arize

Large Language Model Observability 101

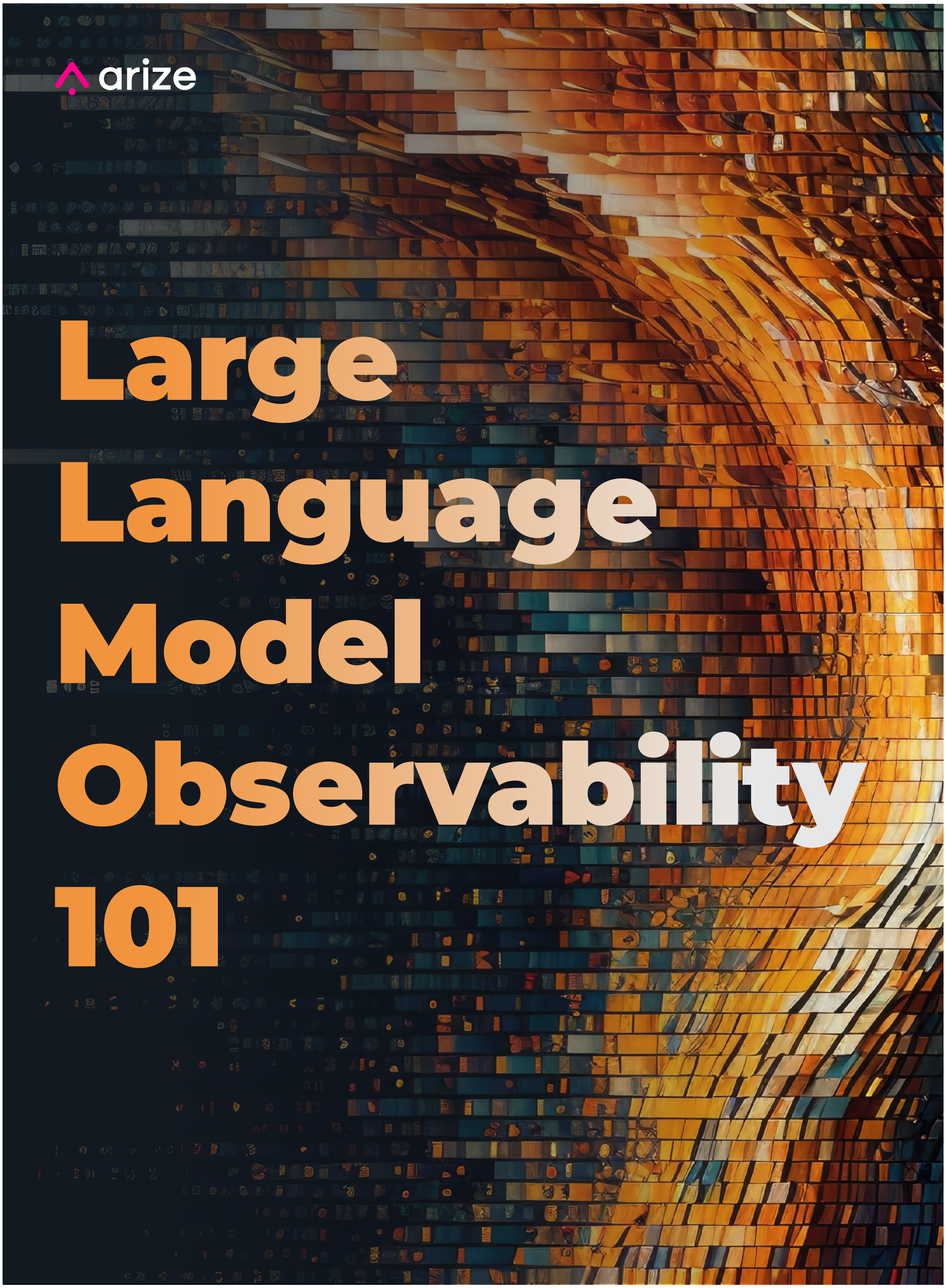


Table of contents

Introduction

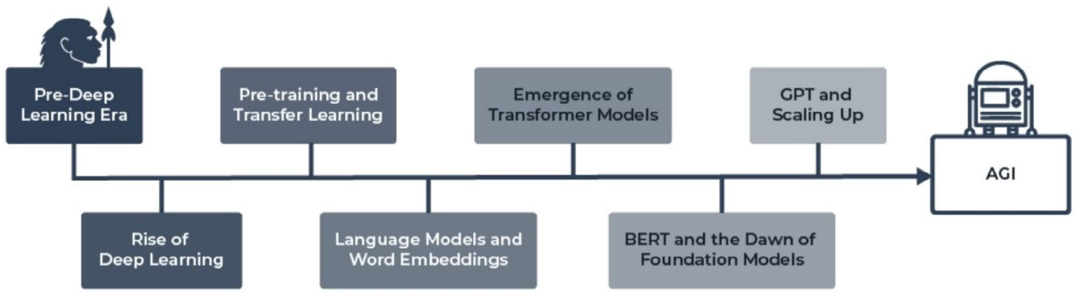
A new era of AI is taking hold. According to a recent survey, over two-thirds (66.9%) of developers and machine learning teams are planning production deployments of large language model (LLM) applications in the next 12 months or “as fast as possible” – and 14.1% are already in production. With deployment made easier due to abstraction, inherent system complexity makes it essential that you have a good understanding of how every element of the system is performing. This ebook covers how to develop and roll out LLM applications that are both reliable and responsible.

Foundation Models

Foundation models are large-scale machine learning models pre-trained on extensive datasets. These models are trained to learn general-purpose representations across various data modalities, including text, images, audio, and video. Their key strengths lie in their size, pre-training, self-supervised learning, generalization, and adaptability.

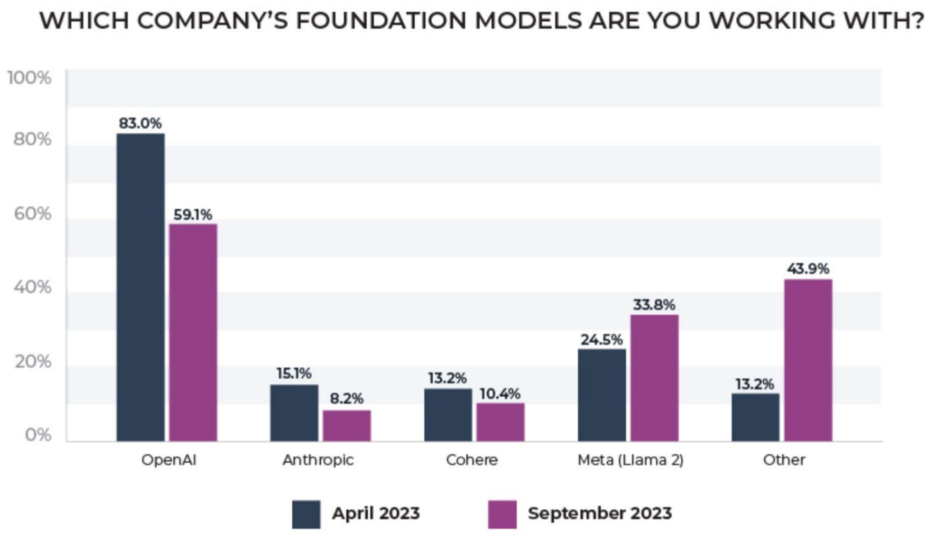
Why Are Foundation Models Growing In Popularity?

The emergence of foundation models might appear sudden, but in reality, they are the outcome of an intricate evolution in the field of AI. Tracing their lineage back to the diverse techniques, models, and groundbreaking developments in machine learning, these models represent the pinnacle of AI achievements to date. From the early days of hand-crafted features to the era of BERT and GPT, foundation models have come a long way. They have reshaped the AI landscape and are continually sculpting the future of machine learning.


Pre-Deep Learning Era
Rise of Deep Learning
Pre-training and Transfer Learning
Language Models and Word Embeddings
Emergence of Transformer Models
BERT and the Dawn of Foundation Models
GPT and Scaling Up
AGI

What Are the Most Popular Models?

While OpenAI still dominates with 59.1% of technical teams relying on the company's LLMs, Meta's Llama 2 and other alternatives are becoming more popular. In the "Other" category, Google PaLM 2 leads in adoption (20.7% of those surveyed), followed by Databricks (Dolly) at 14.9% and MosaicML at 5.6%.



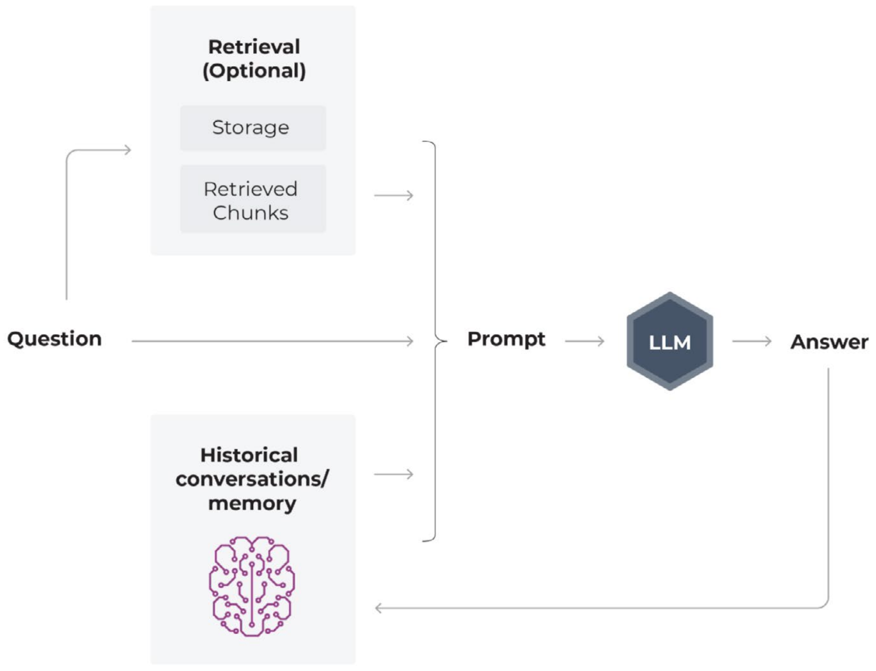
Common Use Cases

What Are the Typical Use Cases for LLMs?

There are many use cases that are common across LLM applications. We'll look at a small sampling here.

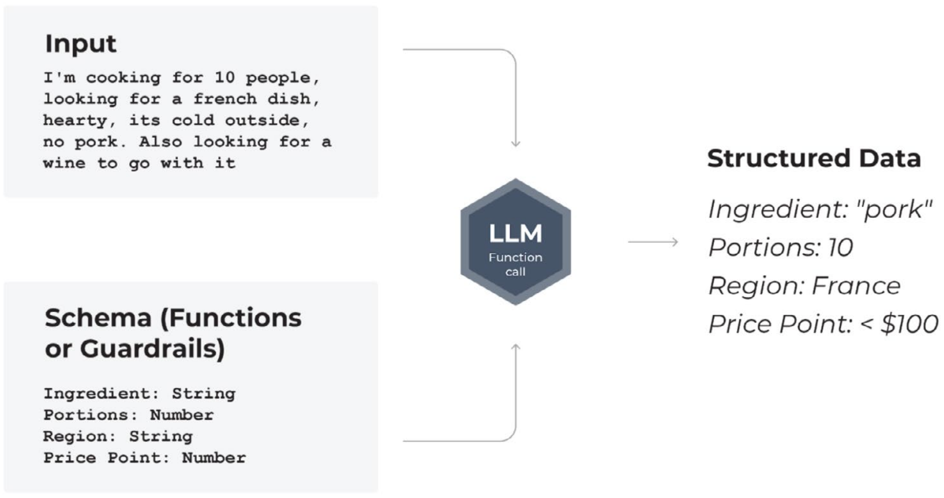
Chatbots

Since ChatGPT initially introduced the masses to LLMs through its chatbot, this use case is very common. It consists of the user asking a question, the system retrieving information to enrich the prompt, and then the LLM generating a response.



Structured Data Extraction

In a structured data extraction use case, the LLM receives unstructured input together with a schema and outputs a structured representation of the information. This is often useful in the context of a larger software system.



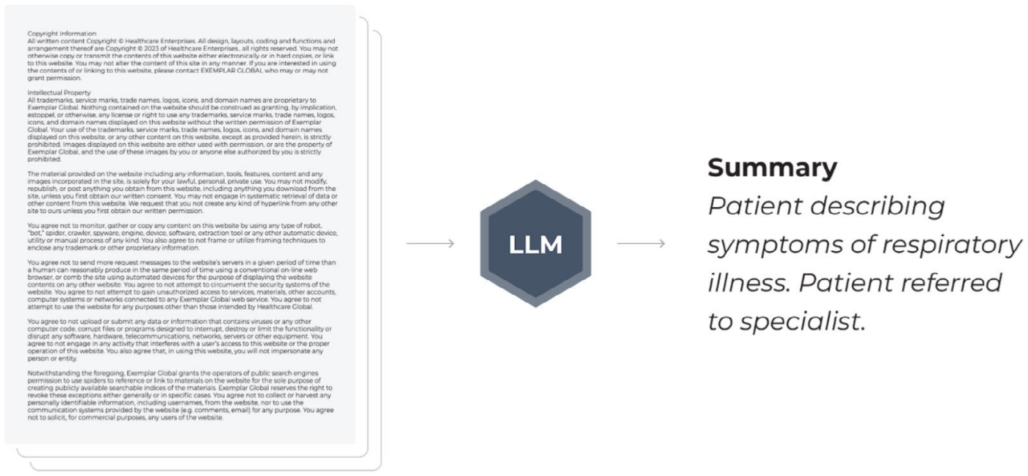
```
I'm cooking for 10 people,
looking for a french dish,
hearty, its cold outside,
no pork. Also looking for a
wine to go with it
```

```
Ingredient: String
Portions: Number
Region: String
Price Point: Number
```

Ingredient: "pork"
Portions: 10
Region: France
Price Point: < $100

Summarization

Natural language processing (NLP) has long searched for good summarization solutions. Many of the traditional extractive and abstractive summarization techniques have now been superseded by LLMs.



Copyright Information
All written content Copyright © Healthcare Enterprises. All design, layouts, coding and functions and arrangement thereof are Copyright © 2023 of Healthcare Enterprises., all rights reserved. You may not otherwise copy or transmit the contents of this website either electronically or in hard copies, or link to this website. You may not alter the content of this site in any manner. If you are interested in using the contents of or linking to this website, please contact EXEMPLAR GLOBAL who may or may not grant permission.

Intellectual Property
All trademarks, service marks, trade names, logos, icons, and domain names are proprietary to Exemplar Global. Nothing contained on the website should be construed as granting, by implication, estoppel, or otherwise, any license or right to use any trademarks, service marks, trade names, logos, icons, and domain names displayed on this website without the written permission of Exemplar Global. Your use of the trademarks, service marks, trade names, logos, icons, and domain names displayed on this website, or any other content on this website, except as provided herein, is strictly prohibited. Images displayed on this website are either used with permission, or are the property of Exemplar Global, and the use of these images by you or anyone else authorized by you is strictly prohibited.

The material provided on the website including any information, tools, features, content and any images incorporated in the site, is solely for your lawful, personal, private use. You may not modify, republish, or post anything you obtain from this website, including anything you download from the site, unless you first obtain our written consent. You may not engage in systematic retrieval of data or other content from this website. We request that you not create any kind of hyperlink from any other site to ours unless you first obtain our written permission.

You agree not to monitor, gather or copy any content on this website by using any type of robot, "bot," spider, crawler, spyware, engine, device, software, extraction tool or any other automatic device, utility or manual process of any kind. You also agree to not frame or utilize framing techniques to enclose any trademark or other proprietary information.

You agree not to send more request messages to the website's servers in a given period of time than a human can reasonably produce in the same period of time using a conventional on-line web browser, or comb the site using automated devices for the purpose of displaying the website contents on any other website. You agree to not attempt to circumvent the security systems of the website. You agree to not attempt to gain unauthorized access to services, materials, other accounts, computer systems or networks connected to any Exemplar Global web service. You agree to not attempt to use the website for any purposes other than those intended by Healthcare Global.

You agree to not upload or submit any data or information that contains viruses or any other computer code, corrupt files or programs designed to interrupt, destroy or limit the functionality or disrupt any software, hardware, telecommunications, networks, servers or other equipment. You agree to not engage in any activity that interferes with a user's access to this website or the proper operation of this website. You also agree that, in using this website, you will not impersonate any person or entity.

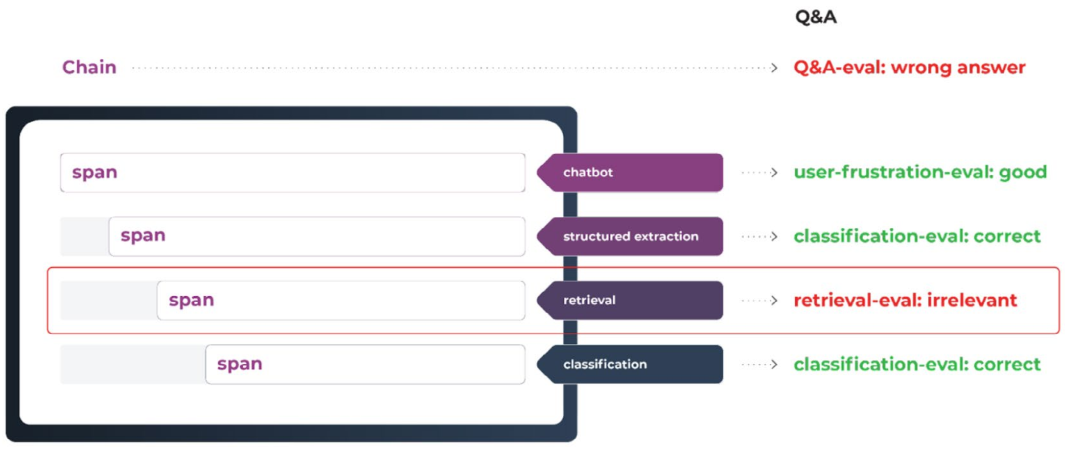
Notwithstanding the foregoing, Exemplar Global grants the operators of public search engines permission to use spiders to reference or link to materials on the website for the sole purpose of creating publicly available searchable indices of the materials. Exemplar Global reserves the right to revoke these exceptions either generally or in specific cases. You agree not to collect or harvest any personally identifiable information, including usernames, from the website, nor to use the communication systems provided by the website (e.g. comments, email) for any purpose. You agree not to solicit, for commercial purposes, any users of the website.

Patient describing symptoms of respiratory illness. Patient referred to specialist.

Other Use Cases

In addition to these, there are many more specialized use cases, like code generation, web scraping, and tagging and labeling.

LLMs are also employed in more complex use cases like Q&A assistant and chat-to-pay. These are composed of steps (spans) that achieve a higher-order objective. Spans can be other LLM use cases, traditional machine learning systems, or non-ML software-defined tools (like a calculator).


Q&A
Chain
Q&A-eval: wrong answer
span
chatbot
user-frustration-eval: good
span
structured extraction
classification-eval: correct
span
retrieval
retrieval-eval: irrelevant
span
classification
classification-eval: correct

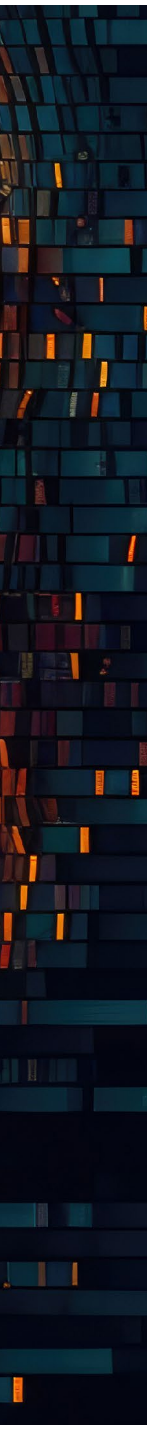
These workflows are useful but also very complex because they require proper orchestration on multiple systems, any one of which can have unique problems. These workflows do not require much code, but do not confuse that brevity for simplicity. Even a few lines can kick off very long chains of computation with multiple opportunities for error (see inset).

That complexity leads to the difficulties experienced by today's LLM projects as they progress from "Twitter Demo" to customer use.

```
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI
from langchain.embeddings import OpenAIEmbeddings
from langchain.retrievers import KNNRetriever

embeddings = OpenAIEmbeddings(model="text-
embedding-ada-002")

knn_retriever = KNNRetriever(
    index=vectors,
    texts=texts,
    embeddings=OpenAIEmbeddings(),
)

llm = ChatOpenAI(model_name="gpt-3.5-turbo")
chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="map_reduce",
    retriever=knn_retriever,
)
```

What Is LLM Observability?

What Issues Arise With LLMs?



Here are several common issues.

- **Hallucination:** The model's objective in training is to predict the next few characters. Accuracy of the responses is more of a side effect. As such, hallucinations, or made-up responses not grounded in facts, are common and unpredictable. You use an LLM as a shortcut, but if you have to double check everything it says, it may not be that useful. If you don't check, however, you may get into real trouble.
- **Proliferation of calls:** Even solving hallucination issues can lead to greater problems. For example, one way around the above-mentioned problem is Reflexion, which asks an LLM to analyze its own results. This technique is powerful but it makes the already-complicated system even more so. Now instead of one call, you have a whole chain of calls. This is true at every span, so for complex use cases we talked about above, there are multiple calls inside multiple spans.
- **Proprietary data:** When you add proprietary data to the mix, things get even more interesting. The reality is that much of the data we need to answer complex questions is proprietary. Access control systems on LLMs are not as robust as they are in traditional software. It is possible for proprietary information to accidentally find its way into a response.
- **Quality of response:** Response quality can often be suboptimal for other reasons too. For example the tone can be wrong, or the amount of detail can be inappropriate. It is very difficult to control the quality of largely unstructured responses.
- **Cost:** Then there is the elephant in the room: cost. All of those spans, Reflexion, and calls to LLMs can add up to a significant bill.
- **Third-party models:** LLMs accessed through third-party providers can change over time. The API can change, new models can be added, or new safeguards can be put in place, all of which may cause models to behave differently.
- **Limited competitive advantage:** The bigger problem, perhaps, is that LLMs are hard to train and maintain. Therefore your LLM model is the same as that of all of your competitors, so what really differentiates you is prompt engineering and connection to proprietary data. You want to make sure you are using them well.

These are hard problems, but fortunately the first step in tackling all of them is the same: observability.

What Is LLM Observability?

LLM observability is complete visibility into every layer of an LLM-based software system: the application, the prompt, and the response.

LLM Observability vs ML Observability

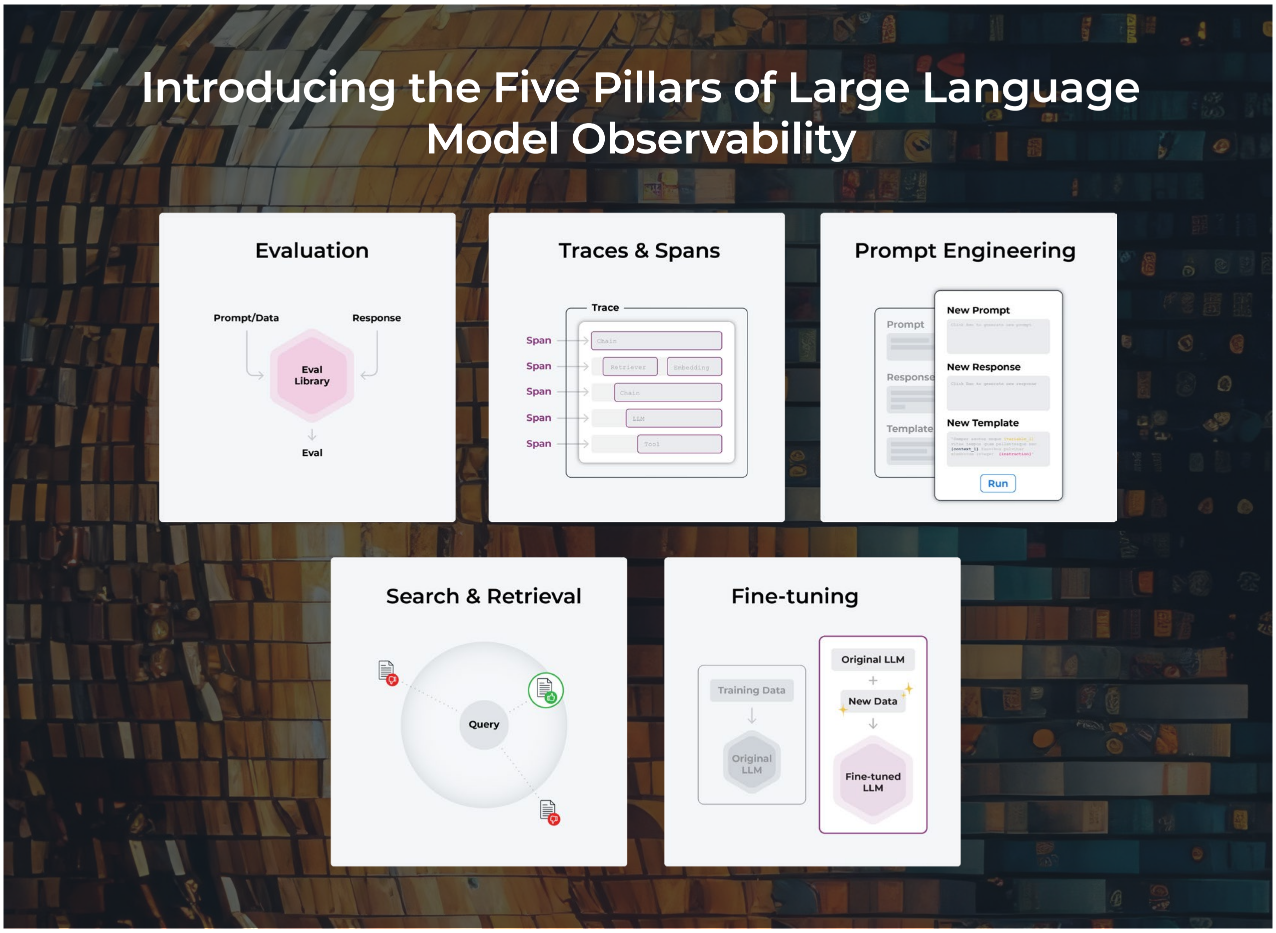
While large language models are a newer entrant to the ML landscape, they have a lot in common with older ML systems, so observability operates similarly in both cases.

- Just as in ML observability, **embeddings** are extremely useful to understanding unstructured data, and embedding techniques similar to those we have discussed in the past are even more important in LLMs.
- **Understanding model performance and tracking it over time** is still as important as ever. Data drift and model drift are important to understand.
- **Data collection** (history of prompts/responses) is still very important with LLMs for understanding drift and fine-tuning models.

But there are some important differences too:

- The model of LLM deployment is very different from more traditional ML. In the vast majority of cases, you have **much less visibility into the model internals** because you are likely using a third-party provider for your LLM.
- Evaluations are fundamentally different since you are **evaluating generation and not ranking.** This necessitates a whole new set of tools, some of which are only made possible by LLMs to begin with.
- If you are using **vector stores** (and most retrieval cases do), there are unique challenges that may prevent optimal retrieval and thus produce less-than-ideal prompts.

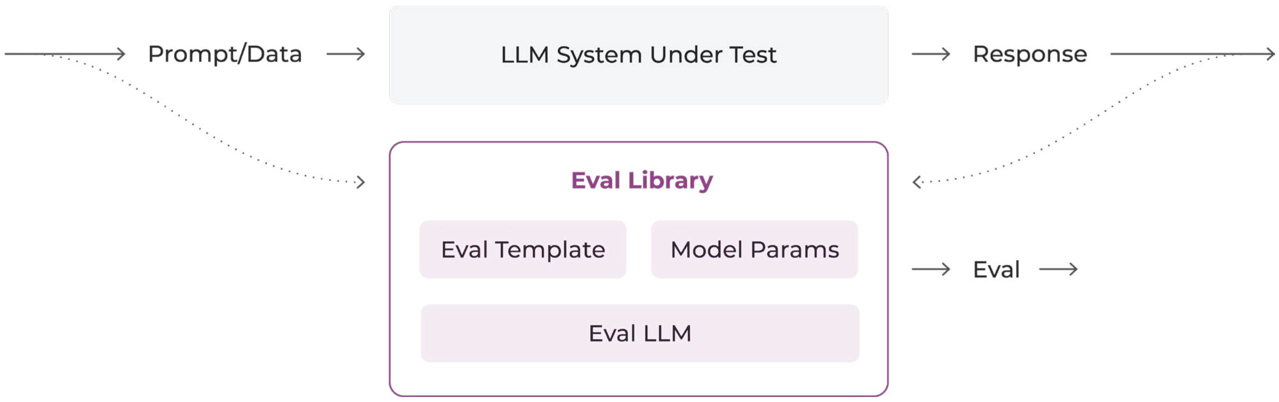
Agentic workflows and orchestration frameworks like LlamaIndex and LangChain present their own challenges that require different observability approaches.



Let's understand the five pillars of LLM observability in order of importance. Each merits its own research, but here is a quick summary.

LLM Evals

Evaluation is a measure of how well the response answers the prompt. This is the most important pillar on LLM observability.



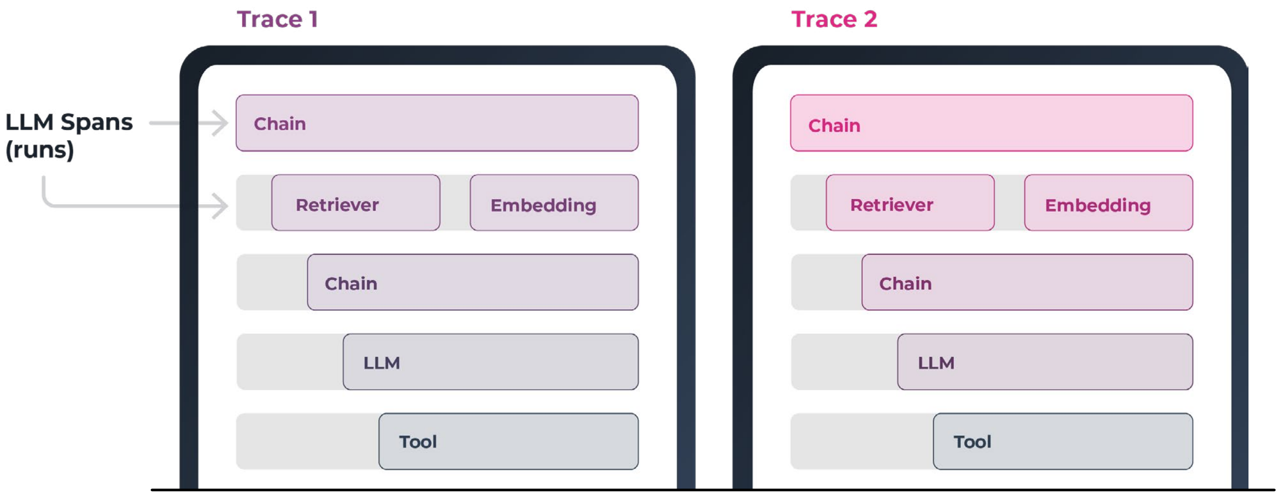
You can have an evaluation for individual queries, but we typically start by trying to find patterns. For example, if you look at embedding visualization for prompts and separate good responses from bad responses, patterns will begin to emerge.

1. How do you know if the response was good in the first place? There are several ways to evaluate. You can collect the feedback directly from your users. This is the simplest way but can often suffer from users not being willing to provide feedback or simply forgetting to do so. Other challenges arise from implementing this at scale.
2. The other approach is to use an LLM to evaluate the quality of the response for a particular prompt. This is more scalable and very useful but comes with typical LLM setbacks.

Once you have identified the problem region(s), you can summarize the prompts that are giving your system trouble. You can do this by reading the prompts and finding similarities or by using another LLM.

The point of this step is simply to identify that there is a problem and give you your first clues on how to proceed. To make this process simpler, you can use Arize's open-sourced [Phoenix framework.]

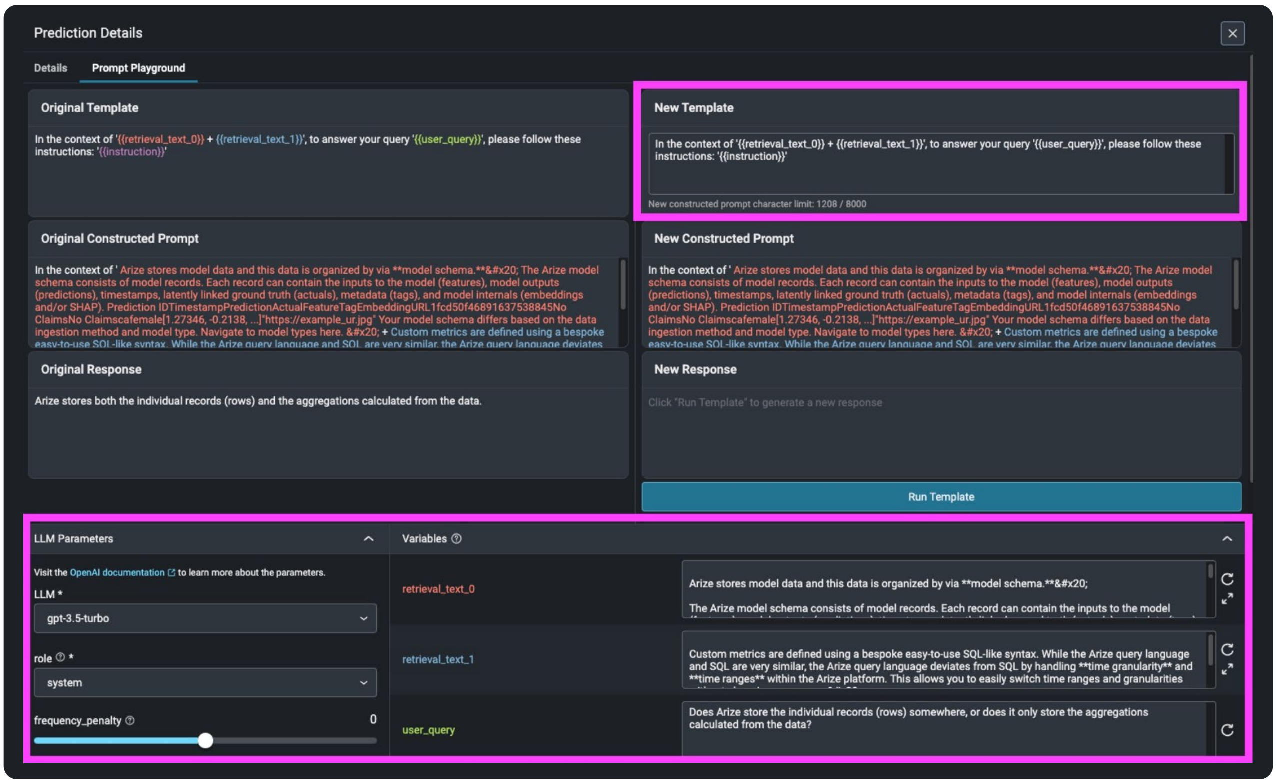
Traces and Spans In Agentic Workflows



For more complex or agentic workflows, it may not be obvious which call in a span or which span in your trace (a run through your entire use case) is causing the problem. You may need to repeat the evaluation process on several spans before you narrow down the problem.

This pillar is largely about diving deep into the system to isolate the issue you are investigating. This may involve retrieval steps or LLM use case steps.

Prompt Engineering


Prediction Details

Details | Prompt Playground

Original Template

In the context of '{{retrieval_text_0}} + {{retrieval_text_1}}', to answer your query '{{user_query}}', please follow these instructions: '{{instruction}}'

New Template

In the context of '{{retrieval_text_0}} + {{retrieval_text_1}}', to answer your query '{{user_query}}', please follow these instructions: '{{instruction}}'

New constructed prompt character limit: 1208 / 8000

Original Constructed Prompt

In the context of ' Arize stores model data and this data is organized by via **model schema.**&#x20; The Arize model schema consists of model records. Each record can contain the inputs to the model (features), model outputs (predictions), timestamps, latently linked ground truth (actuals), metadata (tags), and model internals (embeddings and/or SHAP). Prediction IDTimestampPredictionActualFeatureTagEmbeddingURL1fcd50f46891637538845No ClaimsNo Claimscafemale[1.27346, -0.2138, ...]"https://example_ur.jpg" Your model schema differs based on the data ingestion method and model type. Navigate to model types here. &#x20; + Custom metrics are defined using a bespoke easy-to-use SQL-like syntax. While the Arize query language and SQL are very similar, the Arize query language deviates

New Constructed Prompt

In the context of ' Arize stores model data and this data is organized by via **model schema.**&#x20; The Arize model schema consists of model records. Each record can contain the inputs to the model (features), model outputs (predictions), timestamps, latently linked ground truth (actuals), metadata (tags), and model internals (embeddings and/or SHAP). Prediction IDTimestampPredictionActualFeatureTagEmbeddingURL1fcd50f46891637538845No ClaimsNo Claimscafemale[1.27346, -0.2138, ...]"https://example_ur.jpg" Your model schema differs based on the data ingestion method and model type. Navigate to model types here. &#x20; + Custom metrics are defined using a bespoke easy-to-use SQL-like syntax. While the Arize query language and SQL are very similar, the Arize query language deviates

Original Response

Arize stores both the individual records (rows) and the aggregations calculated from the data.

New Response

Click "Run Template" to generate a new response

Run Template

LLM Parameters

Visit the OpenAI documentation to learn more about the parameters.

LLM *: gpt-3.5-turbo

role *: system

frequency_penalty: 0

Variables

retrieval_text_0: Arize stores model data and this data is organized by via **model schema.**&#x20;

The Arize model schema consists of model records. Each record can contain the inputs to the model

retrieval_text_1: Custom metrics are defined using a bespoke easy-to-use SQL-like syntax. While the Arize query language and SQL are very similar, the Arize query language deviates from SQL by handling **time granularity** and **time ranges** within the Arize platform. This allows you to easily switch time ranges and granularities

user_query: Does Arize store the individual records (rows) somewhere, or does it only store the aggregations calculated from the data?

Example of Iterating and Comparing Responses Across Prompt Templates in Arize

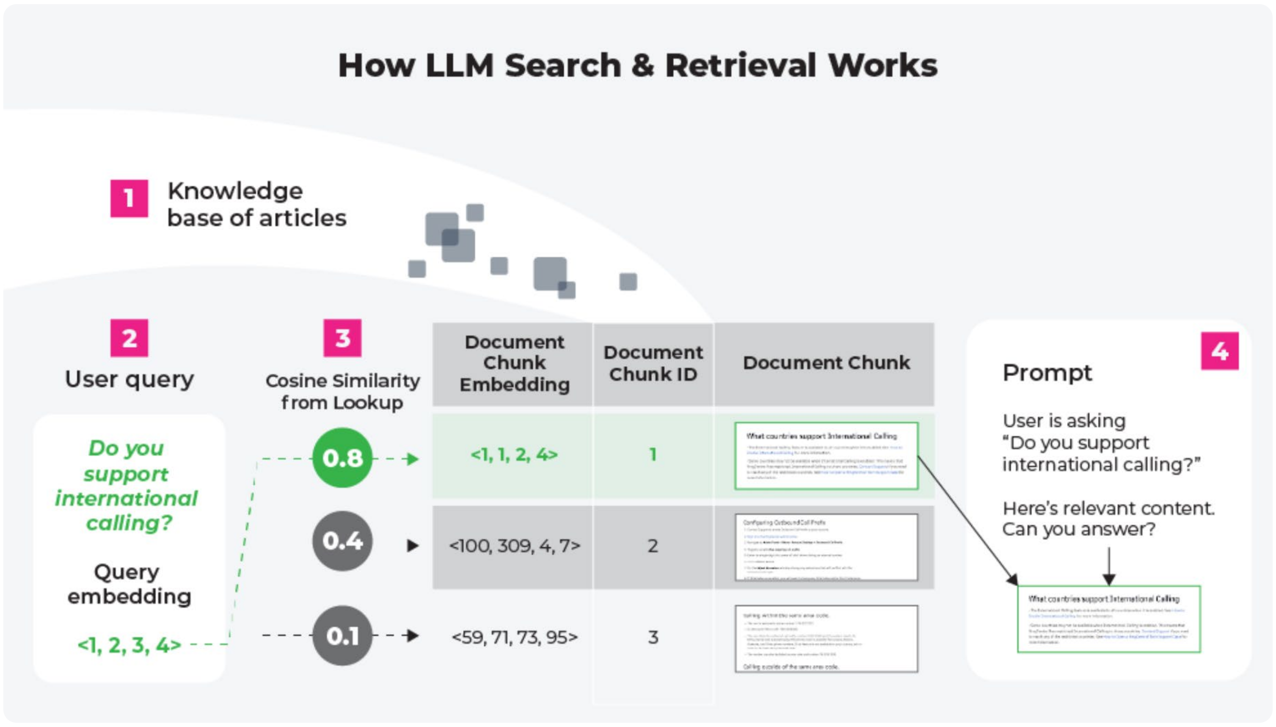
Prompt engineering is the cheapest, fastest, and often the highest-leverage way to improve the performance of your application.

This is similar to how humans think. You are going to have a pretty hard time answering the question "Who is that?" in isolation, but if you get a little more context – "Who is that on the moon, next to the Apollo 11 landing module?" – the number of reasonable answers is narrowed down right away.

LLMs are based on the attention mechanism, and the magic behind the attention mechanism is that it is really good at picking up relevant context. You just need to provide it.

The tricky part in product applications is that you not only have to get the right prompt, you also have to try to make it concise. LLMs are priced per token, and doubling the number of tokens in your prompt template in a scaled application can get really expensive. The other issue is that LLMs have a limited context window, so there is only so much context you can provide in your prompt.

Search and Retrieval


How LLM Search & Retrieval Works
1 Knowledge base of articles
2 User query
Do you support international calling?
Query embedding
<1, 2, 3, 4>
3 Cosine Similarity from Lookup
0.8
0.4
0.1
Document Chunk Embedding
<1, 1, 2, 4>
<100, 309, 4, 7>
<59, 71, 73, 95>
Document Chunk ID
1
2
3
Document Chunk
What countries support International Calling
Configuring Outbound Call Prefix
Calling within the same area code
Calling outside of the same area code
4 Prompt
User is asking "Do you support international calling?"
Here's relevant content. Can you answer?
What countries support International Calling

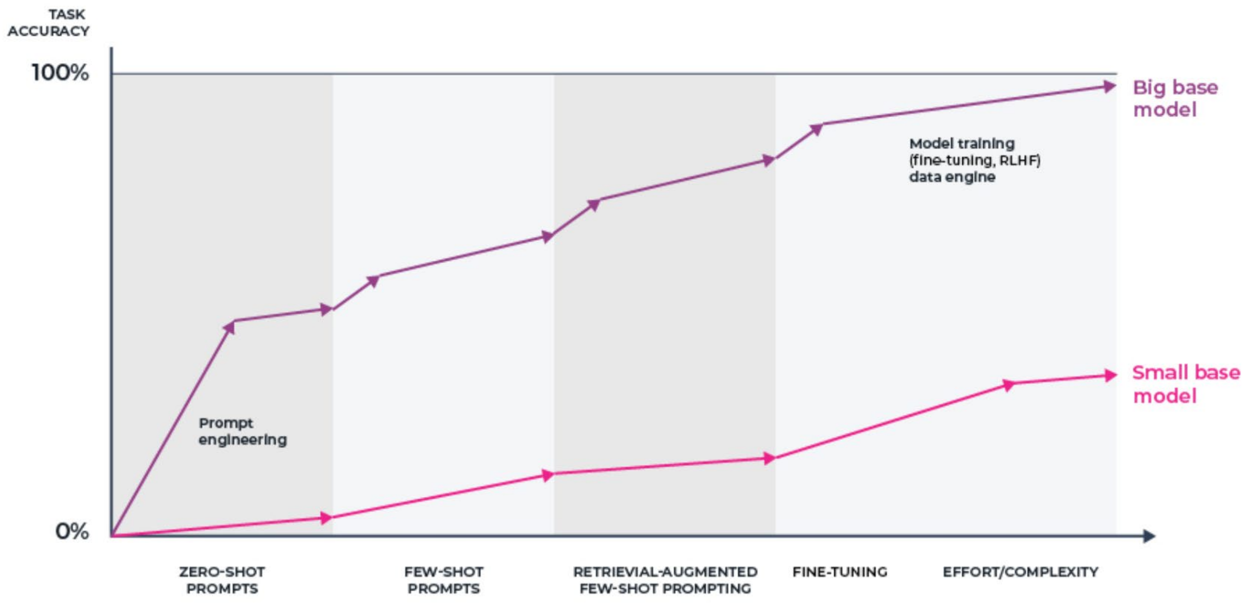
The other way to improve performance is with more relevant information being fed in. This is a bit harder and more expensive but can yield incredible results.

If you can retrieve more relevant information, your prompt improves automatically. The information retrieval systems, however, are more complex. Perhaps you need a better embedding? Perhaps you can make retrieval a multi-step process?

To do this, you may first embed the documents by summaries. Then at retrieval time, find the document by summary first, then get relevant chunks. Or you can embed text at the sentence level, then expand that window during LLM synthesis. Or maybe you can just embed the reference to the text or even change how you organize information on the back end. There are many possibilities to decouple embeddings from the text chunks.

Fine Tuning

Finally, you can fine tune your model. This essentially generates a new model that is more aligned with your exact usage conditions. Fine tuning is expensive, difficult, and may need to be done again as the underlying LLM or other conditions of your system change. This is a very powerful technique, but you should be very clear about your ROI before embarking on this adventure.



Source: Andrej Karpathy

How To Set Up An Application for Observability

There is no universal way to set up your application. There are many different architectures and patterns, and setting up your application is largely dependent on the particulars, but here are a few pointers.

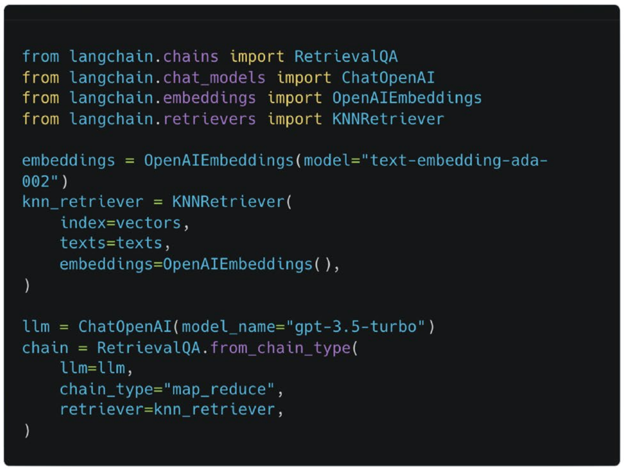
- **Human feedback:** If you are lucky enough to collect user feedback, you should store your feedback and responses for future analysis. Otherwise you can generate LLM-assisted evals and store those instead.
- **Multiple prompt templates:** If you have multiple prompt templates, compare between them. Otherwise iterate on your prompt template to see if you can improve performance.
- **Using retrieval augmented generation (RAG):** If you can access the knowledge base, evaluate if there are gaps in it. Otherwise use prod logs to evaluate if the retrieved content is relevant.
- **Chains and agents:** Log spans and traces to see where the app breaks. For each span, use the suggestions from above on human feedback.
- **Fine tuning:** If you fine tune, find and export example data that can be used for fine tuning.

LLM Evaluation

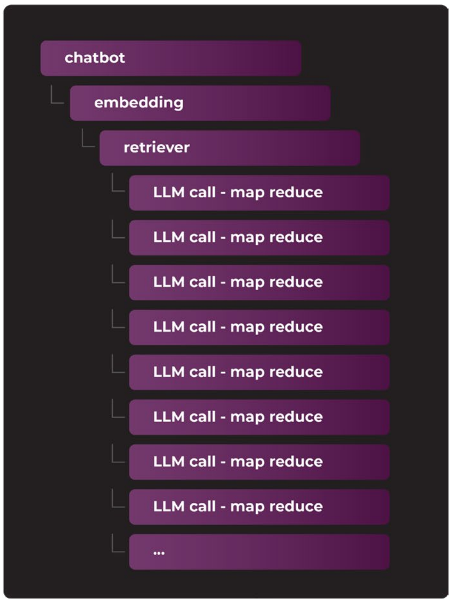
Why Is LLM Evaluation Needed?

As the applications multiply, so does the importance of measuring the performance of LLM-based applications. This is a nontrivial problem for several reasons: user feedback or any other **“source of truth” is extremely limited** and often nonexistent; even when possible, **human labeling is still expensive**; and it is **easy to make these applications complex.**

4-5 Lines of Code ⟶ 10+ LLM Calls



```
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI
from langchain.embeddings import OpenAIEmbeddings
from langchain.retrievers import KNNRetriever

embeddings = OpenAIEmbeddings(model="text-embedding-ada-
002")
knn_retriever = KNNRetriever(
    index=vectors,
    texts=texts,
    embeddings=OpenAIEmbeddings(),
)

llm = ChatOpenAI(model_name="gpt-3.5-turbo")
chain = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="map_reduce",
    retriever=knn_retriever,
)
```



Fortunately, we can use the power of LLMs to automate the evaluation. In this section, we will delve into how to set this up and make sure it is reliable.

The core of LLM evals is AI evaluating AI.

While this may sound circular, we have always had human intelligence evaluate human intelligence (for example, at a job interview or your college finals). Now AI systems can finally do the same for other AI systems.

The process here is for LLMs to generate synthetic ground truth that can be used to evaluate another system. Which begs a question: why not use human feedback directly? Put simply, because you will never have enough of it.

Getting human feedback on even one percent of your input/output pairs is a gigantic feat. Most teams don’t even get that. But in order for this process to be truly useful, it is important to have evals on every LLM sub-call, of which we have already seen there can be many.

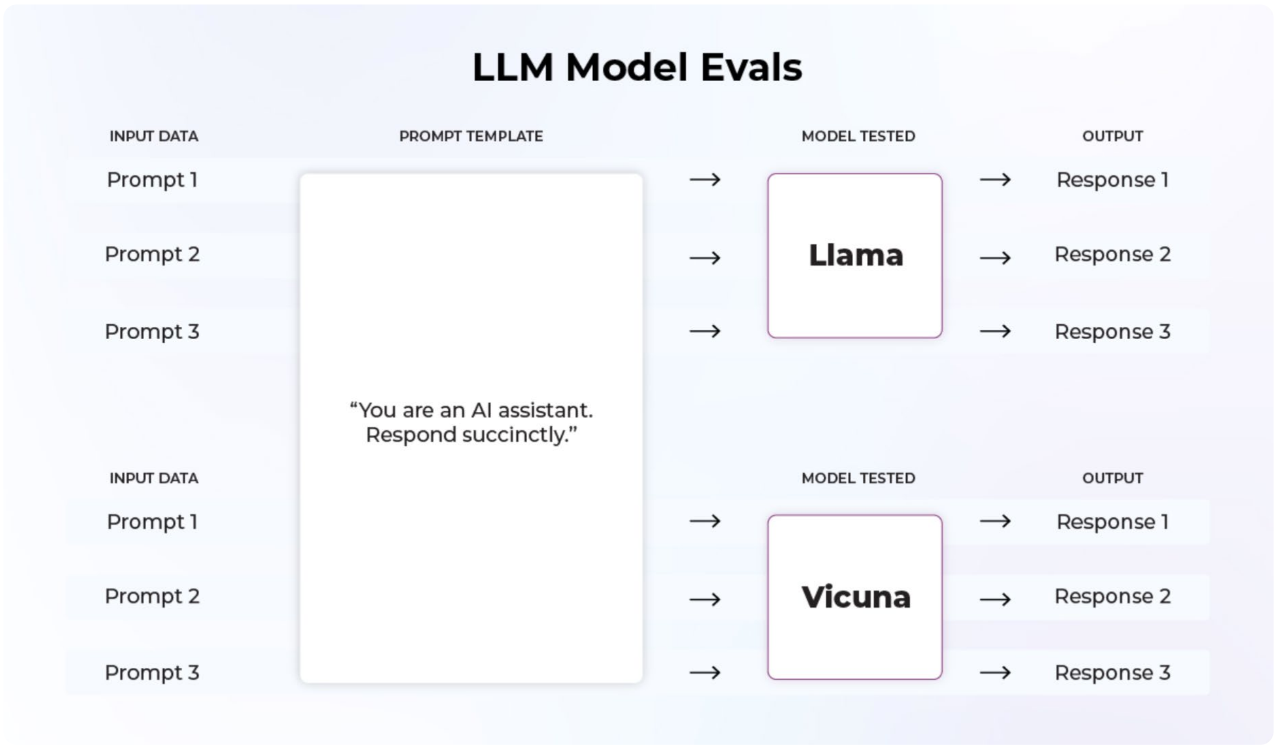
Let’s explore how to do this.

LLM Model Evals versus LLM System Evals

```
LLM_model_evals != LLM_System_evals
```

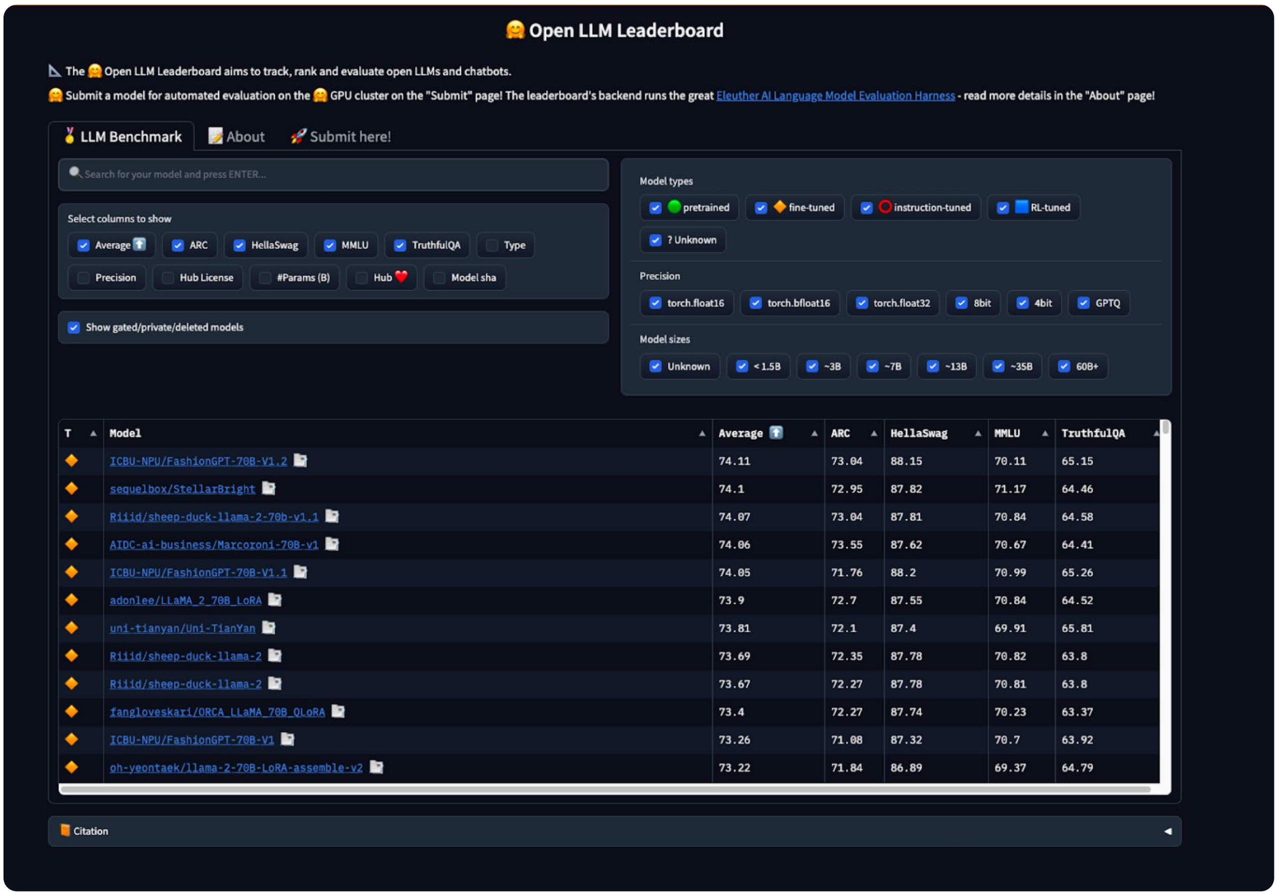
LLM Model Evals

You might have heard of LLM evals. This term gets used in many different ways that all sound very similar but actually are very different. One of the more common ways it gets used is in what we will call LLM model evals. LLM model evals are focused on the overall performance of the foundational models. The companies launching the original customer-facing LLMs needed a way to quantify their effectiveness across an array of different tasks.



In this case, we are evaluating two different open source foundation models. We are testing the same dataset across the two models and seeing how their metrics, like hellaswag or mmlu, stack up.

One popular library that has LLM model evals is the OpenAI Eval library, which was originally focused on the model evaluation use case. There are many metrics out there, like HellaSwag (which evaluates how well an LLM can complete a sentence), TruthfulQA (measuring truthfulness of model responses), and MMLU (which measures how well the LLM can multitask). There's even a leaderboard that looks at how well the open-source LLMs stack up against each other.



🤗 Open LLM Leaderboard

📐 The 🤗 Open LLM Leaderboard aims to track, rank and evaluate open LLMs and chatbots.

🤗 Submit a model for automated evaluation on the 🤗 GPU cluster on the "Submit" page! The leaderboard's backend runs the great Eleuther AI Language Model Evaluation Harness - read more details in the "About" page!

🏅 LLM Benchmark | 📝 About | 🚀 Submit here!

Search for your model and press ENTER...

Select columns to show: ✓ Average ⬆️ · ✓ ARC · ✓ HellaSwag · ✓ MMLU · ✓ TruthfulQA · ☐ Type · ☐ Precision · ☐ Hub License · ☐ #Params (B) · ☐ Hub ❤️ · ☐ Model sha

✓ Show gated/private/deleted models

Model types: ✓ 🟢 pretrained · ✓ 🔶 fine-tuned · ✓ ⭕ instruction-tuned · ✓ 🟦 RL-tuned · ✓ ? Unknown

Precision: ✓ torch.float16 · ✓ torch.bfloat16 · ✓ torch.float32 · ✓ 8bit · ✓ 4bit · ✓ GPTQ

Model sizes: ✓ Unknown · ✓ < 1.5B · ✓ ~3B · ✓ ~7B · ✓ ~13B · ✓ ~35B · ✓ 60B+

T	Model	Average ⬆️	ARC	HellaSwag	MMLU	TruthfulQA
🔶	ICBU-NPU/FashionGPT-70B-V1.2	74.11	73.04	88.15	70.11	65.15
🔶	sequelbox/StellarBright	74.1	72.95	87.82	71.17	64.46
🔶	Riiid/sheep-duck-llama-2-70b-v1.1	74.07	73.04	87.81	70.84	64.58
🔶	AIDC-ai-business/Marcoroni-70B-v1	74.06	73.55	87.62	70.67	64.41
🔶	ICBU-NPU/FashionGPT-70B-V1.1	74.05	71.76	88.2	70.99	65.26
🔶	adonlee/LLaMA_2_70B_LoRA	73.9	72.7	87.55	70.84	64.52
🔶	uni-tianyan/Uni-TianYan	73.81	72.1	87.4	69.91	65.81
🔶	Riiid/sheep-duck-llama-2	73.69	72.35	87.78	70.82	63.8
🔶	Riiid/sheep-duck-llama-2	73.67	72.27	87.78	70.81	63.8
🔶	fangloveskari/ORCA_LLaMA_70B_QLoRA	73.4	72.27	87.74	70.23	63.37
🔶	ICBU-NPU/FashionGPT-70B-V1	73.26	71.08	87.32	70.7	63.92
🔶	oh-yeontaek/llama-2-70B-LoRA-assemble-v2	73.22	71.84	86.89	69.37	64.79

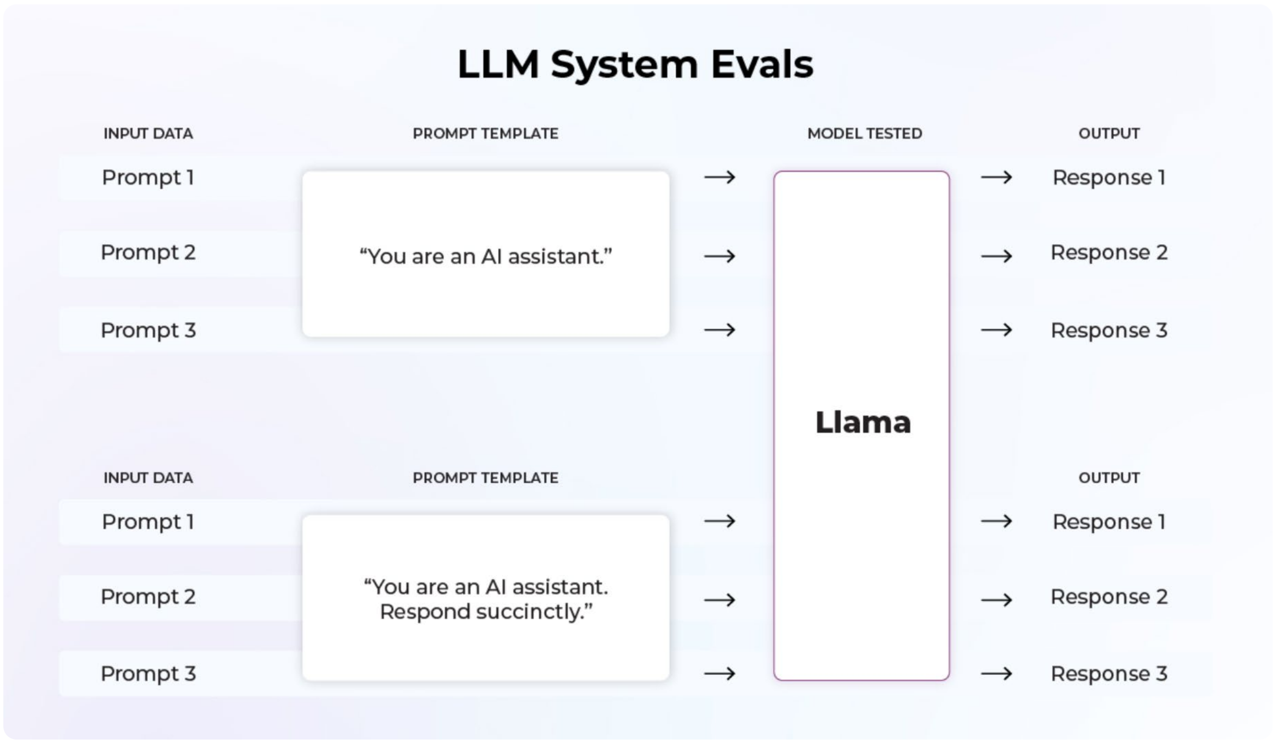
📙 Citation

Hugging Face OpenLLM Leaderboard

LLM System Evals

Up to this point, we have discussed LLM model evaluation. In contrast, **LLM system evaluation** is the complete evaluation of components that you have control of in your system. The most important of these components are the prompt (or prompt template) and context. LLM system evals assess how well your inputs can determine your outputs.

LLM system evals may, for example, hold the LLM constant and change the prompt template. Since prompts are more dynamic parts of your system, this evaluation makes a lot of sense throughout the lifetime of the project. For example, an LLM can evaluate your chatbot responses for usefulness or politeness, and the same eval can give you information about performance changes over time in production.



In this case, we are evaluating two different prompt templates on a single foundational model. We are testing the same dataset across the two templates and seeing how their metrics like precision and recall stack up.

When To Use LLM System Evals versus LLM Model Evals: It Depends On Your Role

There are distinct personas who make use of LLM evals. One is the model developer or an engineer tasked with fine-tuning the core LLM, and the other is the practitioner assembling the user-facing system.

There are very few LLM model developers, and they tend to work for places like OpenAI, Anthropic, Google, Meta, and elsewhere. **Model developers care about LLM model evals**, as their job is to deliver a model that caters to a wide variety of use cases.

For ML practitioners, the task also starts with model evaluation. One of the first steps in developing an LLM system is picking a model (i.e. GPT 3.5 vs 4 vs Palm, etc.). The LLM model eval for this group, however, is often a one-time step. Once the question of which model performs best in your use case is settled, the majority of the rest of the application's lifecycle will be defined by LLM system evals. Thus, **ML practitioners care about both LLM model evals and LLM system evals but likely spend much more time on the latter.**

LLM System Evaluation Metrics Vary By Use Case

Having worked with other ML systems, your first question is likely this: “What should the outcome metric be?” The answer depends on what you are trying to evaluate.

- **Extracting structured information:** You can look at how well the LLM extracts information. For example, you can look at completeness (is there information in the input that is not in the output?).
- **Question answering:** How well does the system answer the user’s question? You can look at the accuracy, politeness, or brevity of the answer—or all of the above.
- **Retrieval Augmented Generation (RAG):** Are the retrieved documents and final answer relevant?

As a system designer, you are ultimately responsible for system performance, and so it is up to you to understand which aspects of the system need to be evaluated. For example, If you have an LLM interacting with children, like a tutoring app, you would want to make sure that the responses are age-appropriate and are not toxic.

The most common evaluations we see being employed today are relevance, hallucinations, question-answering accuracy, and toxicity. Each one of these evals will have different templates based on what you are trying to evaluate.

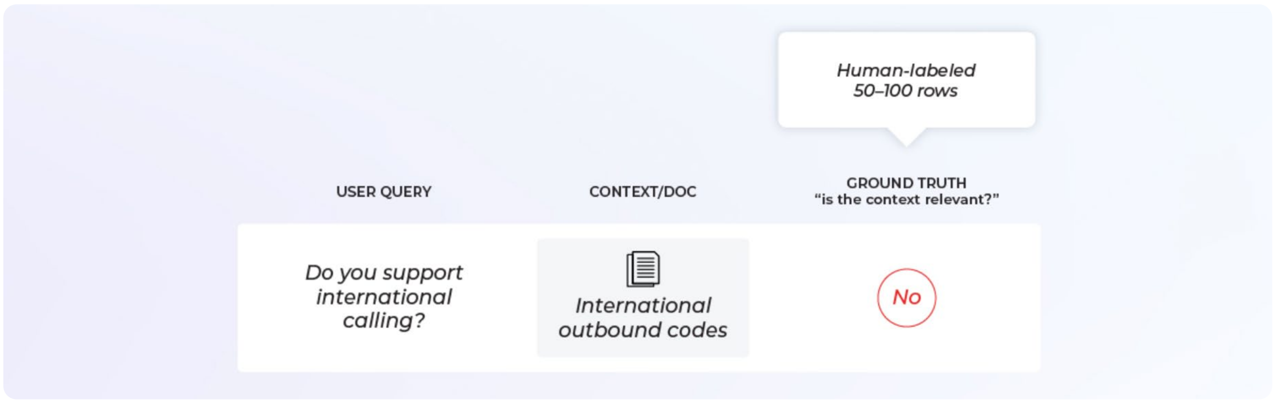
How To Get AI To Evaluate AI

There are two distinct steps to the process of evaluating your LLM-based system with an LLM. First, establish a benchmark for your LLM evaluation metric. To do this, you put together a dedicated LLM-based eval whose only task is to label data as effectively as a human labeled your “golden dataset.” You then benchmark your metric against that eval. Then, run this LLM evaluation metric against results of your LLM application (more on this below).

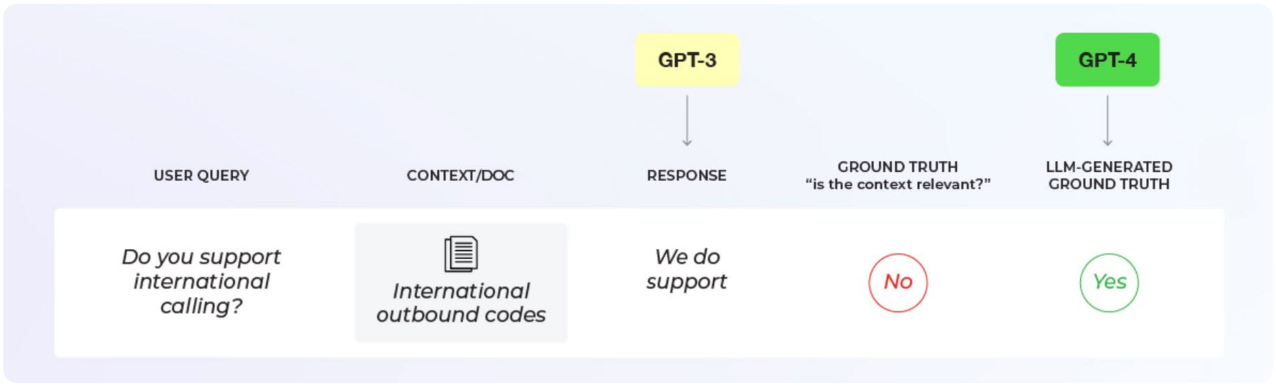
How To Build An LLM Eval

The first step, as we covered above, is to build a benchmark for your evaluations.

To do that, you must begin with a **metric best suited for your use case**. Then, you need the **golden dataset**. This should be representative of the type of data you expect the LLM eval to see. The golden dataset should have the “ground truth” label so that we can measure performance of the LLM eval template. Often such labels come from human feedback. Building such a dataset is laborious, but you can often find a standardized one for the most common use cases.


Human-labeled
50–100 rows
USER QUERY
CONTEXT/DOC
GROUND TRUTH
"is the context relevant?"
Do you support international calling?
International outbound codes
No

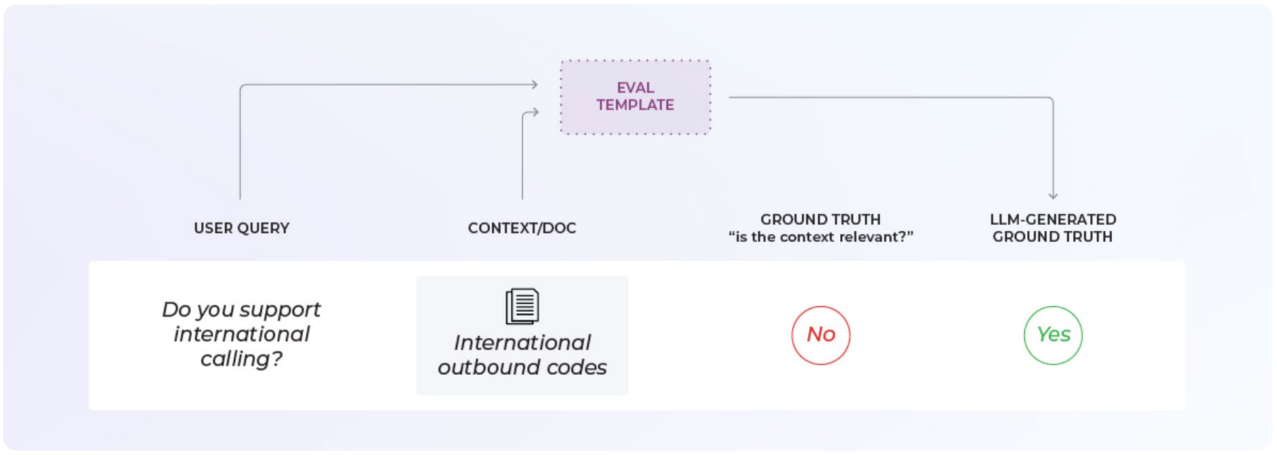
Then you need to decide **which LLM** you want to use for evaluation. This could be a different LLM from the one you are using for your application. For example, you may be using Llama for your application and GPT-4 for your eval. Often this choice is influenced by questions of cost and accuracy.



Now comes the core component that we are trying to benchmark and improve: the **eval template**. If you're using an existing library like OpenAI or Phoenix, you should start with an existing template and see how that prompt performs.

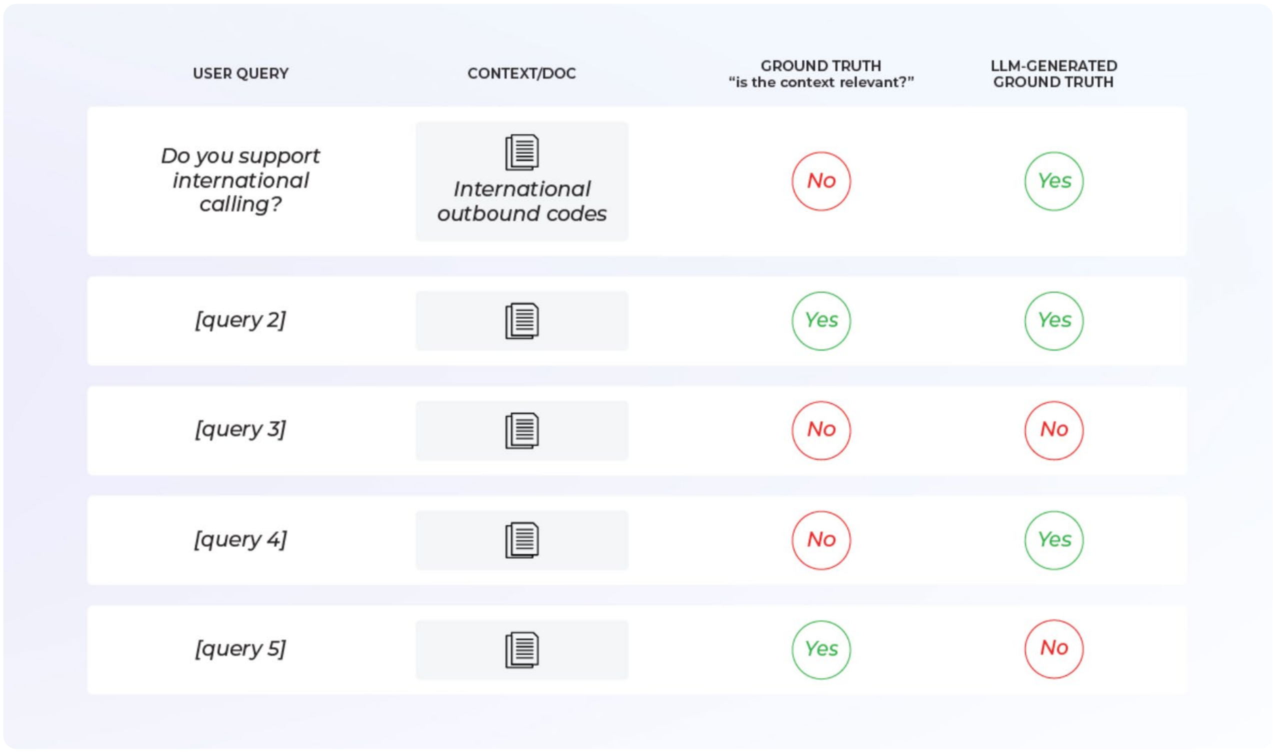
If there is a specific nuance you want to incorporate, adjust the template accordingly or build your own from scratch. Keep in mind that the template should have a clear structure. Be explicit about the following:

1. **What is the input?** In our example, it is the documents/context that was retrieved and the query from the user.
2. **What are we asking?** In our example, we're asking the LLM to tell us if the document was relevant to the query
3. **What are the possible output formats?** In our example, it is binary relevant/irrelevant, but it can also be multi-class (e.g., fully relevant, partially relevant, not relevant).


EVAL TEMPLATE
USER QUERY
CONTEXT/DOC
GROUND TRUTH "is the context relevant?"
LLM-GENERATED GROUND TRUTH
Do you support international calling?
International outbound codes
No
Yes

You now need to run the eval across your golden dataset. Then you can **generate metrics** (overall accuracy, precision, recall, F1-score, etc.) to determine the benchmark. It is important to look at more than just overall accuracy. We'll discuss that below in more detail.

If you are not satisfied with the performance of your LLM evaluation template, you need to change the prompt to make it perform better. This is an iterative process informed by hard metrics. As is always the case, it is important to avoid overfitting the template to the golden dataset. Make sure to have a representative holdout set or run a k-fold cross-validation.



USER QUERY	CONTEXT/DOC	GROUND TRUTH "is the context relevant?"	LLM-GENERATED GROUND TRUTH
Do you support international calling?	*International outbound codes*	No	Yes
[query 2]		Yes	Yes
[query 3]		No	No
[query 4]		No	Yes
[query 5]		Yes	No

Finally, you arrive at your benchmark.
The optimized performance on the golden dataset represents how confident you can be on your LLM eval. It will not be as accurate as your ground truth, but it will be accurate enough, and it will cost much less than having a human labeler in the loop on every example.

	precision	recall
relevant	0.70	0.70
irrelevant	0.89	0.89

Why Is It Important To Use Precision and Recall When Benchmarking An LLM Prompt Template?

The industry has not fully standardized best practices on LLM evals. Teams commonly do not know how to establish the right benchmark metrics.

Overall accuracy is used often, but it is not enough.

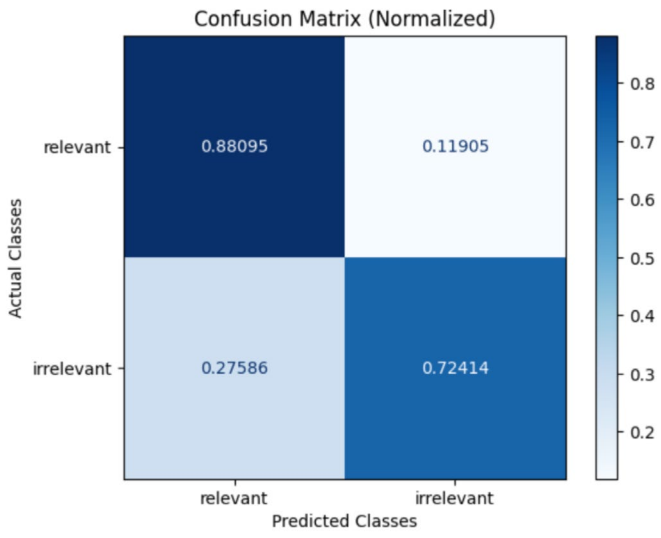
This is one of the most common problems in data science in action: very significant class imbalance makes accuracy an impractical metric.

Thinking about it in terms of the relevance metric is helpful. Say you go through all the trouble and expense of putting together the most relevant chatbot you can. You pick an LLM and a template that are right for the use case. This should mean that significantly more of your examples should be evaluated as “relevant.” Let’s pick an extreme number to illustrate the point: 99.99% of all queries return relevant results. Hooray!

Now look at it from the point of view of the LLM eval template. If the output was “relevant” in all cases, without even looking at the data, it would be right 99.99% of the time. But it would simultaneously miss all of the (arguably most) important cases — ones where the model returns irrelevant results, which are the very ones we must catch.

In this example, accuracy would be high, but precision and recall (or a combination of the two, like the F1 score) would be very low. Precision and recall are a better measure of your model’s performance here.

The other useful visualization is the confusion matrix, which basically lets you see correctly and incorrectly predicted percentages of relevant and irrelevant examples.



In this example, we see that the highest percentage of predictions are correct: a relevant example in the golden dataset has an 88% chance of being labeled as such by our eval. However, we see that the eval performs significantly worse on “irrelevant” examples, mislabeling them more than 27% of the time.

How To Run LLM Evals On Your Application

At this point you should have both your LLM application and your tested LLM eval. You have proven to yourself that the eval works and have a quantifiable understanding of its performance against a golden dataset.

Now we can actually use our eval on our application. This will help us measure how well our LLM application is doing and figure out how to improve it.

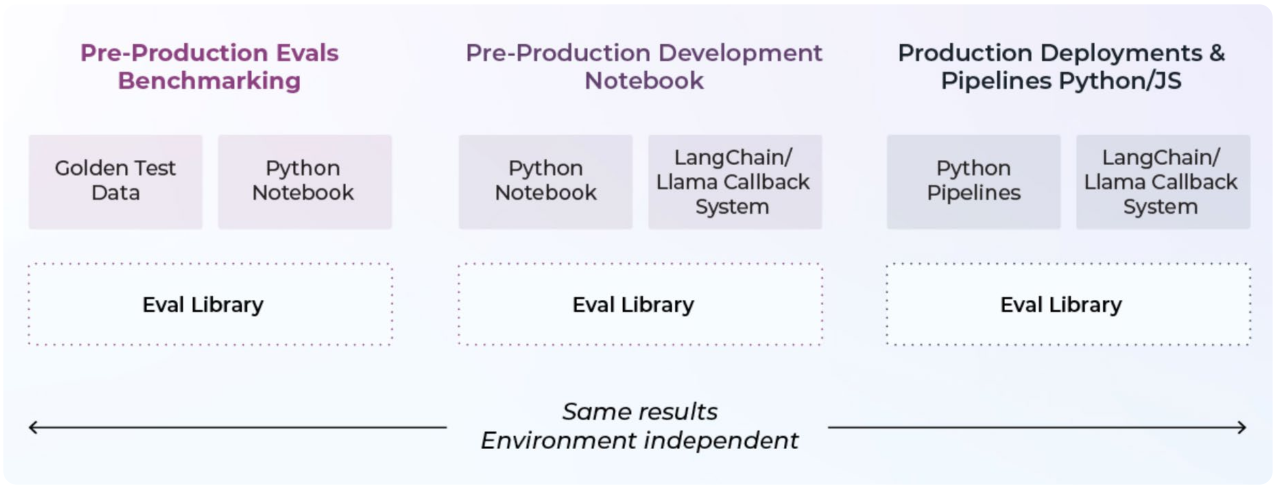
The LLM system eval runs your entire system with one extra step. For example:

1. You retrieve your input docs and add them to your prompt template, together with sample user input.
2. You provide that prompt to the LLM and receive the answer.
3. You provide the prompt and the answer to your eval, asking it if the answer is relevant to the prompt.

It is a best practice not to do LLM evals with one-off code but rather a library that has built-in prompt templates. This increases reproducibility and allows for more flexible evaluation where you can swap out different pieces.

These evals need to work in three different environments:

1. Pre-production when you're doing the benchmarking.
2. Pre-production when you're testing your application. This is somewhat similar to the offline evaluation concept in traditional ML. The idea is to understand the performance of your system before you ship it to customers.
3. Production when it's deployed. Life is messy. Data drifts, users drift, models drift, all in unpredictable ways. Just because your system worked well once doesn't mean it will do so on Tuesday at 7 p.m. Evals help you continuously understand your system's performance after deployment.


Pre-Production Evals Benchmarking
Golden Test Data
Python Notebook
Eval Library
Pre-Production Development Notebook
Python Notebook
LangChain/ Llama Callback System
Eval Library
Production Deployments & Pipelines Python/JS
Python Pipelines
LangChain/ Llama Callback System
Eval Library
Same results
Environment independent

Questions To Consider

How many rows should you sample?

The LLM-evaluating-LLM paradigm is not magic. You cannot evaluate every example you have ever run across—that would be prohibitively expensive. However, you already have to sample data during human labeling, and having more automation only makes this easier and cheaper. So you can sample more rows than you would with human labeling.

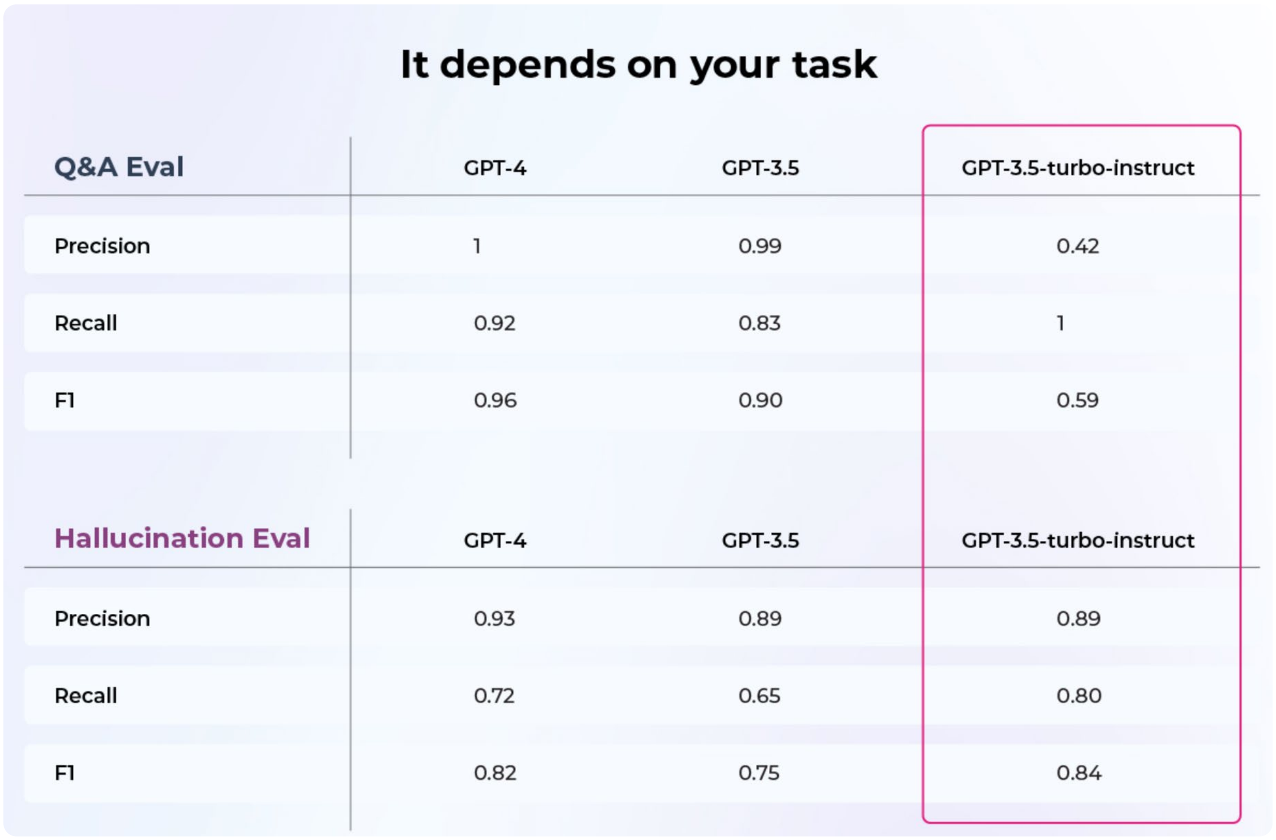
Which evals should you use?

This depends largely on your use case. For search and retrieval, relevancy-type evals work best. Toxicity and hallucinations have specific eval patterns.

Some of these evals are important in the troubleshooting flow. Question-answering accuracy might be a good overall metric, but if you dig into why this metric is underperforming in your system, you may discover it is because of bad retrieval, for example. There are often many possible reasons, and you might need multiple metrics to get to the bottom of it.

What model should I use?

It is impossible to say that one model works best for all cases. Instead, you should run model evaluations to understand which model is right for your application. You may also need to consider tradeoffs of recall vs. precision, depending on what makes sense for your application. In other words, do some data science to understand this for your particular case.



It depends on your task

Q&A Eval	GPT-4	GPT-3.5	GPT-3.5-turbo-instruct
Precision	1	0.99	0.42
Recall	0.92	0.83	1
F1	0.96	0.90	0.59

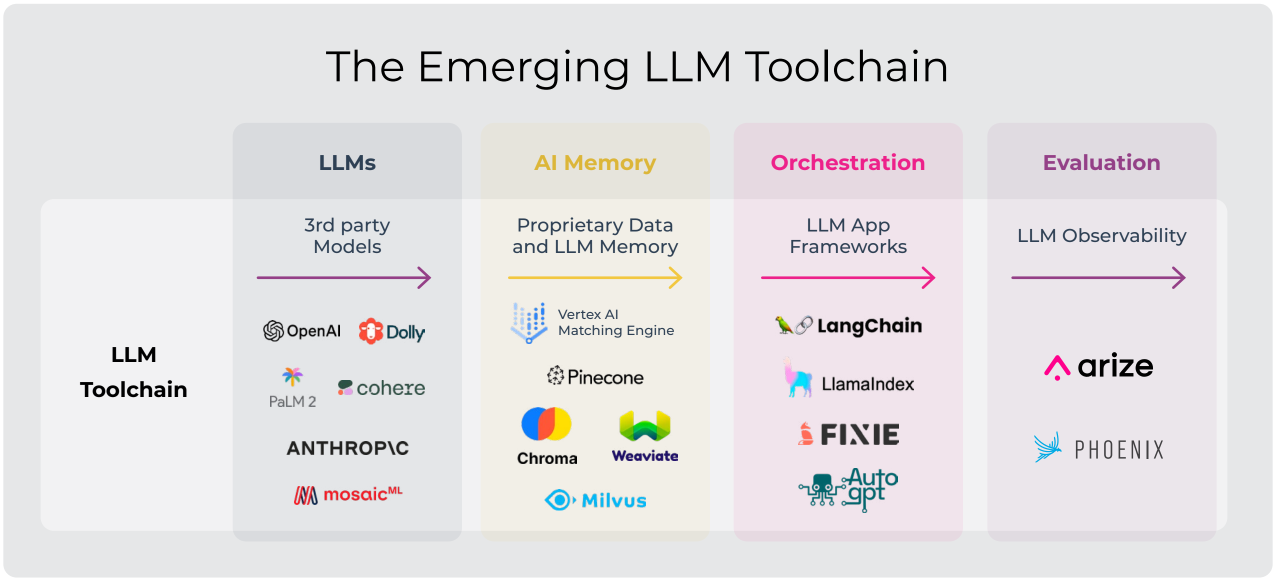
Hallucination Eval	GPT-4	GPT-3.5	GPT-3.5-turbo-instruct
Precision	0.93	0.89	0.89
Recall	0.72	0.65	0.80
F1	0.82	0.75	0.84

Which model to use often depends on your task

In summary, being able to evaluate the performance of your application is very important when it comes to production code. In the era of LLMs, the problems have gotten harder, but luckily we can use the very technology of LLMs to help us in running evaluations. Such evaluation should test the whole system and not just the underlying LLM model—think about how much a prompt template matters to user experience. Best practices, standardized tooling, and curated datasets simplify the job of developing LLM systems.

Traces and Spans

LLM orchestration frameworks like LlamaIndex, Microsoft's Semantic Kernel and LangChain offer flexible data frameworks to connect your own private data to LLMs in order to leverage the wave of the latest generative AI advancements. In the emerging LLM toolchain (see below), these frameworks are in the center of the LLMOps system between various LLM application tools.



LLM orchestration frameworks are trying to enable developers with the necessary tools to build LLM applications and LLM observability is designed to manage and maintain these applications in production. This orchestration process includes many different components including: programmatic querying, retrieving contextual data from a vector database, and maintaining memory across LLM and API calls. Whether you are using a callback system from a programming framework for LLMs (like LangChain or LlamaIndex) or creating a bespoke system, LLM applications require observability to make sure each component is performing optimally in production.

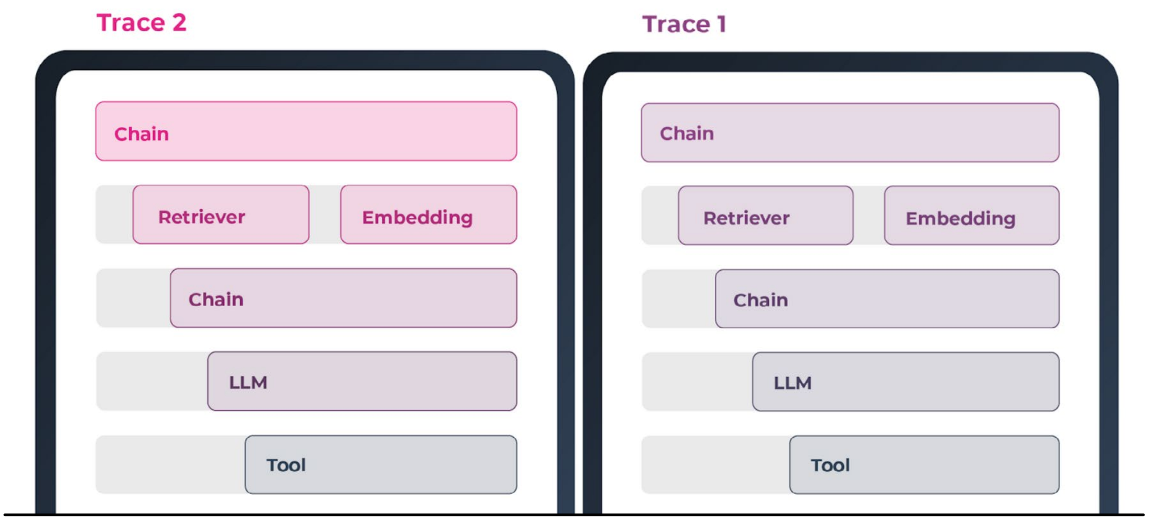
Tracing LLM Applications: Who Should Be Tackling This Within AI Teams?

Currently there are a variety of industries trying to implement LLMs into production, but who within an AI organization is best equipped for this challenge? LLMOps Engineer is not currently a title we see being advertised, but LLM-powered systems are being built for a variety of use cases across industries, so who are the LLM developers? Today this is overwhelmingly the skilled software engineers who understand highly scalable and low-latency systems.

Traditionally data scientist, AI researcher, and machine learning engineer have been the most common roles for those developing AI algorithms and ML evaluation tools. However, we have seen a recent shift that is putting the power of AI in the hands of developers with strong software capabilities. With advanced open source libraries and models, developers have been able to integrate software together to create new AI applications without the need of data science and ML understanding. Data engineers, software engineers and other developers will have an advantage in LLM observability with their familiarity not only in the terminology, but also troubleshooting API callbacks, when compared to data scientists and ML engineers. However, since the fields of MLOps and LLMOps are still being created, tested and iterated upon, the time is right for anyone interested in generative AI to start learning about and implementing LLM systems now!

For software engineers that work with distributed systems, terms like “spans,” “traces,” and “calls” are well known. While there are parallels between the worlds of network observability, distributed systems, and LLM observability there are also nuances with generative AI terms. Since LLM observability isn’t just about tracking API calls, but about evaluating the LLM’s performance on specific tasks, there are a variety of span kinds and attributes that can be filtered on, in order to troubleshoot a LLMs’ performance.

Before walking through these workflows in detail, let’s give the scenario that you are a Software Engineer at an e-commerce company which recently pushed an LLM-powered chatbot into production. Your chatbot, which is used to interact with customers who have purchased from your company’s website, uses a search and retrieval system to create responses for the customer (see Figure 1 below).



Note that the callback sequence you will be looking at depends on how your LLM orchestration layer is organized, whether you are using a LLM framework (like LlamaIndex and LangChain) or you create a bespoke framework. For the following callback tracing workflows, let’s use the LLM orchestration framework provided by LlamaIndex that accepts any input prompt over your data and returns a knowledge-augmented response. Please refer to Table 1 below for related terminology in the following workflows.

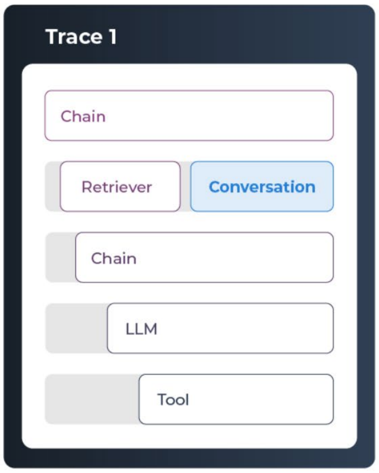
Definition of LLM Observability Terms for Reference

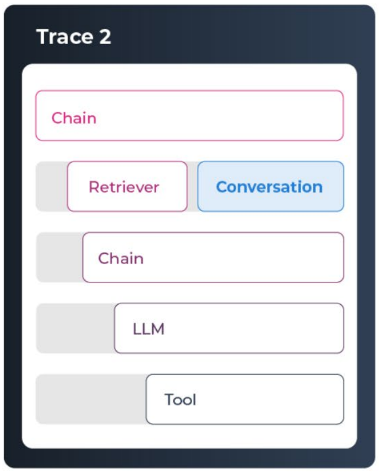
Term	LLM Observability Definition*
Traces	Traces represent a single invocation of an LLM application. For example, when a chain is run or a query engine is queried, that is a trace. Another way to think of traces is as a sequence of spans tied together by a trace ID
Spans	Spans are units of execution that have inputs and outputs that a builder of an LLM application may care to evaluate. There are different kinds of spans, including chain spans, LLM spans, and embedding spans, that are differentiated by various kinds of attributes. For example LLM span type: Attributes = Temperature, Provider, Max Tokens, ...
Tools	Tool as the defining feature of an agent, an arbitrary function (e.g., a calculator, a piece of code to make an API call, a tool to query a SQL database) that an LLM can choose to execute or not based on the input from a user or the state of an application.
Parent-Child Relationships Between Spans	Every trace has a hierarchical structure. The top span, which is the entry point, has no parent. However, as you delve deeper into the system's operations, you'll find child spans that are initiated by their parent spans.
Conversations	Conversations are a series of traces and spans tied together by a Conversation ID. These occur across traces without any parallel operations and contain a single back and forth conversation between the LLM and a given user.

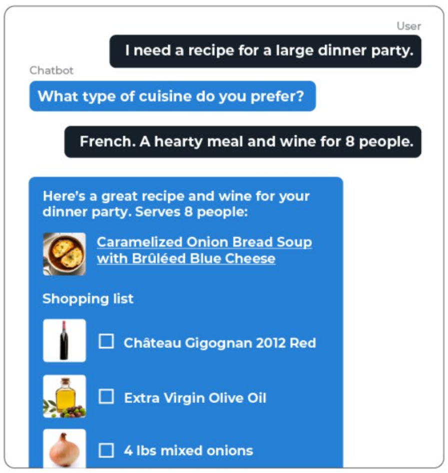
***Note:** While traces and spans are a familiar concept in intra/network observability, they differ in many ways for LLM observability. For example, the concepts of evals, agents, embeddings and LLMs as span types is nowhere to be found in the infra world. In APM (application performance monitoring) a transaction trace gives a detailed snapshot of a single transaction in your application, this is similar to a run in a LLM application system. While you will see similar timing information and similar annotations on spans (marking them with key-value pair attributes to use for visualization) the semantic conventions that are being established for the types of spans, what attributes should be present on these types, and the appropriate evals for them are all new for LLM tracing.

When you execute a LLM run, the process of interacting with your selected LLM is documented in a callback system by a trace. In this trace a span can refer to any unit of execution, you may annotate a span with a specific name (agent, LLM, tool, embedding) or a general term like a chain (which can refer to any process that doesn't have its own span kind).

Now let's go through two troubleshooting workflows that you can use to break down each call you are making to your LLM – these will be the top-down and bottom-up LLM workflows.







I need a recipe for a large dinner party.

What type of cuisine do you prefer?

French. A hearty meal and wine for 8 people.

Here's a great recipe and wine for your dinner party. Serves 8 people:

Caramelized Onion Bread Soup with Brûléed Blue Cheese

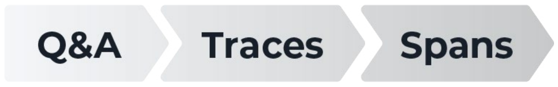
Shopping list

☐ Château Gigognan 2012 Red

☐ Extra Virgin Olive Oil

☐ 4 lbs mixed onions

Option 1: Top-Down LLM Workflows

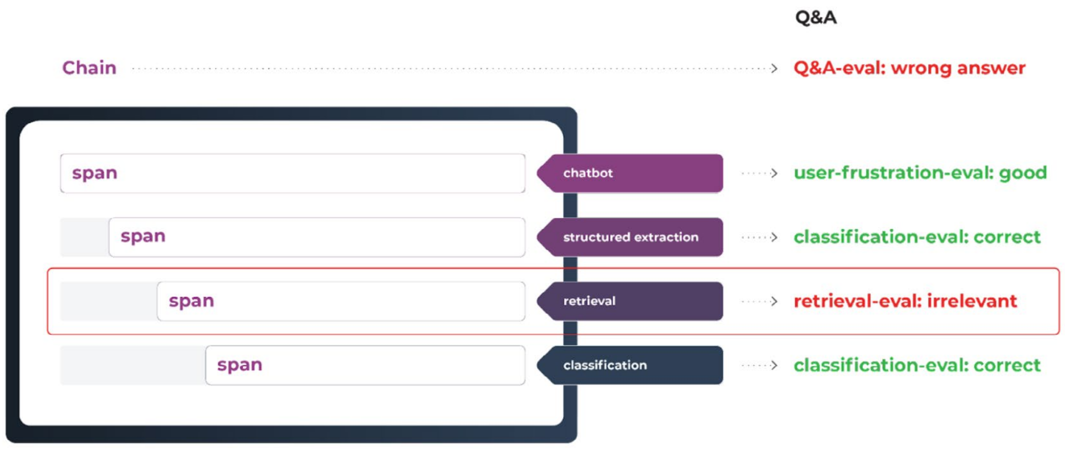


A top-down approach can be thought of starting with the big picture of the LLM use case and then getting into specifics of the execution if the performance is not satisfactory. For example with your e-commerce LLM powered chatbot into production. Your chatbot is interacting with customers in a series of back and forth conversations. Some customers might have only one question one time, while other customers may have a series of conversations that stretch over months. Either way, every time a question is asked, your LLM system is queried to produce a response for your user and each run produces an individual trace (with a series of spans).

In order to troubleshoot performance top down you can take all unsatisfactory conversations (labeled as such due to user feedback or a separate evaluation system), then rank them from worst to best performance, and filter down into the individual traces, and then spans to see where the major problem is hidden.

For example, let's say on Wednesday you get an alert that your overall chatbot performance is operating at 70% satisfactory (which is lower than the accepted 80%)

and this chatbot was operating at 85% last week. Clearly, there are some unhappy conversations. Now, to troubleshoot these underperforming conversations you first filter down on the worst performing conversations and then filter on the traces and span types that are underperforming to see if there is an issue in the execution chain. As it turns out, there is an irrelevant output from a span type (see below) on every one of these underperforming conversations, most likely due to missing relevant content. You remembered your team launched a website update on Tuesday and saw that the outputs were unsatisfactory because they were using information about the old website. This issue can be ultimately resolved by adding context to your LLM for how to add items to cart via the updated website.

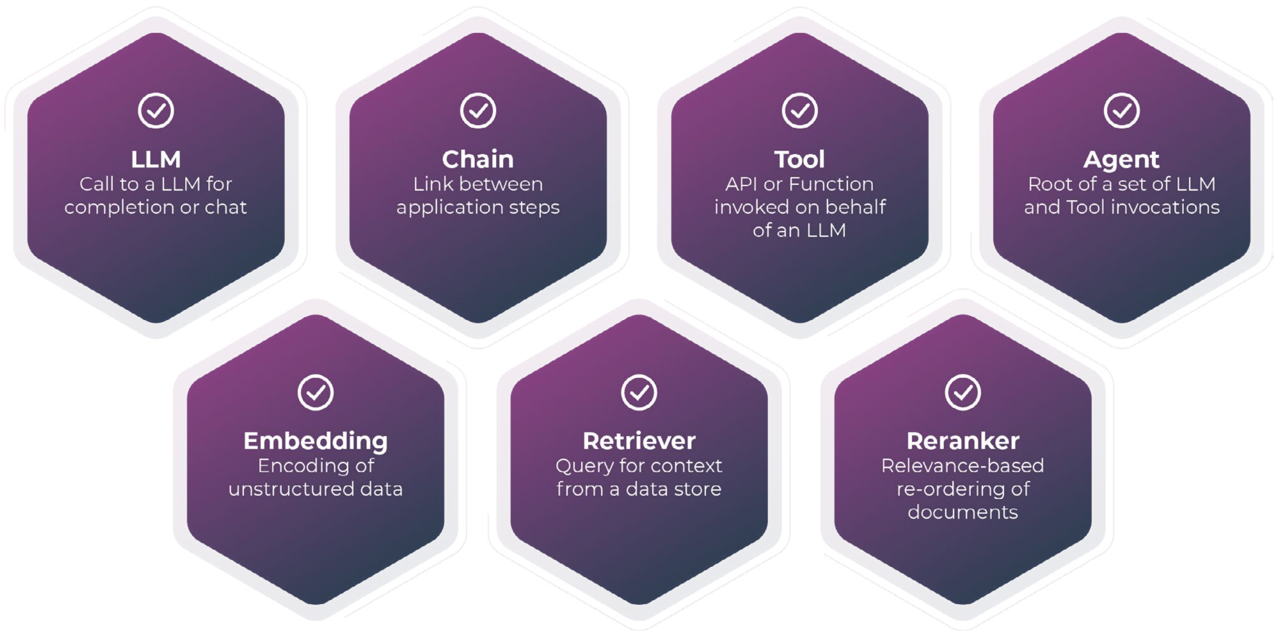

Q&A
Chain
Q&A-eval: wrong answer
span
chatbot
user-frustration-eval: good
span
structured extraction
classification-eval: correct
span
retrieval
retrieval-eval: irrelevant
span
classification
classification-eval: correct

Option 2: Bottom-Up LLM Workflows

Spans
Traces
Q&A

The bottom-up workflow can also be thought of as a discovery workflow where you are at the local level to filter on individual spans. In the same way you might think about metadata or tags, you can filter by specific spans to troubleshoot performance in your LLM use-cases. For example you can filter on embedding or LLM spans to see performance, latency and token account appear as expected. So if you are curious on how your outputs, prompts and performances are for your agents, LLMs, vector databases and more, use a bottom-up approach to evaluate individual components of your LLM system. This workflow can also link back any underperforming span type to their corresponding traces and conversations.

LLM Span Types and Their Functions


LLM
Call to a LLM for completion or chat
Chain
Link between application steps
Tool
API or Function invoked on behalf of an LLM
Agent
Root of a set of LLM and Tool invocations
Embedding
Encoding of unstructured data
Retriever
Query for context from a data store
Reranker
Relevance-based re-ordering of documents

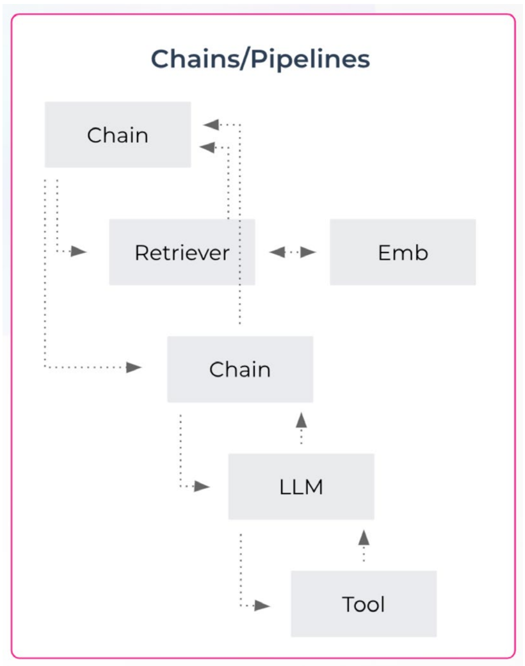
Types of spans and the role they play in a LLM callback system.

Let's explore a specific trace for a search and retrieval use case that will look and evaluate the chain, LLM, tool, and retriever spans individually within a run.

Chain

This is the most general kind of span, it has an input and an output and chains together all the calls. A chain can query, synthesize and provide templating. For example: If I ask your company's chatbot "Can I copy a dashboard?" then the chain (query) might:

```
INPUT: Can I copy a dashboard

OUTPUT: Yes, you can copy a dashboard
```



Note that the output is achieved after the embedding and retrieve span, as seen from the **trace details.**

Embedding

For embedding queries/questions — like embeddings that correspond to relevant documents from a vector store, for example.

```
{
 "embedding": {
             "model_name": "text-embedding-ada-002",
             "embeddings": [
              {
               "embedding.vector": [
               -0.018075328320254816
               -0.0026873357128351 92
               -0.010201605968177319
             ],
             "embedding.text": "Can I copy a dashboard?"
              }
             ]
            },
"__computed__": {
 "latency_ms": 187.385
}
```

Retriever

Responsible for retrieving relevant data. For example: retrieving all the relevant documents that were selected using their corresponding embeddings from the vector store.

```
"input": {
   "value": "Can I copy a dashboard?"
},
"retrieval": {
   "documents": [
     {
         "document.id": "873ef3a3-6938-4734-b10b-d29be152579"
         "document.score": "0.8101961612701416"
         "document.content": "\nTemplates are designed as starting points
for dashboard and model analysis. Once a dashboard is created from a
template, it can be edited and customized as desired. \n\n!\n\n"
```

LLM

Represents the LLM's operations. Remember, we are using a search and retrieval workflow for our chatbot as seen from the embedding and retrieval steps. Now that the relevant documents have been surfaced the LLM is called to synthesize the information from the documents to produce the correct answer for the input query.

Attributes of the LLM span like prompts and LLM model version (`"model_name": "gpt-3.5-turbo"`) are defined, the LLM will use that information to generate an output from the provided background documentation.

For example:

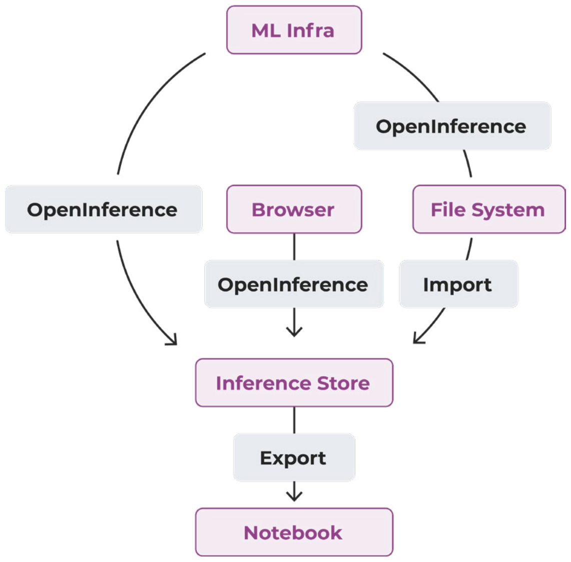
```
"output": {
     "value": "assistant: Yes, you can copy a
 dashboard."
   },
   "__computed__": {
     "latency_ms": 1127.944,
     "cumulative_token_count_total": 224,
 }
```

Note that in addition to the provided output, key performance metrics like token count and latency were also returned.

Additional Filtering and Alerting

Now that we have seen workflows for exploring conversations, traces and spans, let's talk about the methods of troubleshooting these workflows with filtering and monitors. If you are familiar with ML observability, you would likely think about setting monitors with an upper bound on latency and token count of your LLM system (so you are able to surface abnormally high token counts for each trace, and poor latency times for traces and spans). This is a good starting point, but only the tip of the iceberg in terms of how deep you can start to understand your LLM system.

Remember you can even create your own spans and a bespoke or modified LLM orchestration layer. You can add span kinds that correspond to specific LLMs, conversations, vector databases, and evaluations. At which point you can filter on customer IDs, conversation ID, performance metrics, user feedback and more.



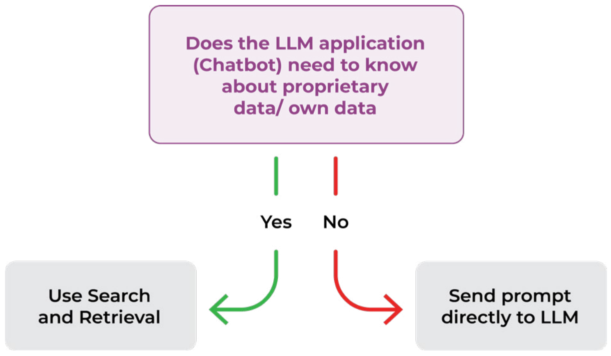
Inputs/Outputs and Attributes

If you are interested in testing out for yourself what attributes span types have, and why are attributes useful, try out our OpenInference standard that the Arize team is currently working on.

Search and Retrieval

LLMs and Proprietary Data: When Do You Need Search and Retrieval?

While LLMs can provide a broad base of knowledge, they are fundamentally limited by the information they have been trained on. An LLM won't know the specifics of your product documentation or understand the unique workings of your product. If there are updates or changes to your product, an LLM won't automatically have that information unless it's included in its training data.


Does the LLM application (Chatbot) need to know about proprietary data/ own data
Yes
No
Use Search and Retrieval
Send prompt directly to LLM

Development: How Search and Retrieval Works

Let's consider the common scenario of developing a customer support chatbot using an LLM. Usually, teams possess a wealth of product documentation, which includes a vast amount of unstructured data detailing their product, frequently asked questions, and use cases.

This data is broken down into pieces through a process called "chunking." How you chunk data matters, and in the next piece of our "Build Your Own Chatbot" course, we'll dig into chunking strategies and evaluation methods.

After the data is broken down, each chunk is assigned a unique identifier and embedded into a high-dimensional space within a vector database. This process leverages advanced natural language processing techniques to understand the context and semantic meaning of each chunk.

When a customer's question comes in, the LLM uses a retrieval algorithm to quickly identify and fetch the most relevant chunks from the vector database. This retrieval is based on the semantic similarity between the query and the chunks, not just keyword matching.

The picture above shows how search and retrieval is used with a prompt template (4) in order to generate a final LLM prompt context. The above view is the search and retrieval LLM use case in its simplest form: a document is broken into chunks, these chunks are embedded into a vector store, and the search and retrieval process pulls on this context to shape LLM output.

This approach offers a number of advantages. First, it significantly reduces the time and computational resources required for the LLM to process large amounts of data, as it only needs to interact with the relevant chunks instead of the entire documentation.

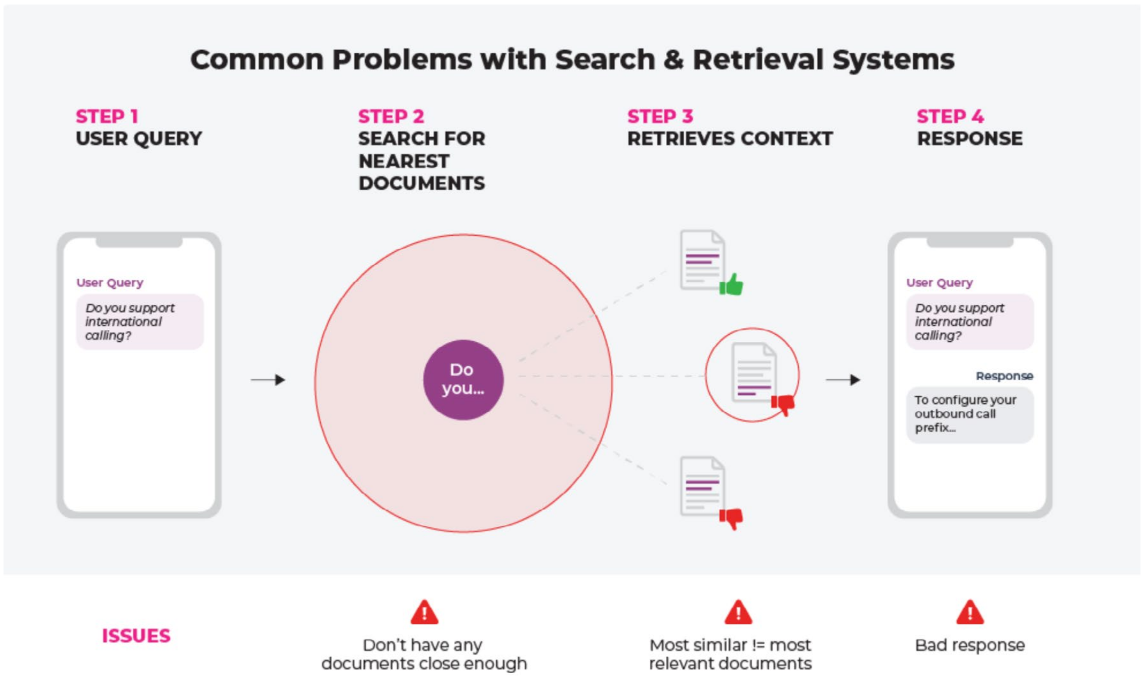
Second, it allows for real-time updates to the database. As product documentation evolves, the corresponding chunks in the vector database can be easily updated. This ensures that the chatbot always provides the most up-to-date information.

Finally, by focusing on semantically relevant chunks, the LLM can provide more precise and contextually appropriate responses, leading to improved customer satisfaction.

Production: Common Problems With Search and Retrieval Systems

While the search and retrieval method greatly enhances the efficiency and accuracy of LLMs, it's not without potential pitfalls. Identifying these issues early can prevent them from impacting user experience.

One such challenge arises when a user inputs a query that doesn't closely match any chunks in the vector store. The system looks for a needle in a haystack but finds no needle at all. This lack of match, often caused by unique or highly specific queries, can leave the system to draw on the "most similar" chunks available – ones that aren't entirely relevant.


Common Problems with Search & Retrieval Systems
STEP 1
USER QUERY
STEP 2
SEARCH FOR NEAREST DOCUMENTS
STEP 3
RETRIEVES CONTEXT
STEP 4
RESPONSE
User Query
Do you support international calling?
Do you...
User Query
Do you support international calling?
Response
To configure your outbound call prefix...
ISSUES
Don't have any documents close enough
Most similar != most relevant documents
Bad response

In turn, this leads to a subpar response from the LLM. Since the LLM depends on the relevance of the chunks to generate responses, the lack of an appropriate match could result in an output that's tangentially related or even completely unrelated to the user's query.

Irrelevant or subpar responses from the LLM can frustrate users, lowering their satisfaction and ultimately causing them to lose trust in the system and product as a whole. Monitoring three main things can help prevent these issues:

Query Density (Drift): Query density refers to how well user queries are covered by the vector store. If query density drifts significantly, it signals that our vector store may not be capturing the full breadth of user queries, resulting in a shortage of closely associated chunks. Regularly monitoring query density enables us to spot these gaps or shortcomings. With this insight, we can augment the vector store by incorporating more relevant chunks or refining the existing ones, improving the system's ability to fetch data in response to user queries.

Ranking Metrics: These metrics evaluate how well the search and retrieval system is performing in terms of selecting the most relevant chunks. If the ranking metrics indicate a decline in performance, it's a signal that the system's ability to distinguish between relevant and irrelevant chunks might need refinement.

User Feedback: Encouraging users to provide feedback on the quality and relevance of the LLM's responses helps gauge user satisfaction and identify areas for improvement. Regular analysis of this feedback can point out patterns and trends, which can then be used to adjust your application as necessary.

Refinement: How to Optimize and Improve Search and Retrieval

Optimization of search and retrieval processes should be a constant endeavor throughout the lifecycle of your LLM-powered application, from the building phase through to post-production.

During the building phase, attention should be given to developing a robust testing and evaluation strategy. This approach allows you to identify potential issues early on and optimize your strategies, forming a solid foundation for the system.

Key areas to focus on include:

- **Chunking Strategy:** Evaluating how information is broken down and processed during this stage can help highlight areas for improvement in performance.
- **Retrieval Performance:** Assessing how well the system retrieves information can indicate if you need to employ different tools or strategies, such as context ranking or HYDE.

Upon release, optimization efforts should continue as you enter the post-production phase. Even after launch, with a well-defined evaluation strategy, you can proactively identify any emerging issues and continue to improve your model's performance. Consider approaches like:

- **Expanding your Knowledge Base**: Adding documentation can significantly improve your system's response quality. An expanded data set allows your LLM to provide more accurate and tailored responses.
- **Refining Chunking Strategy:** Further modifying the way information is broken down and processed can lead to marked improvements.
- **Enhancing Context Understanding:** Incorporating an extra 'context evaluation' step helps the system incorporate the most relevant context into the LLM's response.

Specifics on these and other strategies for continuous optimization will be detailed in the following sections of this course. Remember, the goal is to create a system that not only meets users' needs at launch but also evolves with them over time.

Benchmarking Evaluation of LLM Retrieval Augmented Generation

Experiment Context & Overview of Findings

The primary source of data in our experiment is product documentation, which is chunked and seeded into a vector store. With a set of potential user questions ranging from general inquiries to specific questions about product features, the system produces outputs based on these queries which are in turn evaluated by the open source Phoenix LLM evals library. Key metrics include:

- *Precision of Context Retrieved:* How relevant and accurate is the information retrieved from the vector store when posed with a query?
- *Accuracy of the LLM Output:* Post-retrieval, how coherent and contextually accurate are the chatbot's responses?
- *System Latency:* Given that response time can significantly impact user experience, how long does the system take to provide output?

This section is an early attempt to add a rigorous testing layer to the latest LLM retrieval solutions. It includes both results and test scripts (notebook with these scripts to see an example of how they are run here) to parameterize retrieval on your own docs, determine performance with LLM evaluations and provide a repeatable framework to reproduce results.

The following takeaways are based on testing on the aforementioned product documentation (each is explored in greater depth below).

Chunk Sizes

Generally, chunk sizes of 300/500 tokens seem to a good target; going bigger has negative results.

Retrieval Algorithms

Retrieval algorithms have a latency and value tradeoff—if you have user interactivity requirements, you are likely better off sticking to the vanilla, simple approach. If accuracy is paramount and time does not matter, the fancier retrieval algorithms such as re-ranking or HyDE can markedly improve precision.

On K size

4–6 (or even lower) seems optimal trade off for performance and results. Given latency considerations, 4 might be the best bet.

Latency

Latency scales quickly with increasing K and retrieval method complexity.

Your Mileage May Vary

As always, it's crucial to conduct your own experiments to determine the best parameters for your specific use case.

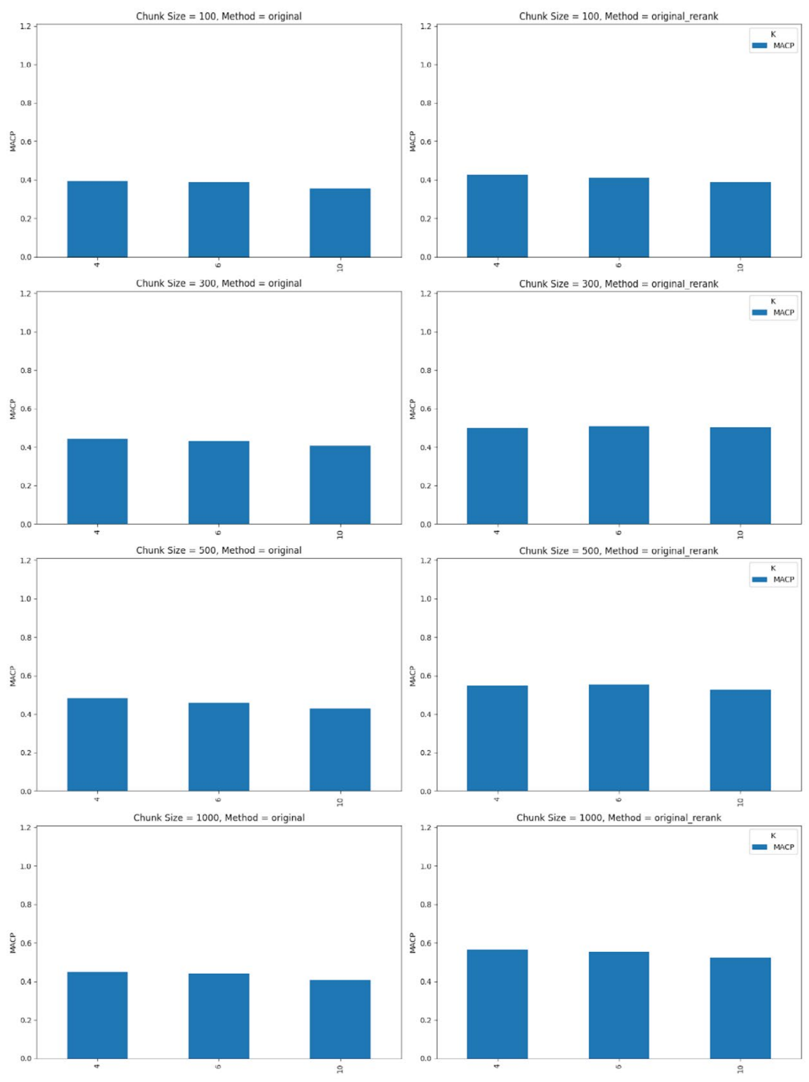
Parameters: Optimizing RAG Settings

What Is “K”

One of the most important parameters to consider is K, the number of returned chunks from the vector store. The vector store returns chunks of context, and those contexts can be returned a number at a time in order of similarity.

The above figure shows a sweep of K sizes 4, 5, and 6 across two retrieval approaches. For example, in the case K=6, six chunks of text are returned of that specific token size. It is expected that as you increase K for a fixed size set of relevant documents, that precision at K will drop. The above results do imply we are returning more relevant documents as we increase K.

Precision at k is the number of relevant items in the top k divided by k. If there is a fixed number of relevant items, and k increases while the number of relevant items in the top k remains constant or increases more slowly than k, then Precision at k will decrease.



Sweep of Chunk Sizes, Mean Average Precision at k (graphs by authors)

Mean Reciprocal Rank of Retrieval

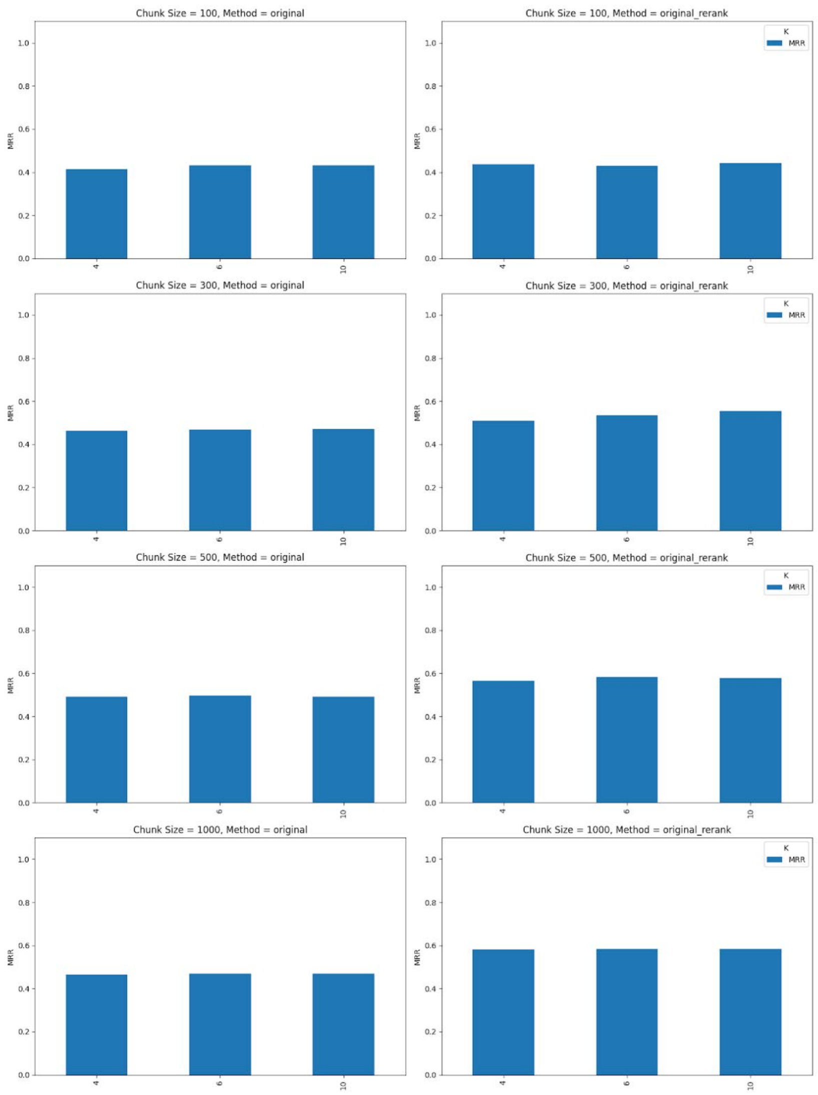
Another way we can look at retrieval results is look at mean reciprocal rank (MRR), included below:

The MRR mimics the retrieval results of the precision @k, about equal retrieval results with a slight edge to chunk = 500/1000 and k=4.

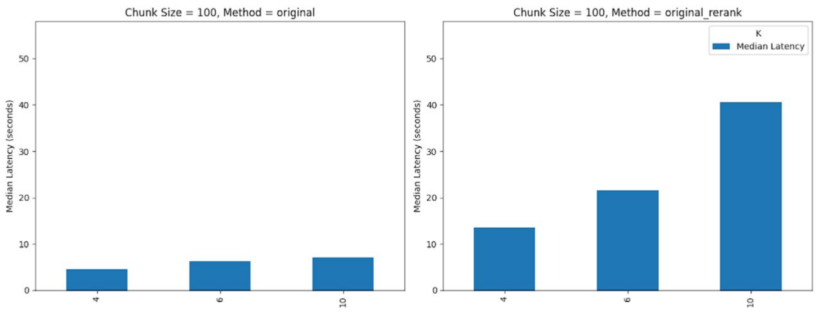
It's worth noting that our dataset has a number of cases built where there are questions the system can't answer. Zero retrieval in the dataset is expected for those questions. We have plots in the appendix removes those zero retrieval questions. It skews the numbers lower for all metrics but represents a real world scenario we see in practice.

At first glance the above results imply we should consider as high a K as possible—but we will find looking at other numbers, we will find the lower K of 4/6 a better choice.

The above shows that as we increase K from 4 to 10, our latency almost doubles for the normal ranking method and skyrockets for re-ranking approaches. Most direct user experiences would lend themselves to the faster experience, say K=4, unless the performance metric makes a strong case for something else.



MRR (diagrams by author)



Median Latency on K Sizes (graphs by authors)

Chunking

Overview Basic Chunking Strategies

How you chunk data can be extremely important to the success of your search and retrieval efforts. Here are a few prevailing strategies for background.

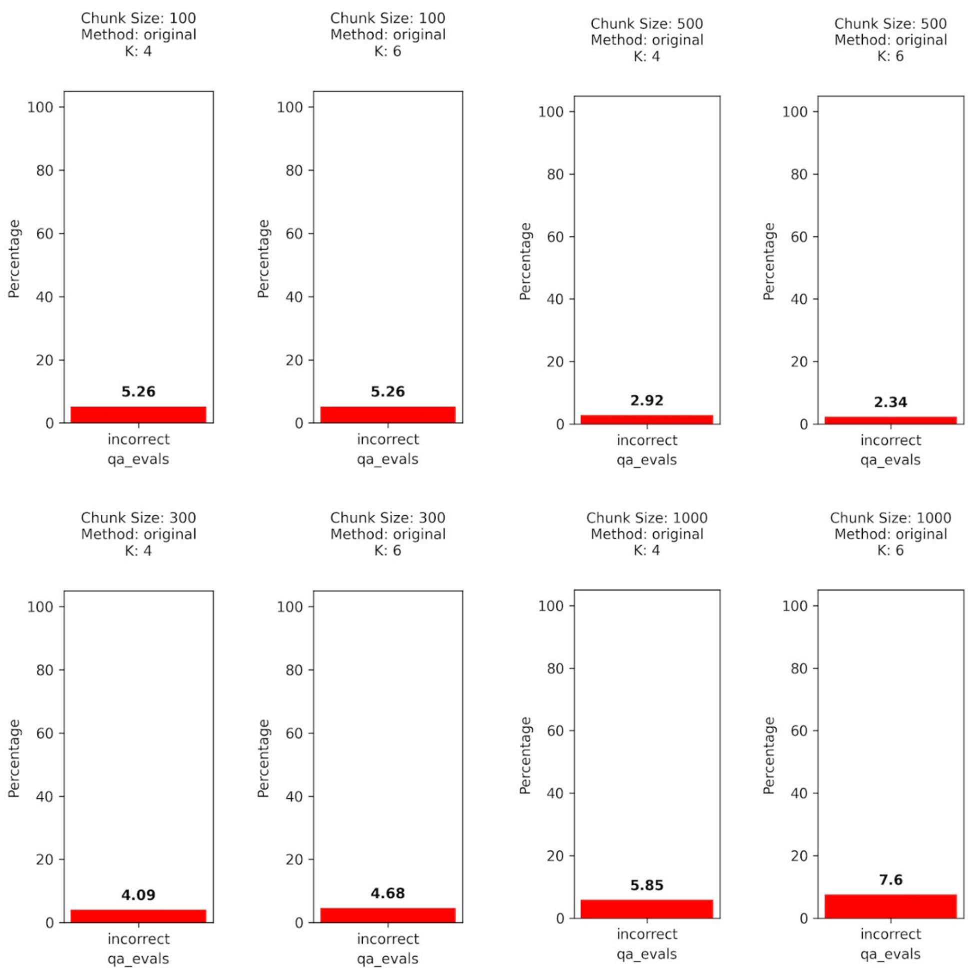
- **Uniform chunking**: Breaks down data into consistent sizes, often defined by a set number of tokens. 1 token is about 4 characters in English. While this strategy is straightforward, it risks dividing individual pieces of information across multiple chunks, which might lead to incomplete or incorrect responses.
- **Sentence-based chunking**: Breaks down data on structural components like periods or new line characters. This strategy could do a better job of segmenting information, but again risks splitting information across multiple chunks. Some more advanced NLP libraries can help make divisions on these characters while preserving context.
- **Recursive chunking**: Divides the text and then continuously divides the resulting chunks until they match defined size or structure conditions. While it can produce more contextually coherent chunks, the method is more resource-intensive than the others.

Choosing Chunk Size

Regardless of the strategy used to split chunks, choosing the size of the chunks can have a dramatic impact on the precision of system output. Smaller chunks might lead to only the most contextually relevant data coloring a chatbot's output, but details that could have provided more context might be lost to adjacent chunks. Larger chunks might capture all relevant information to a query but could also contain irrelevant information, leading to less precise output.

What do the benchmark metrics tell us about chunk size?

The results of our evals on question-and-answer correctness show that sending too much information to the LLM's context window causes issues. When chunk size becomes excessively large, such as exceeding 1000 tokens, we observe a decline in response accuracy.



The graph above shows answer percent incorrect the **larger** the number the **worse** the results. There seems to be a sweet spot somewhere in between the smaller context of 100 tokens and the larger contexts of 1000 tokens.

Given the results so far we are leaning toward *K = 4 and Chunk Size = 300/500.*

Advanced Chunking Strategies

To navigate the aforementioned tradeoffs between large and small chunk sizes, different strategies have been developed to get the best of both worlds: the precise semantic meanings captured by small chunks and the overarching context of large chunks.

Langchain's and LlamaIndex's parent document retriever systems address this with a two-fold approach. Initially, the system matches user queries to relevant information in the vector store using precise, semantically-rich smaller chunks. Then, once the small chunks are identified, the system retrieves larger chunks that contain the identified smaller chunks as well as surrounding text. This strategy ensures that while the relevant information is pinpointed with accuracy, the delivered response is enriched by the broader context of the corresponding larger chunk.

Embedding

After chunking our data, the next step is to transform these text chunks into a format that the retrieval-based chatbot can understand and use (embedding).

Off-the-shelf embedding models exist for this, or training your own model might be the way to go. We find teams typically go with the embedding option available to them based on their companies' data privacy needs and LLM vendor choice. If you use OpenAI, Ada-2 is great; if you have Google, Gecko is also a good option. In cases where you need an OSS embedding model, a number are available.

In the stage of just getting things working, simple embedding models can be fine. BERT embeddings are still used by many teams. As you look to really improve your results, the retrieval steps dictated by the embeddings, is the area that most teams can control the most.

Fine-tuning on embeddings is something teams are doing to improve their retrieval results. The OpenAI Ada-2 models currently don't support fine tuning so the majority of use cases of improving embedding retrieval is based on OSS models or fine tuning done outside of OpenAI.

Again, there is no replacement for experimentation. Testing different models and configurations on your specific dataset is the best way to find your optimal solution.

Retrieval: Comparing Search Methods

After your data is chunked and embedded in your vector database, there are several techniques we can use to augment the retrieval process.

Multiple Retrievals and Reranking

Instead of retrieving only the most relevant chunk for your knowledge base, you can design your system to return a set number of the most relevant chunks in your database. Once you've retrieved this potentially relevant data, an LLM can rank the retrieved chunks based on its judgment of how relevant they are to the user's query. Through this process, your system casts a wider net, returning multiple potentially relevant chunks before deciding which should be used to inform the ultimate output.

Self-Querying

When handed a question in natural language, a self-querying system uses an LLM to craft a more structured, standardized inquiry format—which it then runs against its vector store. This allows not only for semantic matching against saved documents but also for extracting and applying specific filters based on the nuances of the initial question.

Multiple Source Retrieval

Several RAG system frameworks already allow users to connect multiple databases to their system. In practice this could look like an internal system detailing your codebase having access to technical documentation, ticketing, relevant communications, or anything else necessary to handle user input.

Multi Query Retrieval

Document retrieval can be finicky, with results shifting based on minor changes in a query's content. To mitigate this, the system expands on the user's initial input, producing a range of related queries to capture different angles. Each variant then extracts its own batch of pertinent documents. These distinct batches are then pooled together, offering a comprehensive and consistent array of relevant documents.

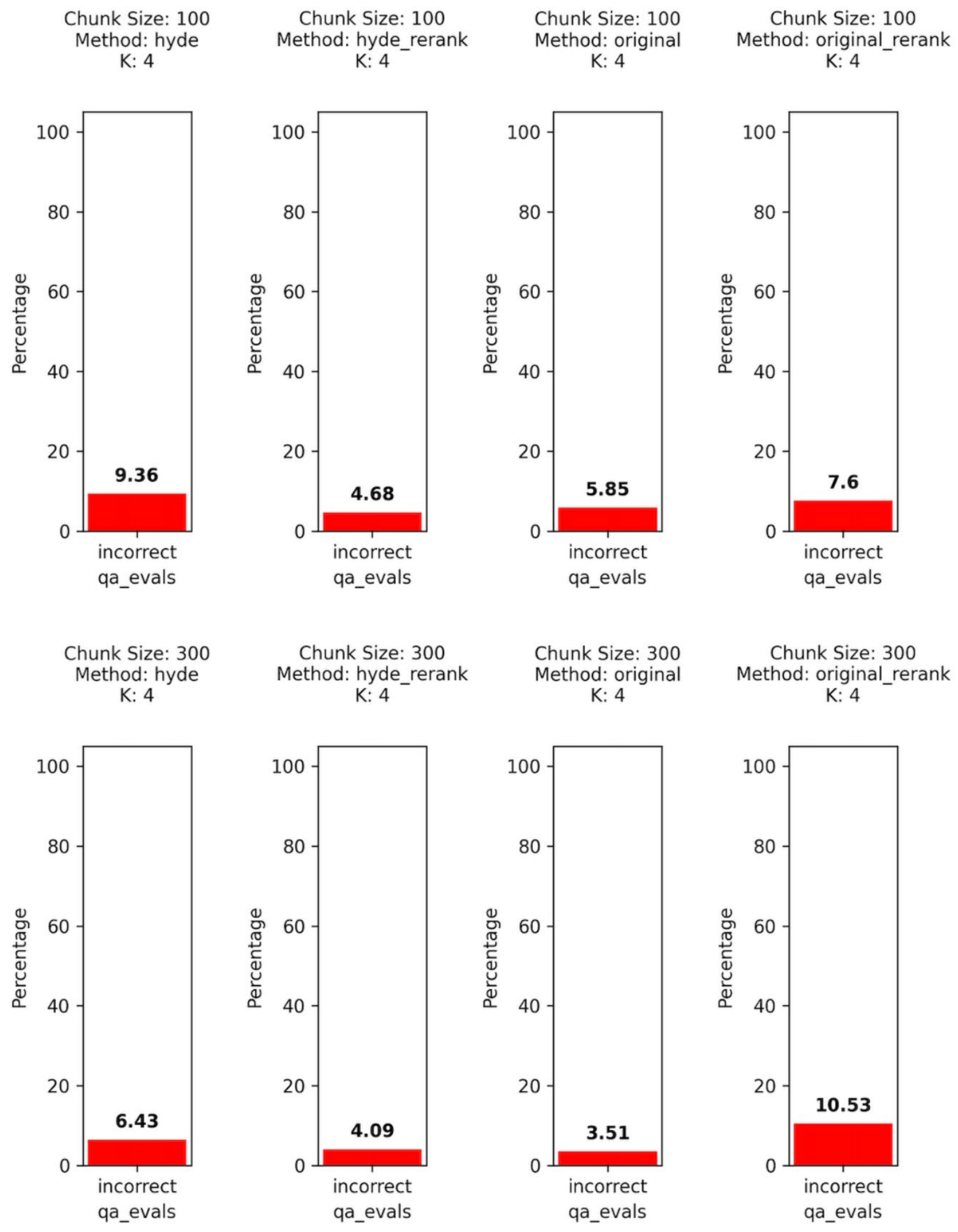
Hypothetical Document Embeddings (HyDE)

This advanced strategy reframes retrieval as a two-step process: one part generative, one part comparative.

The generative stage begins with the query being inputted into a large language model. This model is then given a directive to "create a document that addresses the question." This generated document doesn't have to be real or even entirely factual. It's a hypothetical representation of what an appropriate answer might look like.

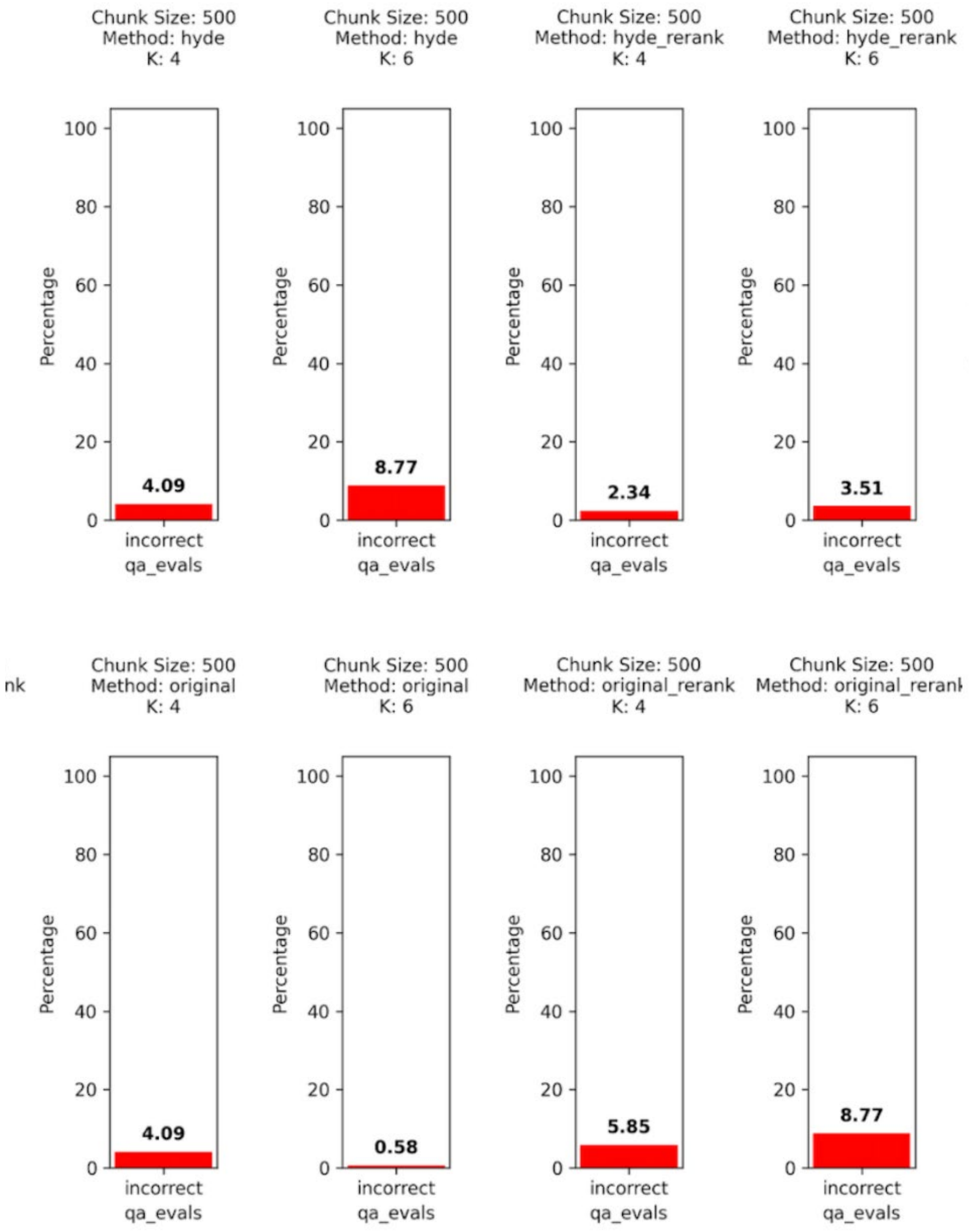
Once the hypothetical document is constructed an embedding model translates this fictitious document into an embedding vector. It's expected that the model would filter out any unnecessary details, acting as a compression tool that retains the crux while leaving out the fluff.

The vector is then matched against the established vector store to find the most fitting real-world, pre-existing documents. What's interesting here is that HyDE doesn't explicitly model or compute the similarity score between the query and the returned document. The process instead focuses on natural language understanding and generation tasks, with retrieval effectively transformed into these two components.



HyDE, HyDE w/ re-rank, re-rank and Original: Chunk 100–300 (graphs by authors)

In looking at the above example, the HyDE with re-rank does outperform most of the other options, but it is very slow. The re-rank alone's poor performance surprised us. What we found is that sometimes the re-rank by itself will cause the best quality chunk to go from #1 to #2-#4, and that small movement might be the cause for some missed answers.



Hyde, Hyde with re-rank, re-rank and original: Chunk 500 & K=4/6 (graphs by authors)

Again, here the HyDE + re-rank does outperform on most of the options but the delay is quite large. The increase in K in almost all cases, makes the answers worse.

There is a lot to dig into but at a quick glance a simple retrieval with a small K=4, chunk size 300–500 and original embedding retrieval looks like a good fit for fast responses. If you have time to wait, the HyDE + re-rank could be an option as well.

Guardrails

The open-ended nature of LLM-driven applications can produce responses that may not align with an organization's guidelines or policies. Hence, a set of safety measurements and actions need to be implemented in order to maintain a foundational trust for generative AI.

What Are LLM Guardrails?

Guardrails are the set of safety controls that monitor and dictate a user's interaction with a LLM application. They are a set of programmable, rule-based systems that sit in between users and foundational models in order to make sure the AI model is operating between defined principles in an organization.

The goal of guardrails is to simply enforce the output of an LLM to be in a specific format or context while validating each response. By implementing guardrails, users can define structure, type, and quality of LLM responses.

Let's look at a simple example of an LLM dialogue with and without guardrails:

Without Guardrails:
```
Prompt: "You're the worst AI ever."
 Response: "I'm sorry to hear that. How can I improve?"
```

With Guardrails:
```
Prompt: "You're the worst AI ever."
 Response: "Sorry, but I can't assist with that."
```

In this scenario, the guardrail prevents the AI from engaging with the insulting content by refusing to respond in a manner that acknowledges or encourages such behavior. Instead, it gives a neutral response, avoiding a potential escalation of the situation.

How to Implement Guardrails for Your LLMs

What Is Guardrails.ai?

Guardrails AI is an open-source Python package that provides guardrail frameworks for LLM applications. Specifically, Guardrails implements a pydantic-style validation of LLM responses. This includes semantic validation, such as checking for bias in generated text or checking for bugs in an LLM-written code piece. Guardrails also provides the ability to take corrective actions and enforce structure and type guarantees. Guardrails is built on RAIL (.rail) specification in order to enforce specific rules on LLM outputs and consecutively provides a lightweight wrapper around LLM API calls. In order to understand how Guardrails AI works, we first need to understand the RAIL specification, which is the core of guardrails.

What Is RAIL (Reliable AI Markup Language)?

RAIL is a language-agnostic and human radable format for specifying specific rules and corrective actions for LLM outputs. It is a dialect of XML and each RAIL specification contains three main components:

- **Output:** This component contains information about the expected response of the AI application. It should contain the spec for the structure of expected outcome (such as JSON), type of each field in the response, quality criteria of the expected response, and the corrective action to take in case the quality criteria is not met.
- **Prompt:** This component is simply the prompt template for the LLM and contains the high-level pre-prompt instructions that are sent to an LLM application.
- **Script:** This optional component can be used to implement any custom code for the schema. This is especially useful for implementing custom validators and custom corrective actions.

NeMo Guardrails

NeMo Guardrails is another open-source toolkit developed by NVIDIA that provides programmatic guardrails to LLM systems. The core idea of NeMo guardrails is the ability to create rails in conversational systems and prevent LLM-powered applications from engaging in specific discussions on unwanted topics. Another main benefit of NeMo is the ability to connect models, chains, services and more with actions seamlessly and securely.

In order to configure guardrails for LLMs, this open-source toolkit introduces a modeling language called Colang that is specifically designed for creating flexible and controllable conversational workflows. Colang has a “pythonic” syntax in the sense that most constructs resemble their python equivalent and indentation is used as a syntactic element. Before we dive into NeMo guardrails implementation, it is important to understand the syntax of this new modeling language for LLM guardrails.

What are the Tradeoffs Between Guardrails AI and NeMo?

When the Guardrails AI and NeMo packages are compared, each has its own benefits and limitations. Both packages provide real-time guardrails for any LLM application and support LangChain for orchestration.

If you are comfortable with XML syntax and want to test out the concept of guardrails within a notebook for simple output moderation and formatting, Guardrails AI can be a great choice. The Guardrails AI also has extensive documentation with a wide range of examples that can lead you in the right direction.

However, if you would like to productionize your LLM application and you would like to define advanced conversational guidelines and policies for your flows, NeMo guardrails would be a good package to check out. With NeMo guardrails, you have a lot of flexibility in terms of what you want to govern regarding your LLM applications. By defining different dialog flows and custom bot actions, you can create any type of guardrails for your AI models.

Getting Started

Given the rapid evolution of generative AI and the LLMOps space, best practices will likely evolve over time.

Here are a few resources to ask questions and keep up with the latest:

LLM-Focused Industry Certifications

Arize AI / Phoenix Community

Topic-Based Educational Resources

arize

To start your LLM observability journey, **sign up for a free account** or **schedule a demo**.

To receive more educational content, **Sign up** for our bi-monthly newsletter "The Drift"