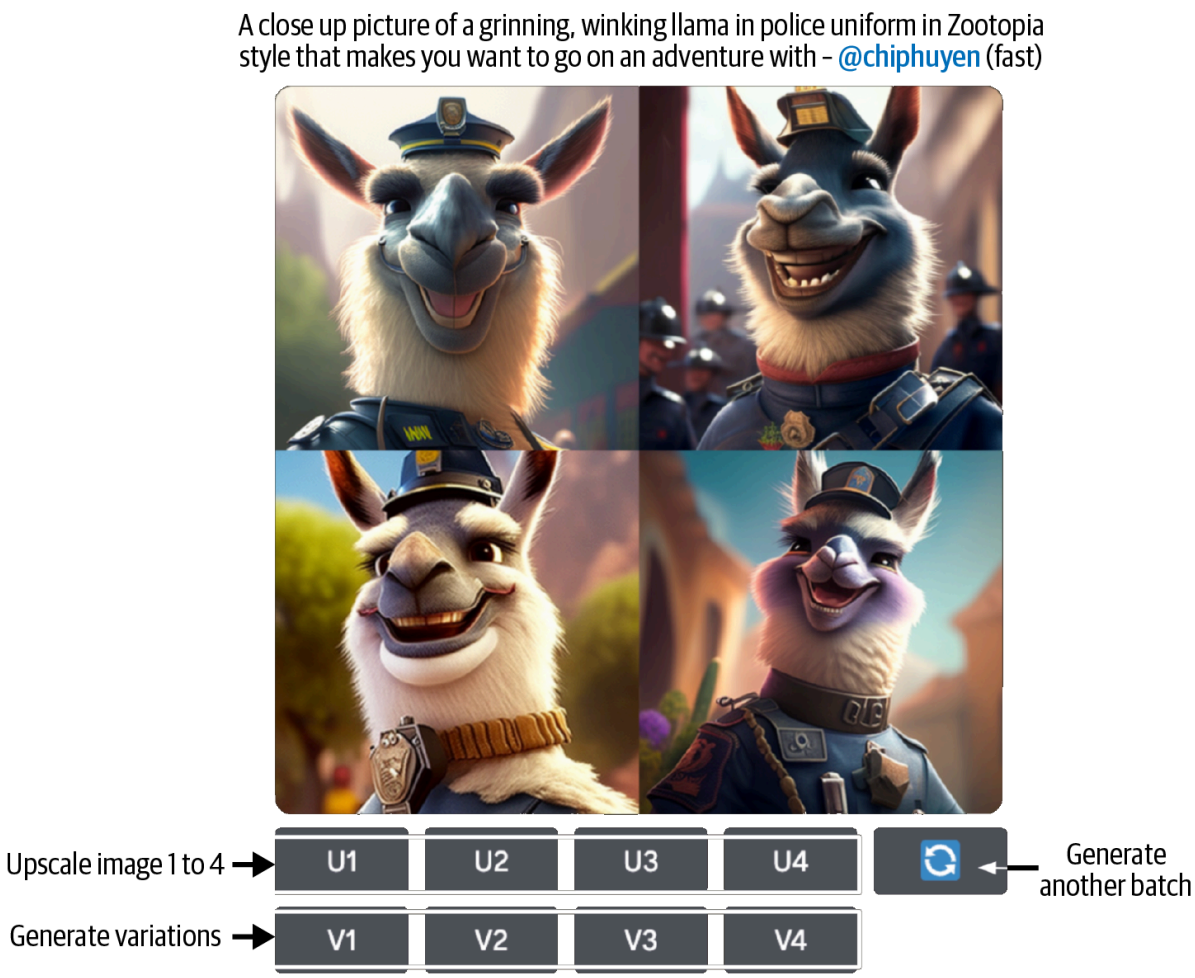

Figure 10-18. Midjourney's workflow allows the app to collect implicit feedback.

Code assistants like GitHub Copilot might show their drafts in lighter colors than the final texts, as shown in Figure 10-19. Users can use the Tab key to accept a suggestion or simply continue typing to ignore the suggestion, both providing feedback.

```
 9 class Solution:
10     def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
11         """
12         Do not return anything, modify nums1 in-place instead.
13         """
14             < 1/2 > Accept Tab Accept Word ⌘ → ···
15         while p1 >= 0 and p2 >= 0:
               if nums1[p1] > nums2[p2]:
                   nums1[p] = nums1[p1]
                   p1 -= 1
               else:
                   nums1[p] = nums2[p2]
                   p2 -= 1
               p -= 1
```

Figure 10-19. GitHub Copilot makes it easy to both suggest and reject a suggestion.

One of the biggest challenges of standalone AI applications like ChatGPT and Claude is that they aren't integrated into the user's daily workflow, making it hard to collect high-quality feedback the way integrated products like GitHub Copilot can. For example, if Gmail suggests an email draft, Gmail can track how this draft is used or edited. However, if you use ChatGPT to write an email, ChatGPT doesn't know whether the generated email is actually sent.

The feedback alone might be helpful for product analytics. For example, seeing just the thumbs up/thumbs down information is useful for calculating how often people are happy or unhappy with your product. For deeper analysis, though, you would need context around the feedback, such as the previous 5 to 10 dialogue turns. This context can help you figure out what went wrong. However, getting this context might not be possible without explicit user consent, especially if the context might contain personally identifiable information.

For this reason, some products include terms in their service agreements that allow them to access user data for analytics and product improvement. For applications without such terms, user feedback might be tied to a user data donation flow, where users are asked to donate (e.g., share) their recent interaction data along with their feedback. For example, when submitting feedback, you might be asked to check a box to share your recent data as context for this feedback.

Explaining to users how their feedback is used can motivate them to give more and better feedback. Do you use a user's feedback to personalize the product to this user, to collect statistics about general usage, or to train a new model? If users are concerned about privacy, reassure them that their data won't be used to train models or won't leave their device (only if these are true).

Don't ask users to do the impossible. For example, if you collect comparative signals from users, don't ask them to choose between two options they don't understand. For example, I was once stumped when ChatGPT asked me to choose between two possible answers to a statistical question, as shown in Figure 10-20. I wish there was an option for me to say, "I don't know".

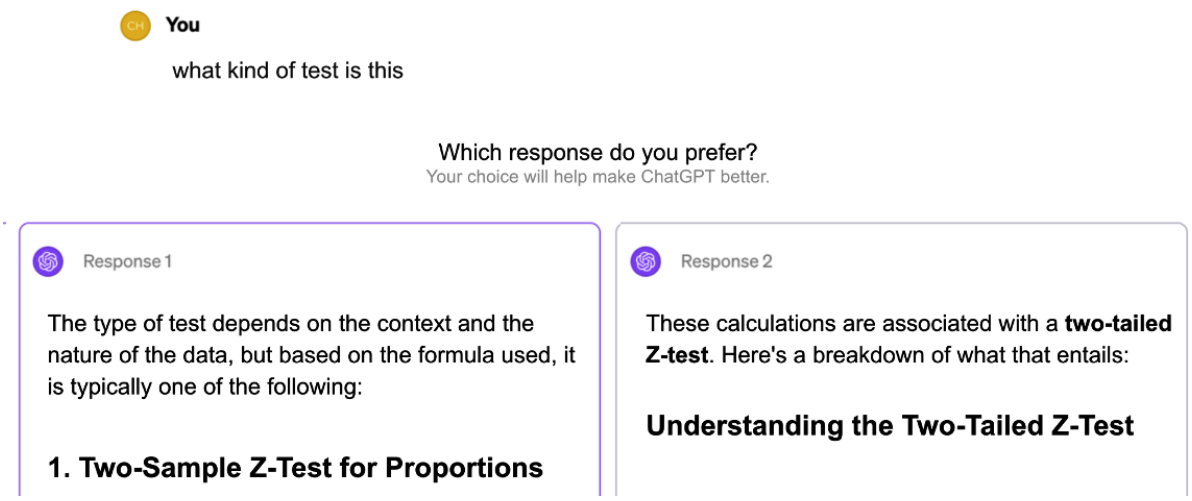

Figure 10-20. An example of ChatGPT asking a user to select the response the user prefers. However, for mathematical questions like this, the right answer shouldn't be a matter of preference.

Add icons and tooltips to an option if they help people understand it. Avoid a design that can confuse users. Ambiguous instructions can lead to noisy feedback. I once hosted a GPU optimization workshop, using Luma to collect feedback. When I was reading the negative feedback, I was confused. Even though the responses were positive, the star ratings were 1/5. When I dug deeper, I realized that Luma used emojis to represent numbers in their feedback collection form, but the angry emoji, corresponding to a one-star rating, was put where the five-star rating should be, as shown in Figure 10-21.

Be mindful of whether you want users' feedback to be private or public. For example, if a user likes something, do you want this information shown to other users? In its early days, Midjourney's feedback—someone choosing to upscale an image, generate variations, or regenerate another batch of images—was public.

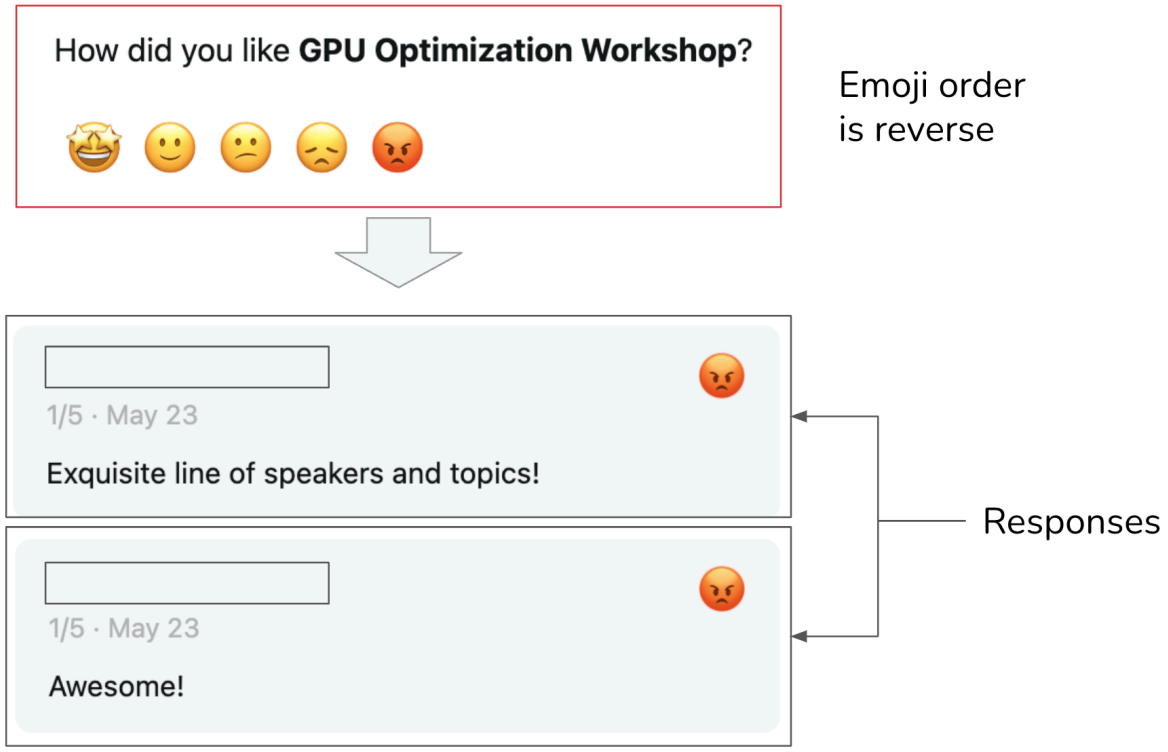

Figure 10-21. Because Luma put the angry emoji, corresponding to a one-star rating, where a five-star rating should've been, some users mistakenly picked it for positive reviews.

The visibility of a signal can profoundly impact user behavior, user experience, and the quality of the feedback. Users tend to be more candid in private—there's a lower chance of their activities being judged[11]—which can result in higher-quality signals. In 2024, X (formerly Twitter) made "likes" private. Elon Musk, the owner of X, claimed a significant uptick in the number of likes after this change.

However, private signals can reduce discoverability and explainability. For example, hiding likes prevents users from finding tweets their connections have liked. If X recommends tweets based on the likes of the people you follow, hiding likes could result in users' confusion about why certain tweets appear in their feeds.

Feedback Limitations

There's no doubt of the value of user feedback to an application developer. However, feedback isn't a free lunch. It comes with its own limitations.

Biases

Like any other data, user feedback has biases. It's important to understand these biases and design your feedback system around them. Each application has its own biases. Here are a few examples of feedback biases to give you an idea of what to look out for:

Leniency bias

Leniency bias is the tendency for people to rate items more positively than warranted, often to avoid conflict because they feel compelled to be nice or because it's the easiest option. Imagine you're in a hurry, and an app asks you to rate a transaction. You aren't happy with the transaction, but you know that if you rate it negatively, you'll be asked to provide reasons, so you just choose positive to be done with it. This is also why you shouldn't make people do extra work for your feedback.

On a five-star rating scale, four and five stars are typically meant to indicate a good experience. However, in many cases, users may feel pressured to give five-star ratings, reserving four stars for when something goes wrong. According to Uber, in 2015, the average driver's rating was 4.8, with scores below 4.6 putting drivers at risk of being deactivated.

This bias isn't necessarily a dealbreaker. Uber's goal is to differentiate good drivers from bad drivers. Even with this bias, their rating system seems to help them achieve this goal. It's essential to look at the distribution of your user ratings to detect this bias.

If you want more granular feedback, removing the strong negative connotation associated with low ratings can help people break out of this bias. For example, instead of showing users numbers one to five, show users options such as the following:

- "Great ride. Great driver."
- "Pretty good."
- "Nothing to complain about but nothing stellar either."
- "Could've been better."

- "Don't match me with this driver again."[12]

Randomness

Users often provide random feedback, not out of malice, but because they lack motivation to give more thoughtful input. For example, when two long responses are shown side by side for comparative evaluation, users might not want to read both of them and just click on one at random. In the case of Midjourney, users might also randomly choose one image to generate variations.

Position bias

The position in which an option is presented to users influences how this option is perceived. Users are generally more likely to click on the first suggestion than the second. If a user clicks on the first suggestion, this doesn't necessarily mean that it's a good suggestion.

When designing your feedback system, this bias can be mitigated by randomly varying the positions of your suggestions or by building a model to compute a suggestion's true success rate based on its position.

Preference bias

Many other biases can affect a person's feedback, some of which have been discussed in this book. For example, people might prefer the longer response in a side-by-side comparison, even if the longer response is less accurate—length is easier to notice than inaccuracies. Another bias is *recency bias*, where people tend to favor the answer they see last when comparing two answers.

It's important to inspect your user feedback to uncover its biases. Understanding these biases will help you interpret the feedback correctly, avoiding misleading product decisions.

Degenerate feedback loop

Keep in mind that user feedback is incomplete. You only get feedback on what you show users.

In a system where user feedback is used to modify a model's behavior, *degenerate feedback loops* can arise. A degenerate feedback loop can happen when the predictions themselves influence the feedback, which, in turn, influences the next iteration of the model, amplifying initial biases.

Imagine you're building a system to recommend videos. The videos that rank higher show up first, so they get more clicks, reinforcing the system's belief that they're the best picks. Initially, the difference between the two videos, A and B, might be minor, but because A was ranked slightly higher, it got more clicks, and the system kept boosting it. Over time, A's ranking soared, leaving B behind. This feedback loop is why popular videos stay popular, making it tough for new ones to break through. This issue is known as "exposure bias," "popularity bias," or "filter bubbles," and it's a well-studied problem.

A degenerate feedback loop can alter your product's focus and use base. Imagine that initially, a small number of users give feedback that they like cat photos. The system picks up on this and starts generating more photos with cats. This attracts cat lovers, who give more feedback that cat photos are good, encouraging the system to generate even more cats. Before long, your application becomes a cat haven. Here, I use cat photos as an example, but the same mechanism can amplify other biases, such as racism, sexism, and preference for explicit content.

Acting on user feedback can also turn a conversational agent into, for lack of a better word, a liar. Multiple studies have shown that training a model on user feedback can teach it to give users what it thinks users want, even if that isn't what's most accurate or beneficial (Stray, 2023). Sharma et al. (2023) show that AI models trained on human feedback tend toward. sycophancy. They are more likely to present user responses matching this user's view.

User feedback is crucial for improving user experience, but if used indiscriminately, it can perpetuate biases and destroy your product. Before incorporating feedback into your product, make sure that you understand the limitations of this feedback and its potential impact.

Summary

If each previous chapter focused on a specific aspect of AI engineering, this chapter looked into the process of building applications on top of foundation models as a whole.

The chapter consisted of two parts. The first part discussed a common architecture for AI applications. While the exact architecture for an application might vary, this high-level architecture provides a framework for understanding how different components fit together. I used the step-by-step approach in building this architecture to discuss the challenges at each step and the techniques you can use to address them.

While it's necessary to separate components to keep your system modular and maintainable, this separation is fluid. There are many ways components can overlap in functionalities. For example, guardrails can be implemented in the inference service, the model gateway, or as a standalone component.

Each additional component can potentially make your system more capable, safer, or faster but will also increase the system's complexity, exposing it to new failure modes. One integral part of any complex system is monitoring and observability. Observability involves understanding how your system fails, designing metrics and alerts around failures, and ensuring that your system is designed in a way that makes these failures detectable and traceable. While many observability best practices and tools from software engineering and traditional machine learning are applicable to AI engineering applications, foundation models introduce new failure modes, which require additional metrics and design considerations.

At the same time, the conversational interface enables new types of user feedback, which you can leverage for analytics, product improvement, and the data flywheel. The second part of the chapter discussed various forms of conversational feedback and how to design your application to effectively collect it.

Traditionally, user feedback design has been seen as a product responsibility rather than an engineering one, and as a result, it is often overlooked by engi-

neers. However, since user feedback is a crucial source of data for continuously improving AI models, more AI engineers are now becoming involved in the process to ensure they receive the data they need. This reinforces the idea from Chapter 1 that, compared to traditional ML engineering, AI engineering is moving closer to product. This is because of both the increasing importance of data flywheel and product experience as competitive advantages.

Many AI challenges are, at their core, system problems. To solve them, it's often necessary to step back and consider the system as a whole. A single problem might be addressed by different components working independently, or a solution could require the collaboration of multiple components. A thorough understanding of the system is essential to solving real problems, unlocking new possibilities, and ensuring safety.

1 An example is when a Samsung employee put Samsung's proprietary information into ChatGPT, accidentally leaking the company's secrets.

2 It's possible that users ask the model to return an empty response.

3 A few early readers told me that the idea of ignoring guardrails in favor of latency gave them nightmares.

4 As of this writing, the aggregated market capitalization of a few of the largest observability companies (Datadog, Splunk, Dynatrace, New Relic) is close to $100 billion.

5 My book, *Designing Machine Learning Systems* (O'Reilly, 2022), also has a chapter on monitoring. An early draft of the chapter is available on my blog at "Data Distribution Shifts and Monitoring".

6 Because of this, some orchestrator tools want to be gateways. In fact, so many tools seem to want to become end-to-end platforms that do everything.

7 One key disadvantage of launching an open source application instead of a commercial application is that it's a lot harder to collect user feedback. Users can take your open source application and deploy it themselves, and you have no idea how the application is used.

8 Not only can you collect feedback about AI applications, you can use AI to analyze feedback, too.

[9] I wish there were inpainting for text-to-speech. I find text-to-speech works well 95% of the time, but the other 5% can be frustrating. AI might mispronounce a name or fail to pause during dialogues. I wish there were apps that let me edit just the mistakes instead of having to regenerate the whole audio.

[10] When I ask this question at events I speak at, the responses are conflicted. Some people think showing full responses gives more reliable feedback because it gives users more information to make a decision. At the same time, some people think that once users have read full responses, there's no incentive for them to click on the better one.

[11] See "Ted Cruz Blames Staffer for 'Liking' Porn Tweet" (Nelson and Everett, *POLITICO*, September 2017) and "Kentucky Senator Whose Twitter Account 'Liked' Obscene Tweets Says He Was Hacked" (Liam Niemeyer, WKU Public Radio, March 2023).

[12] The options suggested here are only to show how options can be rewritten. They haven't been validated.