arize

The Definitive Guide to

LLM APP EVALUATION

A practical guide to building and implementing evaluation strategies for AI applications

Table of contents

Table of contents continued

Introduction

Why are evals important?

Large language models (LLMs) are an incredible tool for developers and business leaders to create new value for consumers. They make personal recommendations, translate between structured and unstructured data, summarize large amounts of information, and do so much more.

As the applications multiply, so does the importance of measuring the performance of LLM-powered systems.

Developers using LLMs build applications to respond to user queries, transform or generate content, and classify and structure data.

It's extremely easy to start building an AI application using LLMs because developers no longer have to collect labeled data or train a model. They only need to create a prompt to ask the model for what they want. However, this comes with tradeoffs. LLMs are generalized models that aren't fine tuned for a specific task. With a standard prompt, these applications demo really well, but in production environments, they often fail in more complex scenarios.

You need a way to judge the quality of your LLM outputs. An example would be judging the quality of these chat outputs on relevance, hallucination %, and conversation correctness.

When you adjust your prompts or retrieval strategy, you will know whether your application has improved—and by how much—using evaluation. The dataset you are evaluating determines how trustworthy and generalizable your evaluation metrics are to production use. A limited dataset could showcase high scores on evaluation metrics, but perform poorly in real-world scenarios.

Paradigm Shift

While at first glance the shift from traditional software testing methods like integration and unit testing to LLM application evaluations may seem drastic, both approaches share a common goal: ensuring that a system behaves as expected and delivers consistent, reliable outcomes. Fundamentally, both testing paradigms aim to validate the functionality, reliability, and overall performance of an application.

In traditional software engineering:

- **Unit Testing** isolates individual components of the code, ensuring that each function works correctly on its own.
- **Integration Testing** focuses on how different modules or services work together, validating the correctness of their interactions.

In the world of LLM applications, these goals remain, but the complexity of behavior increases due to the non-deterministic nature of LLMs.

- **Dynamic Behavior Evaluation:** Rather than testing deterministic code, LLM evaluations focus on how the application responds to various inputs in non-deterministic situations, examining not just accuracy but also context relevance, coherence, and product level user experience.
- **Task or Product Level Assessments:** Evaluations are now centered on the application's ability to complete user-specific tasks, such as resolving queries, generating coherent responses, or interacting seamlessly with external systems (e.g., function calling). The eval is often a product experience assessment of how well the AI is working on intelligent tasks.

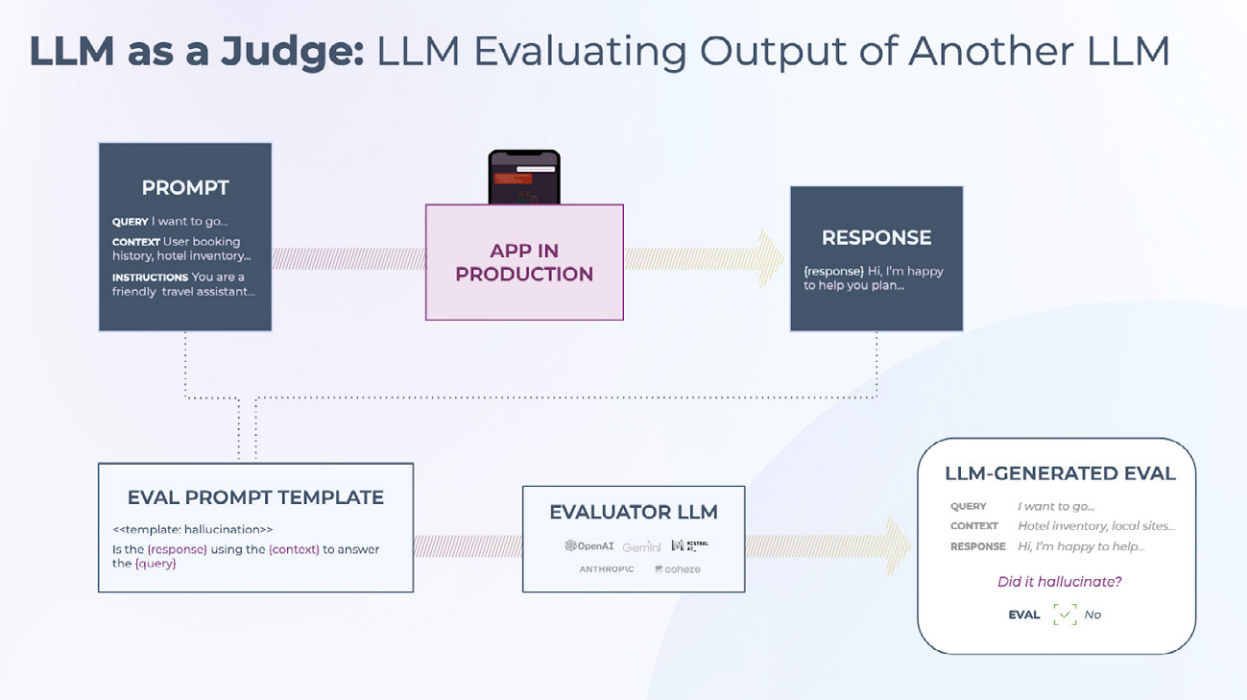
Both paradigms emphasize predictability and consistency, with the key difference being that LLM applications require dynamic, context-sensitive evaluations, as their outputs can vary with different inputs. However, the underlying principle remains: ensuring that the system (whether it's traditional code or an LLM-driven application) performs as designed, handles edge cases, and delivers value reliably.

LLM Eval Types

LLM as a Judge Eval

Often called LLM as a judge, LLM-assisted evaluation uses AI to evaluate AI — with one LLM evaluating the outputs of another and providing explanations.

LLM-assisted evaluation is often needed because 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 LLM applications complex.



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

While using AI to evaluate AI 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 often do not 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. In such cases, LLM-assisted evals help you benchmark and test in development prior to production. 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.

```
You are given a question, an answer and reference text. You must
determine whether the given answer correctly answers the question based
on the reference text. Here is the data:
    [BEGIN DATA]
    ************
    [Question]: {question}
    ************
    [Reference]: {context}
    ************
    [Answer]: {sampled_answer}
    [END DATA]
Your response must be a single word, either "correct" or "incorrect",
and should not contain any text or characters aside from that word.
"correct" means that the question is correctly and fully answered by
the answer. "incorrect" means that the question is not correctly or
only partially answered by the answer.
```

Code-based Eval

Code-based LLM evaluations are methods that use programming code to assess the performance, accuracy, or behavior of large language models (LLMs). These evaluations typically involve creating automated scripts or CI/CD test cases to measure how well an LLM performs on specific tasks or datasets. A code-based eval is essentially a python or JS/TS unit test.

Code-based evaluation is sometimes preferred as a way to reduce costs as it does not introduce token usage or latency. When evaluating a task such as code generation, a code-based eval will often be the preferred method since it can

be hard coded, and follows a set of rules. However, for evaluation that can be subjective, like hallucination, there's no code evaluator that could provide that label reliably, in which case LLM as a Judge needs to be used.

Common use cases for code-based evaluators include:

LLM APPLICATION TESTING

Code-based evaluations can test the LLM's performance at various levels—focusing on ensuring that the output adheres to the expected format, includes necessary data, and passes structured, automated tests.

> **Test Correct Structure of Output:** In many applications, the structure of the LLM's output is as important as the content. For instance, generating JSON-like responses, specific templates, or structured answers can be critical for integrating with other systems.

> **Test Specific Data in Output:** Verifying that the LLM output contains or matches specific data points is crucial in domains such as legal, medical, or financial fields where factual accuracy matters.

> **Structured Tests:** Automated structured tests can be employed to validate whether the LLM behaves as expected across various scenarios. This might involve comparing the outputs to expected responses or validating edge cases.

EVALUATING YOUR EVALUATOR

Evaluating the effectiveness of your evaluation strategy ensures that you're accurately measuring the model's performance and not introducing bias or missing crucial failure cases. Code-based evaluation for evaluators typically involves setting up meta-evaluations, where you evaluate the performance or validity of the evaluators themselves.

In order to evaluate your evaluator, teams need to create a set of hand annotated test datasets. These test datasets do not need to be large in size, 100+ examples are typically enough to evaluate your evals. In Arize Phoenix, we include test datasets with each evaluator to help validate the performance for each model type.

We recommend this guide for for a more in depth review of how to improve and check your evaluators.

Eval Output Type

Depending on the situation, the evaluation can return different types of results:

> **Categorical (Binary):** The evaluation results in a binary output, such as true/false or yes/no, which can be easily represented as 1/0. This simplicity makes it straightforward for decision-making processes but lacks the ability to capture nuanced judgements.

> **Categorical (Multi-class):** The evaluation results in one of several predefined categories or classes, which could be text labels or distinct numbers representing different states or types.

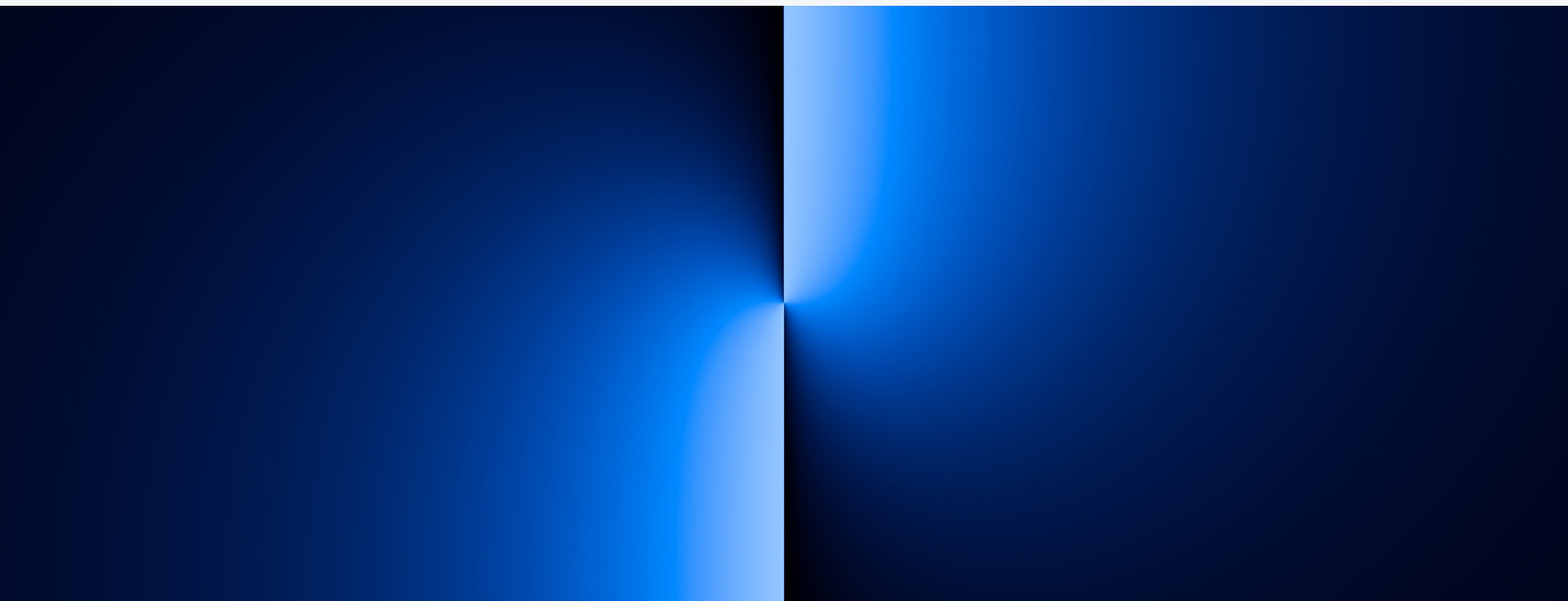
> **Continuous Score:** The evaluation results in a numeric value within a set range (e.g. 1-10), offering a scale of measurement. We don't recommend using this approach.

> **Categorical Score:** A value of either 1 or 0. The categorical score can be pretty useful as you can average your scores but don't have the disadvantages of continuous range.

Although score evals are an option, we recommend using categorical evaluations in production environments. LLMs often struggle with the subtleties of continuous scales, leading to inconsistent results even with slight prompt modifications or across different models. Repeated tests have shown that scores can fluctuate significantly, which is problematic when evaluating at scale.

Categorical evals, especially multi-class, strike a balance between simplicity and the ability to convey distinct evaluative outcomes, making them more suitable for applications where precise and consistent decision-making is important.

```
class ExampleResult(Evaluator):
    def evaluate(self, input, output, dataset_row, metadata, **kwargs) ->
EvaluationResult:
        print("Evaluator Using All Inputs")
        return(EvaluationResult(score=score, label=label,
explanation=explanation)

class ExampleScore(Evaluator):
    def evaluate(self, input, output, dataset_row, metadata, **kwargs) ->
EvaluationResult:
        print("Evaluator Using A float")
        return 1.0

class ExampleLabel(Evaluator):
    def evaluate(self, input, output, dataset_row, metadata, **kwargs) ->
EvaluationResult:
        print("Evaluator label")
        return "good"
```



Online vs Offline Evaluation

Evaluating LLM applications across their lifecycle requires a two-pronged approach: offline and online. Offline LLM evaluation generally happens during pre-production, and involves using curated or outside datasets to test the performance of your application. Online LLM evaluation runs once your app is in production, and is run on production data. The same evaluator can be used to run online and offline evaluations.

OFFLINE LLM EVALUATION

Offline LLM evaluation generally occurs during the development and testing phases of the application lifecycle. It involves evaluating the model or system in controlled environments, isolated from live, real-time data. The primary focus of offline evaluation is pre-deployment validation CI/CD, enabling AI engineers to test the model against a predefined set of inputs (like golden datasets) and gather insights on performance consistency before the model is exposed to real-world scenarios. This process is crucial for:

> **Prompt and Output Validation:** Offline tests allow teams to evaluate prompt engineering changes and different model versions before committing them to production. AI engineers can experiment with prompt modifications and evaluate which variants produce the best outcomes across a range of edge cases.

> **Golden Datasets:** Evaluating LLMs using golden datasets (high-quality, annotated data) ensures that the LLM application performs optimally in known scenarios. These datasets represent a controlled benchmark, providing a clear picture of how well the LLM application processes specific inputs, and enabling engineers to debug issues before deployment.

> **Pre-production Check:** Offline evaluation is well-suited for running CI/CD tests on datasets that reflect complex user scenarios. Engineers can check the results of offline tests and changes prior to pushing those changes to production.

Note: The "offline" part of "offline evaluations" refers to the data that is being used to evaluate the application. In offline evaluations, the data is pre-production data that has been curated and/or generated, instead of production data captured

from runs of your application. Because of this, the same evaluator can be used for offline and online evaluations. Having one unified system for both offline and online evaluation allows you to easily use consistent evaluators for both techniques.

ONLINE LLM EVALUATION

Online LLM evaluation, by contrast, takes place in real-time, during production. Once the application is deployed, it starts interacting with live data and real users, where performance needs to be continuously monitored. Online evaluation provides real-world feedback that is essential for understanding how the application behaves under dynamic, unpredictable conditions. It focuses on:

> **Continuous Monitoring:** Applications deployed in production environments need constant monitoring to detect issues such as degradation in performance, increased latency, or undesirable outputs (e.g., hallucinations, toxicity). Automated online evaluation systems can track application outputs in real time, alerting engineers when specific thresholds or metrics fall outside acceptable ranges.

> **Real-Time Guardrails:** LLMs deployed in sensitive environments may require real-time guardrails to monitor for and mitigate risky behaviors like generating inappropriate, hallucinated, or biased content. Online evaluation systems can incorporate these guardrails to ensure the LLM application proactively being protected rather than reactively.

Choosing Between Online and Offline Evaluation

While it may seem advantageous to apply online evaluations universally, they introduce additional costs in production environments. The decision to use online evaluations should be driven by the specific needs of the application and the real-time requirements of the business. AI engineers can typically group their evaluation needs into three categories: offline evaluation, guardrails, and online evaluation.

> **Offline evaluation:** Offline evaluations are used for checking LLM application results prior to releasing to production. Use offline evaluations for CI/CD checks of your LLM application.
>
> *Example: Customer service chatbot where you want to make certain changes to a prompt do not break previously correct responses.*

> **Guardrail:** AI engineers want to know immediately if something isn't right and block or revise the output. These evaluations run in real-time and block or flag outputs when they detect that the system is veering off-course.
>
> *Example: An LLM application generates automated responses for a healthcare system. Guardrails check for critical errors in medical advice, preventing harmful or misleading outputs from reaching users in real time.*

> **Online evaluation:** AI engineers don't want to block or revise the output, but want to know immediately if something isn't right. This approach is useful when it's important to track performance continuously but where it's not critical to stop the model's output in real time.
>
> *Example: An LLM application generates personalized marketing emails. While it's important to monitor and ensure the tone and accuracy are correct, minor deviations in phrasing don't require blocking the message. Online evaluations flag issues for review without stopping the email from being sent.*

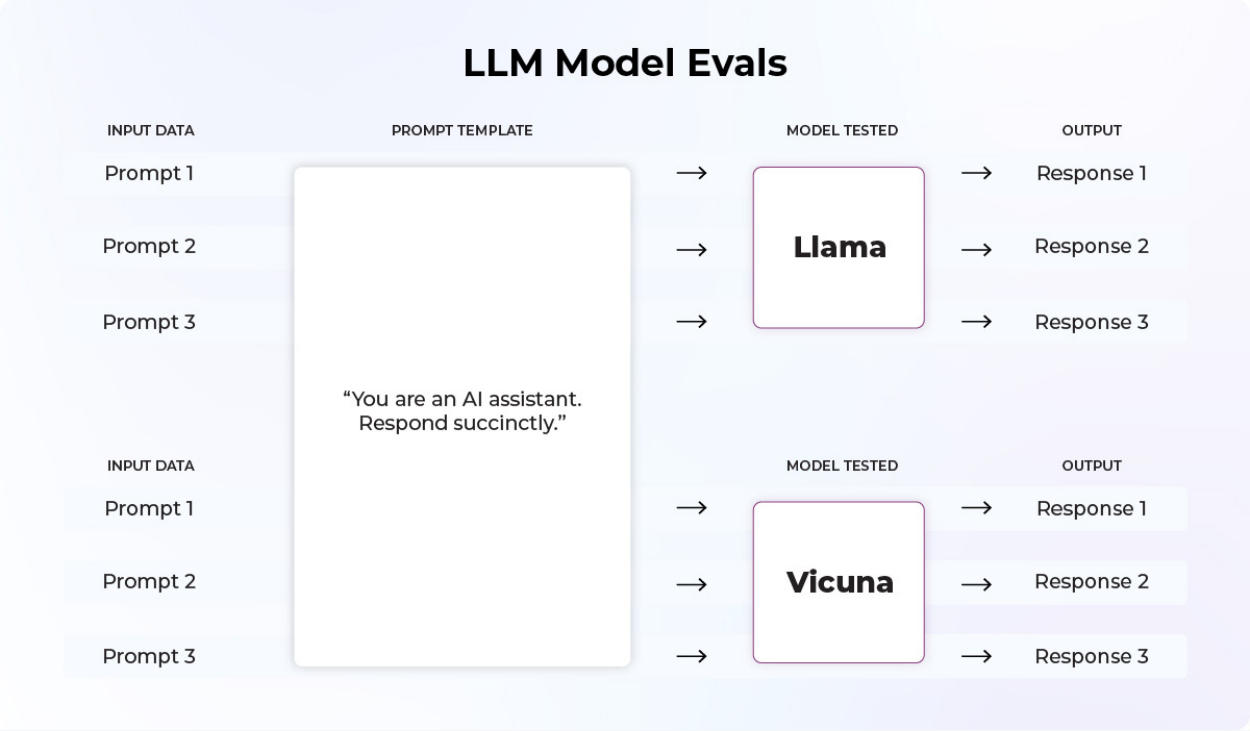
Model Benchmarks vs Task Evals

LLM model evaluations look at overall macro performance of LLMs at an array of tasks and LLM system evaluations — also referred to as LLM task evaluations — are more system and use-case specific, evaluating components an AI engineer building an LLM app can control (i.e. the prompt template or context).

Since the term "LLM evals" gets thrown around interchangeably, this distinction is sometimes lost in practice. It's critical to know the difference, however.

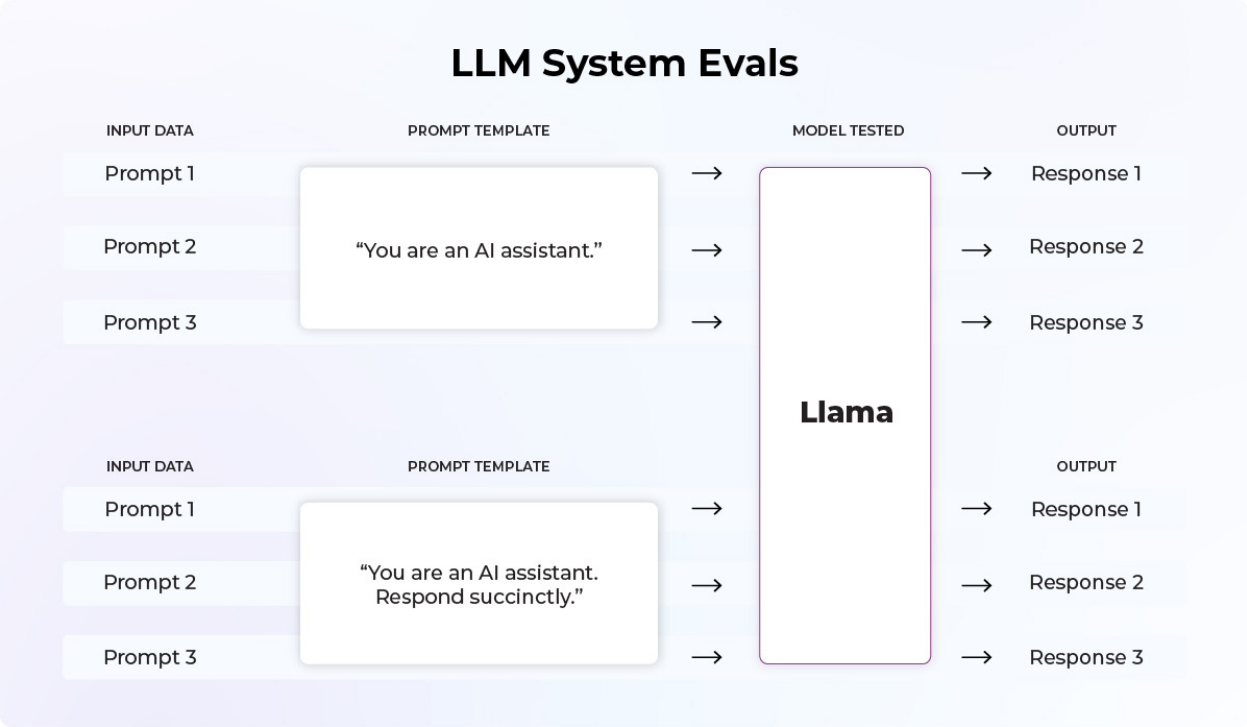
Why? Often, teams consult LLM leaderboards and libraries when such benchmarks may not be helpful for their particular use case. Ultimately, AI engineers building LLM apps that plug into several models or frameworks or tools need a way to objectively evaluate everything at highly specific tasks – necessitating system evals that reflect that fact.

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 contrast, **LLM system evaluation**, also sometimes referred to as LLM task 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 evaluation 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.



There are a lot of common LLM evaluation metrics being employed today, such as relevance, hallucinations, question-answering accuracy, toxicity, and retrieval-specific metrics. However, most teams handcraft metrics based on their business use cases. Each one of these LLM system evals will have different templates based on what is being evaluated.

What are you evaluating?

When evaluating LLM applications, the primary focus is on three key areas: the task, historical performance, and golden datasets. **Task-level evaluation** ensures that the application is performing well on specific use cases, while **historical traces** provide insight into how the application has evolved over time. Meanwhile, **golden datasets** act as benchmarks, offering a consistent way to measure performance against well-established ground truth data.

Pre-Production

Human Annotation and Curated Datasets

In the process of evaluating large language models (LLMs) or other AI systems, building a high-quality, curated golden dataset is essential. A curated golden dataset refers to a collection of examples that are carefully crafted and validated to serve as the "ground truth" or benchmark for evaluating the model's performance. This dataset forms the backbone of many evaluation strategies, ensuring that the results are reliable and consistent.

> **1. Creating a Hand-Crafted Dataset**
> The simplest form of a curated golden dataset starts with hand-crafted examples. In this approach, subject matter experts or dataset designers create examples manually to capture different aspects of the task or domain being evaluated. These examples represent various inputs that the model is expected to handle, along with their correct or expected outputs.

For example, if evaluating a model for summarization, you could manually create a series of text passages with corresponding summaries that are ideal representations of what the model should generate. The strength of this approach is that it allows for the creation of nuanced and challenging examples tailored to specific use cases or edge cases.

2. Annotating the Dataset with Ground Truth
Once the hand-crafted dataset is created, human annotators play a key role in adding ground truth. Ground truth refers to the correct answers or labels that serve as the standard for evaluation. In some cases, annotators may need to modify or refine the original labels if additional examples are included later.

For instance, annotators could be asked to label the output of a language model as "correct" or "incorrect" based on whether it matches the expected behavior. This ground truth will then serve as the reference point when evaluating how well the model performs in comparison.

3. Multi-Annotator Validation: Ensuring Consensus and Accuracy
To ensure the quality and reliability of the ground truth data, it's crucial to validate the annotations across multiple annotators. In this process, several annotators independently label or validate each example. A common approach is to use the consensus of at least two out of three annotators to confirm the correct label. If two out of three annotators agree on the same label, that label becomes the confirmed ground truth.

This multi-annotator strategy helps to reduce bias or errors that could arise from individual perspectives and ensures that the dataset is robust and reliable. Additionally, it is common to perform checks to ensure that annotators have a high level of agreement (called inter-annotator agreement), which further strengthens the trustworthiness of the dataset.

Human Annotation in LLM Evaluation

Human annotation is a critical component in evaluating LLMs and other AI systems. By combining hand-crafted datasets, ground truth labels, and multi-annotator validation, organizations can create a golden dataset that is rich in accuracy and diversity. This dataset allows for a more meaningful and comprehensive evaluation of model performance, providing the context needed to interpret automated metrics and improve model outputs.

Annotations and User Feedback

The rise of Reinforcement Learning from Human Feedback (RLHF) has highlighted the importance of human feedback in training and refining LLM applications. Whether you're manually labeling subtle response variations, curating datasets for experimentation, or logging real-time user feedback, having a robust system for capturing and cataloging annotations is critical to improving the performance and accuracy of your LLM application.

Annotations are custom labels that can be added to LLM traces or spans, allowing AI engineers to track performance and gather insights at a granular level. Annotations help to:

- **Annotate Production Span Data:** eams want to annotate data directly on top of production responses allowing those annotations to be used for filtering, analytics or production data analysis.
- **Categorize Spans or Traces:** Assign categories to specific parts of a conversation or output, enabling more detailed analysis of where a model succeeds or fails.
- **Annotate Datasets for Experimentation:** Use human-labeled data to create high-quality datasets for testing and refining LLM applications, for handcrafted CI/CD tests, few-shot prompting, or targeted evaluations.

- **Annotation Queues:** The use of annotation queues has grown in use in LLM Observability tools. The queues in Arize do not move data, but assign labeling tasks to annotators on top of current data, either production data or dataset data. When labels are needed on very specific types of data, that data is added to queues, and annotators work through those specific queues. The added labels will then appear on the original data.

- **Log Real-Time Feedback:** Collect feedback from live applications through APIs, allowing for dynamic, continuous improvements based on actual user interactions.These are not always viewed as annotations but are worth a mention in this section.

Annotations are particularly valuable for:

- **Evaluating Agreement/Disagreement:** Identifying where human evaluators and LLM evaluations align or diverge can reveal areas for improving automated evaluations.

- **Subject Matter Expertise:** In complex domains (e.g., medical, legal, or customer service applications), expert feedback is crucial to determining the quality of the application. This input complements automated metrics and provides deeper insight into how well the application performs in specialized contexts.

- **Gathering Direct Application Feedback:** Integrate feedback mechanisms directly into live applications, capturing user responses and experiences to continuously improve LLM outputs.

A well-implemented annotation and feedback system is essential for refining LLM applications, ensuring that human expertise and real-world use cases are properly incorporated into the evaluation process.

Creating and Validating Synthetic Datasets as Golden Datasets

Synthetic datasets are artificially created datasets that are designed to mimic real-world information. Unlike naturally occurring data, which is gathered from actual events or interactions, synthetic datasets are generated using algorithms, rules, or other artificial means. These datasets are carefully created to represent specific patterns, distributions, or scenarios that developers and researchers want to study or use for testing.

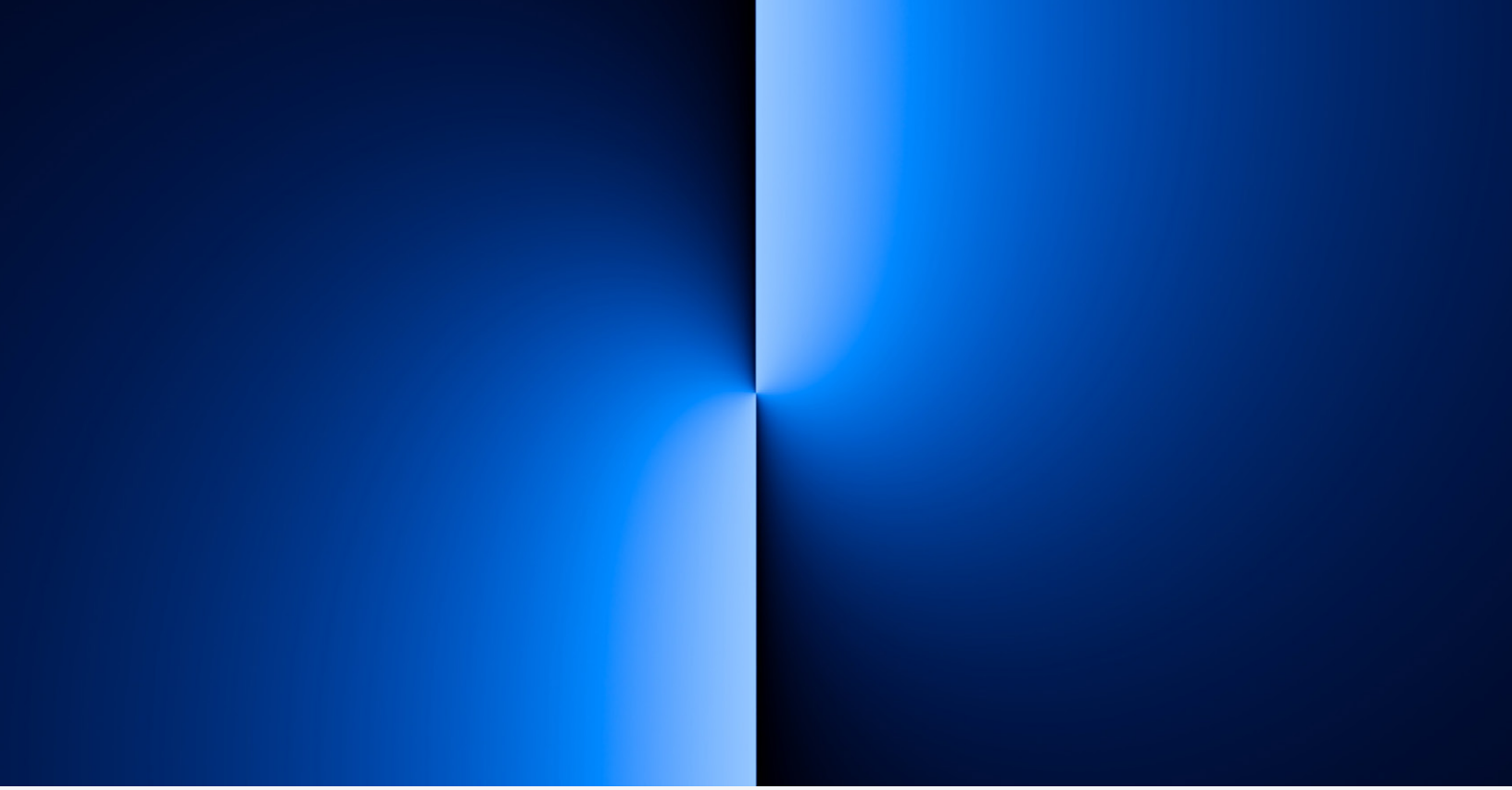
In the context of large language models, synthetic datasets might include:

> Generated text conversations simulating customer support interactions.
>
> Artificial question-answer pairs covering a wide range of topics.
>
> Fabricated product reviews with varying sentiments and styles.
>
> Simulated code snippets with intentional bugs or specific patterns.

By using synthetic data, developers can create controlled environments for experimentation, ensure coverage of edge cases, and protect privacy by avoiding the use of real user data.

The applications of synthetic datasets are varied and valuable:

> They allow us to test and validate model performance, especially for assessing how well models perform specific tasks.
>
> Synthetic data helps generate initial traces of application behavior, facilitating debugging in tools like Arize.
>
> Perhaps most importantly, synthetic datasets serve as “golden data” for consistent experimental results. This is particularly useful when developing and experimenting with applications that haven’t yet launched.



COMBINING SYNTHETIC DATASETS WITH HUMAN EVALUATION

While synthetic datasets offer many advantages, they may sometimes miss key use cases or types of inputs that humans would naturally consider. Human-in-the-loop processes are valuable for dataset improvement.

Recent research has shown that including even a small number of human-annotated examples can significantly improve the overall quality and effectiveness of a synthetic dataset. This hybrid approach combines the scalability of synthetic data with the understanding that human evaluators provide.

Human annotators can add targeted examples to synthetic datasets to address gaps or underrepresented scenarios. This process of augmenting synthetic data with human-curated examples can be easily implemented using tools like Arize.

The addition of these human-annotated examples can be particularly effective in improving the dataset's performance in few-shot learning scenarios, where models need to generalize from a small number of examples.

Best Practices for Synthetic Dataset Use

Synthetic datasets are not static, one-time creations – they are dynamic tools that require ongoing attention. To maintain their usefulness, developer must do several things:

First, implement a regular refresh cycle. Revisit and update your datasets periodically to keep pace with model improvements and account for data drift in real-world applications.

Second, transparency is key in synthetic data generation. Maintain detailed records of the entire process, including the prompts used, models employed, and any post-processing steps applied.

Third, regular evaluation is important. Assess the performance of your synthetic datasets against real-world data and newer models on an ongoing basis.

Finally, when augmenting synthetic datasets with human-curated examples, take a balanced approach. Add enough human input to enhance the dataset's quality and coverage, but be careful not to overwhelm the synthetic component.

By adhering to these practices, you can maximize the long-term value and reliability of your synthetic datasets, making them powerful tools for ongoing model evaluation and experimentation.

Learn more here.

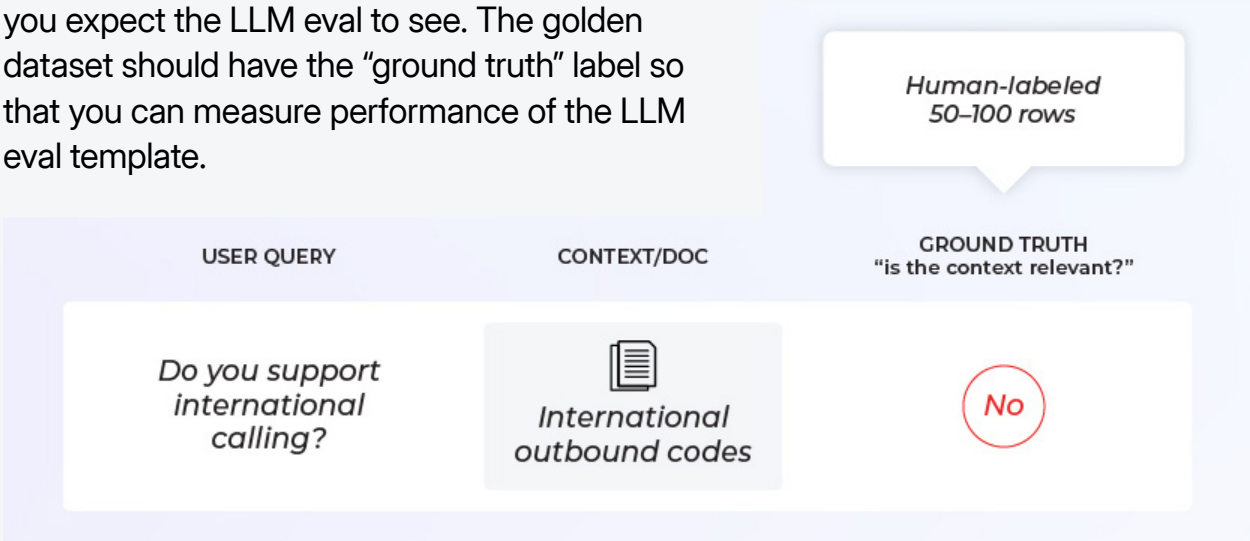
Benchmarking LLM Evaluation

Benchmarking LLM evaluation is critical to ensure your evaluation strategy addresses these core areas:

- **Evaluation Accuracy:** Does the evaluation process correctly capture the quality of outputs generated by the application? This includes testing whether your evaluation metrics (e.g., relevance, accuracy) accurately reflect real-world outcomes like user satisfaction or task completion.
- **Consistency Across Scenarios:** Benchmark the evaluation process for consistency across a variety of application scenarios, including edge cases, stress tests, and diverse input types. The goal is to ensure that evaluations remain reliable and do not favor certain cases over others.
- **Evaluation Latency:** How fast can evaluations be conducted in real-time environments? Latency in generating evaluation results is critical for applications where fast feedback loops are essential.
- **Human vs. Automated Evaluations:** Compare automated evaluation systems (such as those scoring relevance or accuracy) against human evaluators (annotators) to ensure alignment. This benchmarking helps ensure that automated processes reliably approximate human judgment.

BENCHMARKING LLM AS A JUDGE

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 you can measure performance of the LLM eval template.



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 Arize Phoenix, you should start with an existing template and see how that prompt performs.

> What is the input? In our example, it is the documents/context that was retrieved and the query from the user.

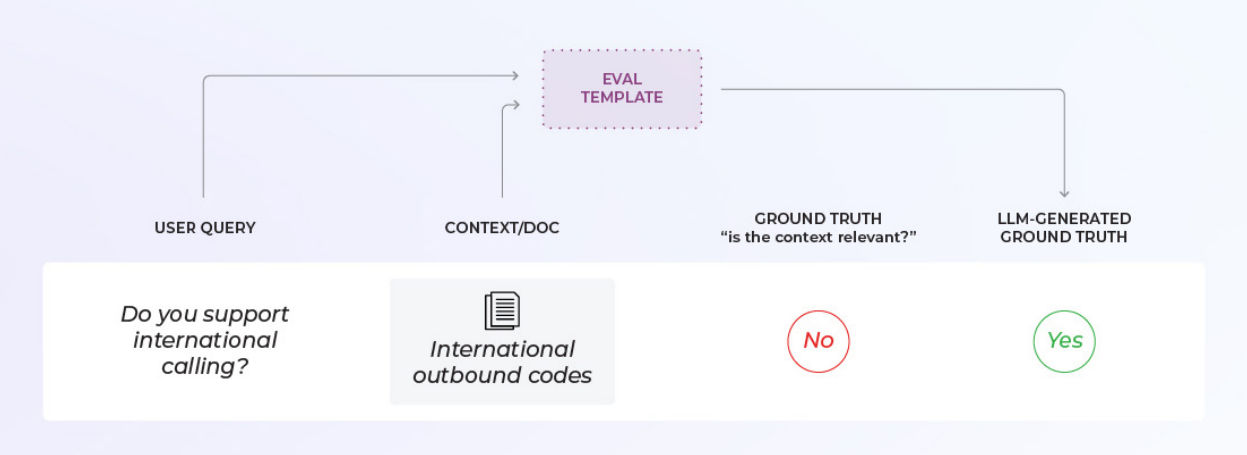
> What are we asking? In our example, we're asking the LLM to tell us if the document was relevant to the query.

> 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).

The more specific you are about how to classify or grade a response, the more accurate your LLM evaluation will become. Here is an example of a custom template which classifies a response to a question as positive or negative.

```
MY_CUSTOM_TEMPLATE = ‘’’
    You are evaluating the positivity or
negativity of the responses to questions.
    [BEGIN DATA]
    ************
    [Question]: {question}
    ************
    [Response]: {response}
    [END DATA]


    Please focus on the tone of the response.
    Your answer must be single word, either
“positive” or “negative”
    ‘’’
```

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.



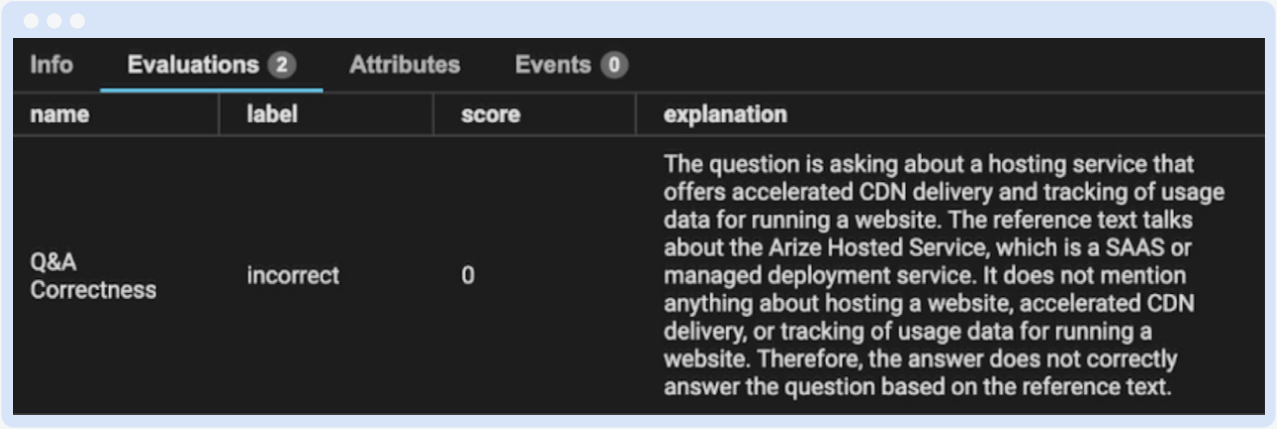
If you are not satisfied with the performance of your LLM evaluation template, you need to change the prompt to make it perform better. This is an iterative process informed by hard metrics. As is always the case, it is important to avoid overfitting the template to the golden dataset.

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

Evals with Explanations

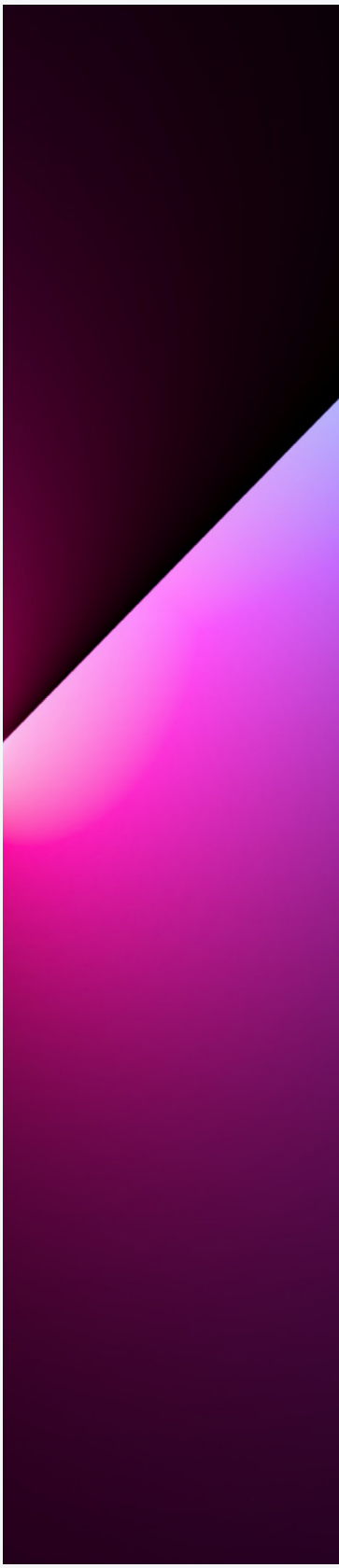
It can be hard to understand in many cases why an LLM responds in a specific way. Explanations showcase why the LLM decided on a specific score for your evaluation criteria, and may even improve the accuracy of the evaluation.



Eval Hierarchy

Evaluations can occur at different levels of granularity. At the most basic level, span-level evaluations assess the performance of specific components within an application's response. A trace-level evaluation looks at trends across multiple full runs of your application, while session-level evaluation expands the scope to multiple interactions. Understanding this hierarchy allows for a more structured and comprehensive evaluation strategy.

Building a Robust Evaluation Approach



In order to ensure that LLMs are performing optimally and reliably, a well-thought-out evaluation framework is essential. A robust evaluation approach needs to account for several factors that influence both the practical usability of the framework and its ability to scale with evolving models. There are many existing evaluation frameworks, including our evals library in Arize Phoenix, and there is always the option to build your own system. Below are the key aspects that to consider when choosing or building a solid and comprehensive evaluation system.

ERGONOMICS: HOW EASY IS IT TO USE?

A user-friendly evaluation framework is crucial for broad adoption and frequent usage. If the framework is too complex or cumbersome, it discourages experimentation and regular assessments. Good ergonomics ensure that setting up and running evaluations is intuitive, even for non-technical users or stakeholders. A good evaluation system should allow for quick setup, intuitive workflows, and ease of collaboration between different stakeholders.

PARALLELIZATION: CAN THE FRAMEWORK SUPPORT PARALLEL EVALUATION CALLS?

The ability to run evaluations in parallel is a game-changer when dealing with large-scale applications or high volumes of requests. Parallelization speeds up the evaluation process, allowing the framework to assess multiple models or multiple datasets simultaneously, leading to more efficient workflows.

ONLINE/OFFLINE EVALUATION SUPPORT

An effective evaluation framework should be adaptable to both online (real-time) and offline (batch-processed) environments. While offline evaluations are crucial for testing in a controlled environment, online evaluations offer insights into real-time performance under dynamic, real-world conditions.

FLEXIBILITY: CUSTOM TEMPLATES FOR LLM AS A JUDGE

The evaluation framework should be flexible enough to accommodate a wide range of use cases, tasks, and metrics. Customizable evaluation templates are key to this flexibility, enabling teams to tailor the evaluation process to specific requirements, including different task types or performance goals.

UI VS. NON-UI: DOES THE FRAMEWORK PROVIDE A UI OR IS IT PURE CODE?

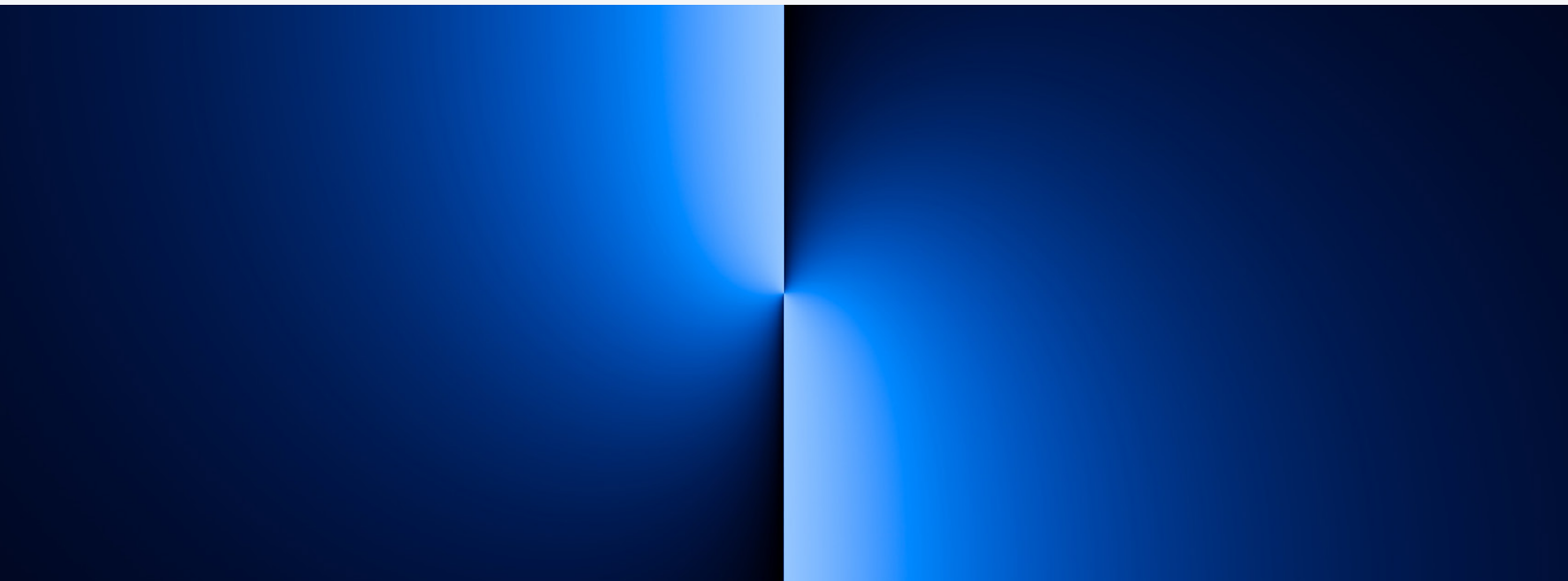
Having both a UI and a non-UI (code-based) option in an evaluation framework caters to a wider audience. A UI allows for easy setup and management of evaluations, especially for non-technical users, while a code-based option offers more customization and control for developers.

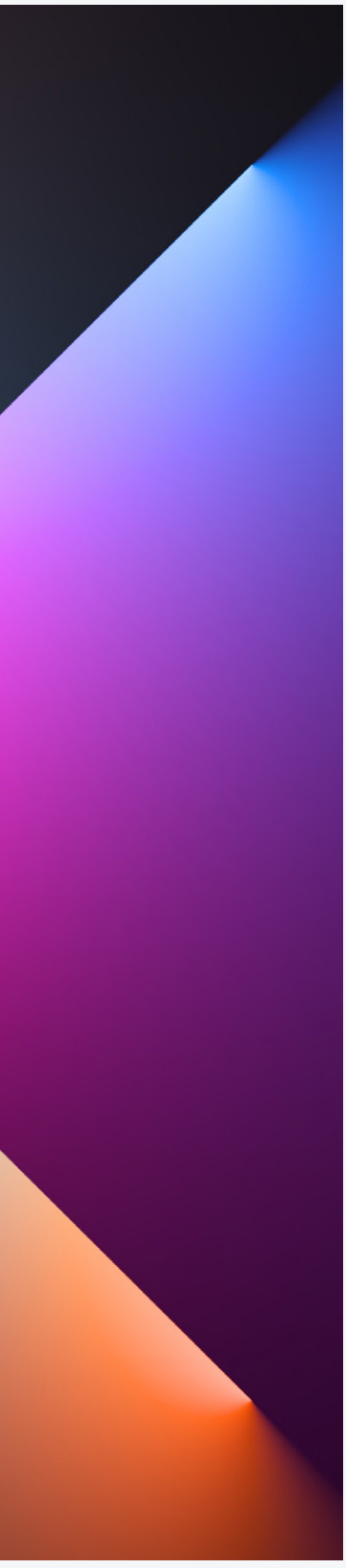
SCALE: HANDLING INCREASING COMPLEXITY AND CALLS

As the complexity of LLM applications grows, so does the need for a scalable evaluation framework. Scalability encompasses the ability to handle increasing calls, higher throughput, and more intricate evaluation criteria without sacrificing performance.

INCLUDING EXPLANATIONS: PROVIDING INTERPRETABILITY

Including explanations in the evaluation results adds a layer of interpretability to the evaluation process. Rather than simply providing a pass/fail result or a numerical score, it is essential to explain why the application performed well or poorly on specific tasks.





Choosing an Evaluation Model

Selecting the right evaluation model is crucial to ensuring that your LLM evaluation approach is effective. The choice depends on how often you are iterating on evaluations (often quite frequently), cost versus accuracy requirements, and flexibility requirements.

Base Models: Many teams when starting on Evaluations leverage the same models they are using for the LLM application. If a team is using GPT-4o or Claude Sonet or Gemini they will use the same model for Evaluations. This gives a lot of flexibility and a place to baseline before moving to a different approach that trades off accuracy vs cost.

Base SLM Models: The SLM model versions of base models are a great choice for both Evals and guardrails. The GPT-4o mini or Gemini/Gemma is 1/10 of the cost of base models and incredibly fast for guardrail type applications. These are a natural place to test against before moving to something more complex and more rigid like a fine tuned model.

LLM Fine Tune: A fine tune of a 1B Llama or Phi model can allow teams to apply a very specific eval, with a good set of language generalization at a reduced cost. We recommend moving to something like this as you have truly scaled out your application and have a set of very focused evaluations you are working to get reduced in total cost.

Fine-Tune Classifier (BERT Based): These models are incredibly cheap but represent a large trade off in generalization ability (BERT does not work across languages or situations well) versus flexibility (if you want to change you need to retrain) with low cost. If cost is your main objective and you are laser focused on a single language and use case, the BERT fine tune might make sense, though there are so many better options we recommend against.

CI/CD

Experiments

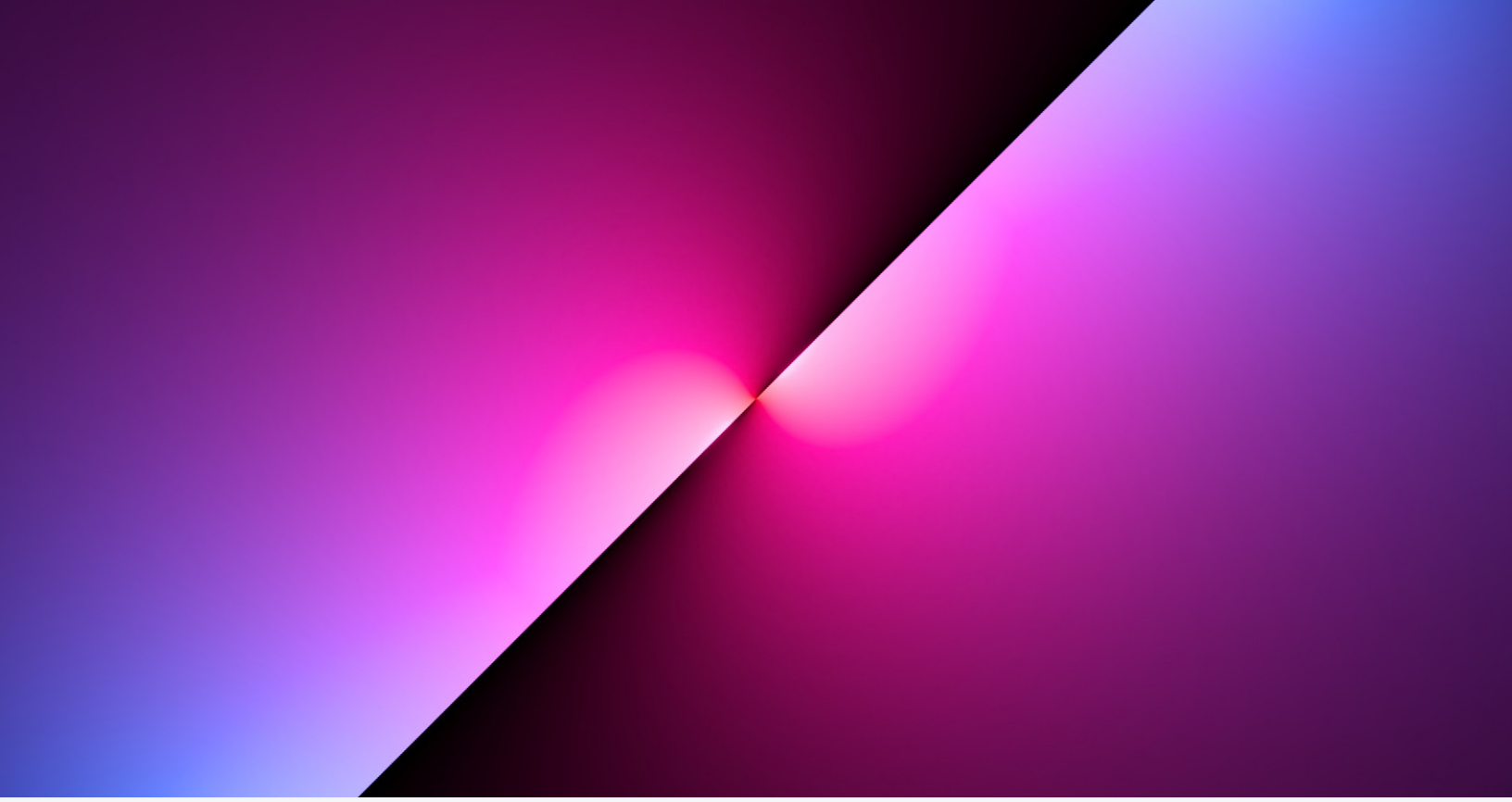
An experiment allows you to systematically test and validate changes in your LLM applications using a curated dataset. By defining a dataset (a collection of examples), creating tasks to generate outputs, and setting up evaluators to assess those outputs, you can run an experiment to see how well your updated pipeline performs. Whether you're testing improvements with a golden dataset or troubleshooting issues with a problem dataset, experiments provide a structured way to measure the impact of your changes and ensure your application is on the right track.

Components of experiments:

> **Datasets:** Datasets are collections of examples that provide the inputs and, optionally, expected reference outputs for evaluating your application. These examples are used in experiments to track improvements to your prompt, LLM, or other parts of your LLM application.

> **Tasks:** A task is any function or process that produces a JSON-serializable output. Typically, a task replicates the LLM functionality you're aiming to test. For instance, if you've made a prompt change, your task will run the examples through the new prompt to generate an output. The task is used in the experiment to process the dataset, producing outputs that will be evaluated in the next steps.

> **Evaluators:** An evaluator is any function that takes the output of a task and provides an assessment. It serves as the measure of success for your experiment, helping you determine whether your changes have achieved the desired results. You can define multiple evaluators, ranging from LLM-based judges to code-based evaluations. The evaluator is central to testing and validating the outcomes of your experiment.



Application/Orchestration Changes

Experiments extend beyond simple prompt or model changes—they also allow AI engineers to test changes in how the LLM application is orchestrated. This could include testing new integrations, API interactions, or external function calls. By experimenting with application or orchestration changes, engineers can optimize the performance of the overall system, not just the LLM output itself.

For example, an application might include task orchestration changes, such as altering how functions are triggered or modifying interaction flows between the LLM and external APIs. Experiments help determine whether these changes improve response times, reduce errors, or enhance overall user experience.

Tracking New Experiments

Effective experiment tracking is essential for measuring progress and avoiding regressions. By maintaining a record of each experiment—along with its configuration, dataset, evaluators, and results—AI engineers can compare past and present performance. This ensures that each change is assessed in context, and improvements or declines are easily traceable across different experiments.

How to Read Results: It's Not Black and White

Interpreting experiment results requires nuance. Experiments often yield mixed outcomes—some evaluations may show improvements while others may not. For example, a change in prompt structure might lead to higher coherence scores but could slightly reduce factual accuracy.

Most teams hand curate annotated examples for an experiment validation set and then use metrics such as an average score, F1, recall or precision on top of evaluation results to assess performance. The use of statistical metrics is not ubiquitous, as many AI engineers come from different non-stats backgrounds, though we do see the use of statistical checks as a growing set of best practices.

Understanding that evaluation results are not always "black and white" is key. It's important to weigh trade-offs and prioritize improvements that align with the specific goals of your application. A robust evaluation process will consider the impact across multiple metrics and tasks, allowing the AI engineer to decide whether the trade-offs make sense for their application.

LLM Evaluators

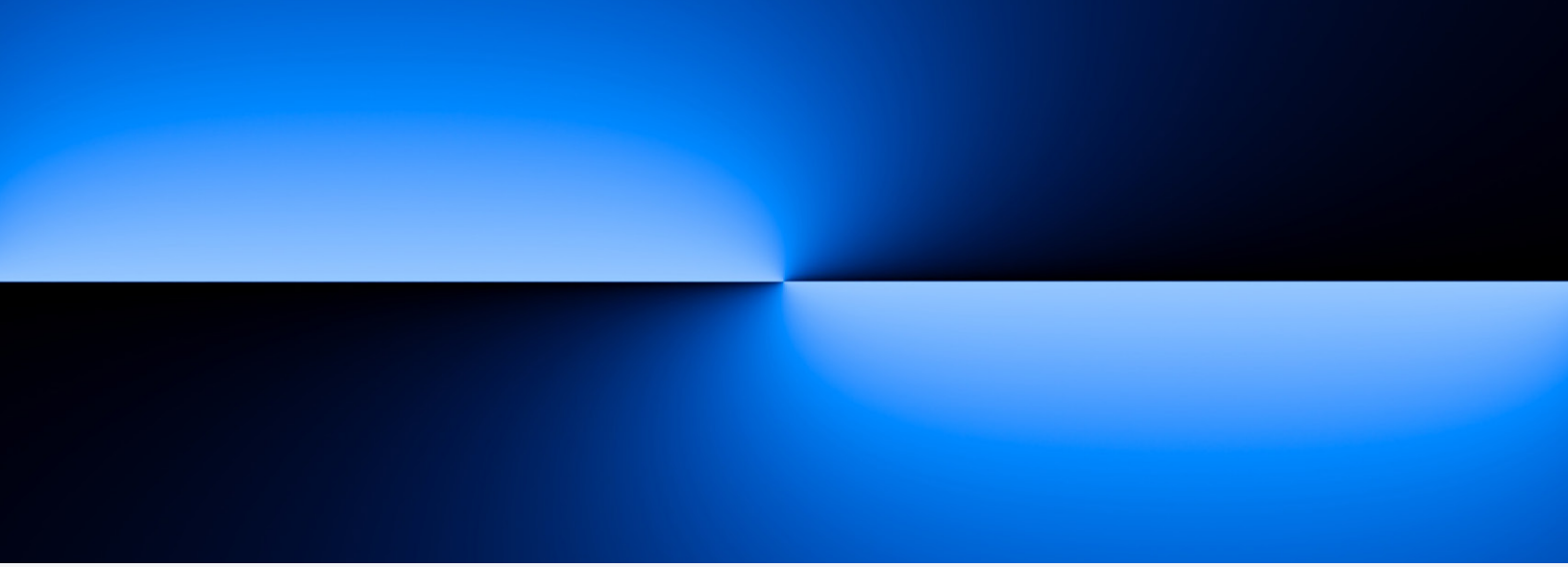
LLM evaluators utilize LLMs as judges to assess the success of your experiment. These evaluators can either use a prebuilt LLM evaluation template or be customized to suit your specific needs.

Here's an example of a LLM evaluator that checks for hallucinations in the model output:

```
from phoenix.evals import (
    HALLUCINATION_PROMPT_RAILS_MAP,
    HALLUCINATION_PROMPT_TEMPLATE,
    llm_classify,
)
from phoenix.experiments.types import EvaluationResult
from openai import OpenAIModel

class HallucinationEvaluator(Evaluator):
    def evaluate(self, output, dataset_row, **kwargs) -> EvaluationResult:
        print("Evaluating outputs")
        expected_output = dataset_row["attributes.llm.output_messages"]

        # Create a DataFrame with the actual and expected outputs
        df_in = pd.DataFrame(
            {"selected_output": output, "expected_output": expected_output}, index=[0]
        )
        # Run the LLM classification
        expect_df = llm_classify(
            dataframe=df_in,
            template=HALLUCINATION_PROMPT_TEMPLATE,
            model=OpenAIModel(model="gpt-4o-mini", api_key=OPENAI_API_KEY),
            rails=HALLUCINATION_PROMPT_RAILS_MAP,
            provide_explanation=True,
        )
        label = expect_df["label"][0]
        score = 1 if label == rails[1] else 0  # Score 1 if output is incorrect
        explanation = expect_df["explanation"][0]

        # Return the evaluation result
        return EvaluationResult(score=score, label=label, explanation=explanation)
```

In this example, the HallucinationEvaluator class evaluates whether the output of an experiment contains hallucinations by comparing it to the expected output using an LLM. The llm_classify function runs the eval, and the evaluator returns an EvaluationResult that includes a score, label, and explanation.

You can customize LLM evaluators to suit your experiment's needs, whether you're checking for hallucinations, function choice, or other criteria where an LLM's judgment is valuable. Simply update the template with your instructions and the rails with the desired output. You can also have multiple LLM evaluators in a single experiment to assess different aspects of the output simultaneously.



Code Based Eval Experiment

Code evaluators are functions designed to assess the outputs of your experiments. They allow you to define specific criteria for success, which can be as simple or complex as your application requires. Code evaluators are especially useful when you need to apply tailored logic or rules to validate the output of your model.

Creating a custom code evaluator is as simple as writing a Python function. By default, this function will take the output of an experiment run as its single argument. Your custom evaluator can return either a boolean or a numeric value, which will then be recorded as the evaluation score.

For example, let's say our experiment is testing a task that should output a numeric value between 1 and 100. We can create a simple evaluator function to check if the output falls within this range.

```
def in_bounds(output):
    return 1 <= output <= 100
```

By passing the in_bounds function to run_experiment, evaluations will automatically be generated for each experiment run, indicating whether the output is within the allowed range. This allows you to quickly assess the validity of your experiment's outputs based on custom criteria.

```
experiment = arize_client.run_
experiment(
    space_id=SPACE_ID,
    dataset_id=DATASET_ID,
    task=run_task,
    evaluators=[in_bounds],
    experiment_name=experiment_name,
)
```

Async Vs Sync Tasks and Evals

Synchronous: The synchronous running of an experiment runs one after another.

Run linear, one after another

Async: The asynchronous running of an experiment runs parallel.

Run in parallel at the same time

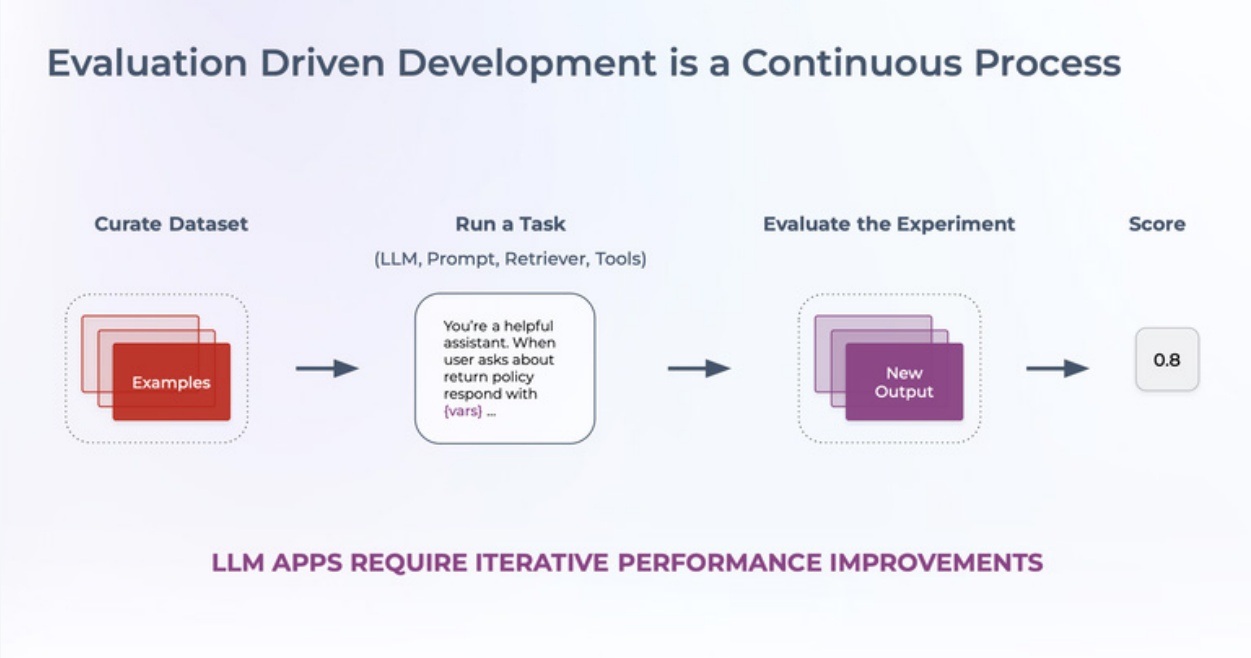
Keep these in mind when choosing between synchronous and asynchronous experiments.

> **Synchronous:** *Slower but easier to debug.* When you are building your tests these are inherently easier to debug. Start with synchronous and then make them asynchronous.

> **Asynchronous:** *Faster*. When timing and speed of the tests matter. Make the tasks and/or evals asynchronous and you can 10x the speed of your runs.

CI/CD for Automated Experimentation

Setting up CI/CD pipelines for LLMs helps you maintain control as your applications evolve. Just like in traditional software, automated testing is crucial to catch issues early. With Arize, you can create experiments that automatically validate changes—whether it's a tweak to a prompt, model, or function—using a curated dataset and your preferred evaluation method. These tests can be integrated with GitHub Actions, so they run automatically when you push a change, giving you confidence that your updates are solid without the need for manual testing.



GitHub Actions allow you to automate workflows directly from your GitHub repository. It enables you to build, test, and deploy your code based on specific events (such as code pushes, pull requests, and more).

KEY CONCEPTS OF GITHUB ACTIONS:

- **Workflows:** Automated processes that you define in your repository.
- **Jobs:** A workflow is composed of one or more jobs that can run sequentially or in parallel.
- **Steps:** Jobs contain steps that run commands in the job's virtual environment.
- **Actions:** The individual tasks that you can combine to create jobs and customize your workflow. You can use actions defined in the GitHub marketplace or create your own.

Datasets

Datasets are integral to evaluation and experimentation.

They are collections of examples that provide the inputs, outputs,and any other attributes needed for assessing your application. Each example within a dataset represents a single data point, consisting of an inputs dictionary, an optional output dictionary, and an optional metadata dictionary. The optional output dictionary often contains the expected LLM application output for the given input.

Datasets allow you to collect data from production, staging, evaluations, and even manually. The examples collected are then used to run experiments and evaluations to track improvements.

Use datasets to:

- Store evaluation test cases for your eval script instead of managing large JSONL or CSV files
- Capture generations to assess quality manually or using LLM-graded evals
- Store user reviewed generations to find new test cases

CREATING DATASETS

There are various ways to get started with datasets:

Manually Curated Examples

This is how we recommend you start. From building your application, you probably have an idea of what types of inputs you expect your application to be able to handle, and what "good" responses look like. You probably want to cover a few different common edge cases or situations you can imagine. Even 20 high quality, manually curated examples can go a long way.

Historical Logs

Once you ship an application, you start gleaning valuable information: how users are actually using it. This information can be valuable to capture and store in datasets. This allows you to test against specific use cases as you iterate on your application.

If your application is going well, you will likely get a lot of usage. How can you determine which datapoints are valuable to add? There are a few heuristics you

can follow. If possible, try to collect end user feedback. You can then see which datapoints got negative feedback. That is super valuable! These are spots where your application did not perform well. You should add these to your dataset to test against in the future. You can also use other heuristics to identify interesting datapoints - for example, runs that took a long time to complete could be interesting to analyze and add to a dataset.

Synthetic Data
Once you have a few examples, you can try to artificially generate examples to get a lot of datapoints quickly. It's generally advised to have a few good handcrafted examples before this step, as the synthetic data will often resemble the source examples in some way.

Promoting Changes

Promoting changes in LLM applications requires rigorous evaluation-driven testing to ensure new updates are stable, accurate, and aligned with the application's goals. Given the non-deterministic nature of LLMs, promoting changes based on evaluation metrics needs to be done thoughtfully, balancing between ensuring reliability and avoiding unnecessary roadblocks.

Writing Eval-Driven Development Tests
To promote changes effectively, it's crucial to integrate GitHub Actions and evaluation-driven tests into your development workflow. Since LLMs are non-deterministic, running a test once isn't enough. The key is passing evaluation tests consistently. AI engineers need to run evaluations multiple times to ensure that results are consistent across runs. Changes should pass a certain number of test runs to guarantee stability and avoid promoting changes based on one-off good results.

What Experiments Should You Run in CI/CD?
When running experiments in your CI/CD pipeline, the goal is to simulate production as closely as possible before deploying changes. Some experiments to include:

> **Ground Truth Evals:** These should be run regardless of what's being tested. If the application fails on ground truth comparisons, something is fundamentally broken, and changes should not be promoted.

> **Threshold-Based Experiments:** Set up experiments to detect if certain evaluation metrics (e.g., hallucinations or correctness) pass a defined threshold. For instance, if hallucination rates spike by 50% compared to a baseline, it may indicate a significant issue that needs addressing before promoting the change.

Should You Block PRs on Failures?
Blocking PRs on evaluation failures should be handled similarly to traditional unit tests. Some evaluation failures, like major issues in ground truth comparisons or significant metric spikes, should block PRs. However, not all evaluation tests should block a PR—some are better suited for post-deployment monitoring to avoid unnecessary bottlenecks.

This is where the paradigm shift comes in: LLM evaluations are your new unit tests. While teams may not block changes for every failure, they represent real tests that detect critical issues in your application, ensuring that only high-quality updates are promoted.

Paradigm Shift: Detaching Experiments from CI/CD

In traditional software development, Continuous Integration and Continuous Deployment (CI/CD) pipelines are designed to catch issues early, ensuring that code changes don't introduce new bugs or degrade performance. However, with LLM applications, this approach requires a significant shift. Unlike traditional applications, LLM applications are not only affected by code changes but also by model updates, or input drift in production. As a result, experiments need to be run regularly in order to detect performance shifts, even when there aren't associated pull requests or active development changes.

Production

Guardrails

As LLM applications become more common, so too do jailbreak attempts, exploitations of these apps, and harmful responses. More and more companies are falling prey to damaging news stories driven by their chatbots selling cars for $1, writing poems critical of their owners, or dealing out disturbing replies.

Fortunately, there is a solution to this problem: LLM guardrails. LLM guardrails allow you to protect your application from potentially harmful inputs, and block damaging outputs before they're seen by a user. As LLM jailbreak attempts become more common and more sophisticated, having a robust guardrails approach is critical.

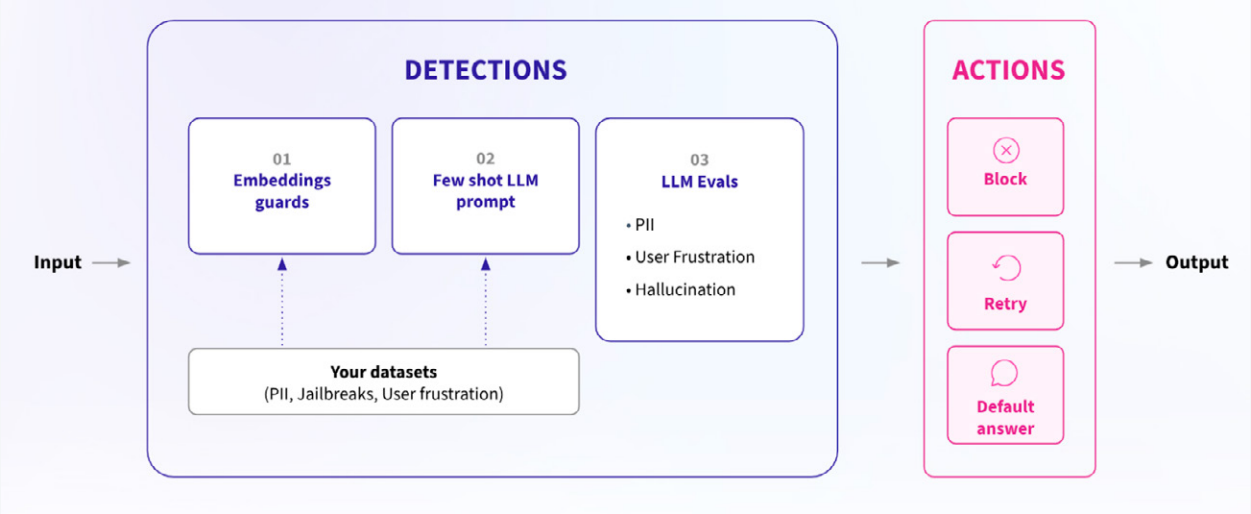
LLM guardrails work in real-time to either catch dangerous user inputs or screen model outputs. There are many different types of guards that can be employed, each specializing in a different potential type of harmful input or output.

Common input guard use cases include:

- Detecting and blocking jailbreak attempts
- Preventing prompt injection attempts
- Removing user personally identifiable information (PII) before it reaches a model

Common output guard use cases include:

- Removing toxic or hallucinated responses
- Removing mentions of a competitor's product
- Screening for relevancy in responses
- Removing NSFW text



Balancing Act of Guards

Implementing guardrails for AI systems is a delicate balancing act. While these safety measures are important for responsible AI deployment, finding the right configuration is necessary to maintain both functionality and security.

It's important to resist the temptation to over-index on guards. It may seem prudent to implement every conceivable safety measure, but this approach can be counterproductive. Excessive guardrails risk losing the intent of the user's initial request or the value of the app's output. Instead, we advise starting with the most critical guards and expanding judiciously as needed. Tools like Arize's AI search can be helpful in identifying clusters of problematic inputs to allow for targeted guard additions over time.

Types of Guards

AI guardrails span input validation and sanitization — like syntax and format checks, content filtering, jailbreak attempt detection — and output monitoring and filtering, which can prevent damage, ensure performance, or evolve over time through dynamic guards. Let's explore the main categories of guards and their applications.

Input Validation and Sanitization
Input validation and sanitization serve as the first line of defense in AI safety. These guards ensure that the data fed into your model is appropriate, safe, and in the correct format.

Syntax and Format Checks
While basic, these checks are important for maintaining system integrity. They verify that the input adheres to the expected format and structure. For instance, if your model expects certain parameters, what happens when they're missing? Consider a scenario where your RAG retriever fails to return documents, or your structured extractor pulls the wrong data. Is your model prepared to handle this malformed request? Implementing these checks helps prevent errors and ensures smooth operation.

Content Filtering
This guard type focuses on removing sensitive or inappropriate content before it reaches the model. Detecting and removing personally identifiable information can help avoid potential privacy issues, and filtering NSFW or toxic language can ensure more appropriate responses from your LLM. We recommend implementing this guard cautiously – overzealous filtering might inadvertently alter the user's original intent. Often, these types of guards are better suited filtering the outputs of your application rather than the inputs.

Jailbreak Attempt Detection
These are the guards that prevent massive security breaches and keep your company out of news headlines. Many collections of jailbreak prompts are available, and even advanced models can fail on up to 40% of these publicly-documented attacks. As these attacks constantly evolve, implementing effective guards can be challenging; we recommend using an embedding-based guard like Arize's, which can adapt to changing strategies. At minimum, use a guard connected to a common library of prompt injection prompts, such as Rebuff.

Output Monitoring and Filtering

Output guards generally fall into two categories: preventing damage, and ensuring performance.

Preventing Damage

Examples of this include:

> **System Prompt Protection:** Some attacks try to expose the prompt templates your system uses. Adding a guard to detect system prompt language in your outputs can mitigate this risk. Just be sure to avoid exposing this same template within your guard's code!

> **NSFW or Harmful Language Detection:** Allowing this type of language in your app's responses can be extremely harmful to user experience and your brand. Use guards to help identify this language.

> **Competitor Mentions:** Depending on your use case, mentioning competitors might be undesirable. Guards can be set up to filter out such references.

Ensuring Performance

When it comes to performance, developers face a choice between using guards to improve your app's output in real-time or running offline evaluations to optimize your pipeline or prompt template. Real-time guards introduce more latency and cost but offer immediate improvements. Offline evaluations allow for pipeline optimization without added latency, though there may be a delay between issue discovery and resolution. We recommend starting with offline evaluations and only adding performance guards if absolutely necessary.

> **Hallucination Prevention:** Guards can prevent hallucinations by comparing outputs with reference texts or, when unavailable, cross-referencing with reliable sources like Wikipedia.

> **Critic Guards:** This broad category involves using a separate LLM to critique and improve your pipeline's output before sending it to the user. These can be instructed to focus on relevancy, conciseness, tone, and other aspects of the response.

Dynamic Guards

Manually updating guards to counter new threats is a near impossible task, quickly becoming unsustainable as attack vectors multiply. Fortunately, two approaches allow us to create adaptive guards that can keep pace with emerging threats: few-shot prompting and embedding-based guards.

While static guards are great at filtering out predefined content like NSFW language, they struggle when faced with sophisticated attacks like jailbreak attempts, prompt injection, and more. These dynamic threats require equally dynamic defenses that can evolve alongside the attackers' strategies.

Few-Shot Prompting
This technique involves adding examples of recent jailbreak attempts or other attacks directly into your guard's prompt. By exposing the guard to real-world attack patterns, you improve its ability to recognize and thwart similar threats in the future.

This is a more sophisticated approach to dynamic protection. It involves comparing the embedding of a user's input against a database of known attack embeddings. By checking the similarity between these representations, the system can identify and block potentially malicious inputs that exceed a predefined similarity threshold. Arize has developed an easy-to-use but powerful implementation of this concept with our ArizeDatasetEmbeddings Guard.

While dynamic guards offer powerful protection, they come with a few considerations:

- Increased computational cost due to larger models or embedding generation.
- Higher latency, potentially impacting response times.
- The need for ongoing maintenance and updates to the attack prompt database.

Dynamic Guards continued

	Dataset Embeddings Guard	Few Shot LLM Guard	General LLM Guard
Advantages	**Customizable:** Customize this Guard to your specific use case by providing few shot examples from real customer chats **Easy to update:** Tackle drift by updating the Guard with new few shot examples as the models and failure modes evolve over time **Low latency / cost:** Does not rely on a large model to evaluate the input message	**Customizable:** Same as Dataset Embeddings **Easy to update:** Same as Dataset Embeddings **Fewer false positives:** Less likely to produce false positives, e.g. can differentiate between jailbreak attempts and role-play	**Customizable:** Instantiate with a custom prompt for a specific use case **Opportunity to optimize performance:** Try our prompt playground to optimize performance via prompt engineering against a golden dataset
Disadvantages	Performance depends on the quality of the source dataset LLM evaluator call introduces cost and latency	Performance depends on the quality of the source dataset LLM evaluator call introduces cost and latency	Performance depends on the quality of the prompt LLM evaluator call introduces cost and latency Prompt may need to be continually re-engineered as models and use cases evolve

It's important to weigh these factors against the level of protection required for your specific use case.

Continuous Improvement

Self-Improving Evaluations

In the world of LLM evaluation, self-improving evaluations represent an approach where models not only undergo testing but also learn from their mistakes and continuously improve their own evaluation methods. The idea behind this is to create an adaptive evaluation system that refines itself over time, leading to more accurate assessments of LLM performance.

1. CONTINUOUS LEARNING FROM ERRORS

Self-improving evals work by systematically identifying areas where the model performs poorly and using that feedback to adjust future evaluations. For example, if a model frequently makes errors in answering complex questions, the evaluation framework can highlight those areas and provide focused feedback to guide future adjustments.

This approach allows the evaluation framework to learn and evolve, enabling:

> **Identification of weak points:** Pinpointing tasks or datasets where the model struggles the most.

> **Dynamic refinement:** The evaluation framework adjusts its criteria or datasets, incorporating harder examples or edge cases.

> **Self-correction:** Over time, the system improves by updating its metrics or test cases based on the model's performance, continuously raising the standard for evaluation.

2. FEEDBACK LOOPS

A crucial aspect of self-improving evals is the use of feedback loops. These loops allow models to learn from the evaluations themselves. Feedback on where the model underperforms can guide fine-tuning or retraining efforts, leading to more effective performance improvements.

By embedding feedback loops into the evaluation process, you can ensure that models become more robust over time. This is particularly important for models that will encounter changing environments or evolving datasets, as they will need to adapt in real time.

Improving LLM Evaluation Systems

Improving LLM evaluation systems is usually done by either iterating on the prompt template used, adding few-shot examples, or by fine-tuning the underlying eval model.

FEW-SHOT PROMPTING EVALUATIONS

Few-shot prompting refers to the practice of providing an LLM with a small number of examples (or shots) to help it understand the task at hand. In few-shot prompting evaluations, the goal is to see how well the model can generalize and perform based on very limited input data.

1. Setting Up Few-Shot Prompting Evals

In this evaluation method, the model is given a few labeled examples as part of the prompt to guide its response. For example, in evaluating a summarization model, you might provide two or three examples of text along with their summaries before asking the model to generate a summary for a new text. The evaluation then measures how well the model learns from these few examples to generate accurate outputs for unseen data.

Few-shot prompting evaluations simulate real-world use cases where large training datasets are unavailable. This is crucial for:

- **Rapid prototyping:** Evaluating how a model handles new tasks with minimal input.
- **Generalization:** Testing the model's ability to generalize from very little data.

2. Curating a Dataset for Few-Shot Evaluations

Curating a dataset for few-shot evaluation is about finding the right balance between variety and representativeness. Unlike larger datasets, where quantity might mask inconsistencies, few-shot datasets must be carefully selected to represent the full scope of the task. You can do so by either manually curating them or hand writing examples, or use an LLM to synthetically generate them.

```
You are a data analyst. You are using LLMs to summarize a document.
Create a CSV of 20 test cases with the following columns:

1. Input: The full document text, usually five paragraphs of articles
about beauty products.

2. Prompt Variables: A JSON string of metadata attached to the
article, such as the article title, date, and website URL

3. Output: The one line summary
```

Steps for curating an effective few-shot dataset:

> **Diverse examples:** Include a range of inputs that cover different types of challenges the model may face. This ensures that the few examples are representative of broader tasks.

> **Edge cases:** Incorporate examples that test the model's limits, including rare or tricky scenarios that might confuse it.

> **Consistency in ground truth:** Ensure that the correct answers (ground truth) provided for each example are consistent and well-defined, as the model will be evaluated on how well it follows these examples.

By carefully curating a few-shot dataset, you allow the model to demonstrate its ability to quickly learn and adapt to new information, which is a critical component in measuring its overall robustness.

3. Updating Prompts with Examples

How do you determine which examples to add to your prompt? You can select your examples based on the cosine similarity between each of the given prompt variables. You can add examples of queries that are very similar to the user query to increase your precision.

You can also use an LLM to summarize the examples and insert them as additional instructions. As you add additional examples, you can start catching more and more edge cases, add them to your prompt, and re-test them against your golden dataset to ensure reliability.

```
In this task, you will be presented with a query, a reference text
and an answer. The answer is generated to the question based on the
reference text. The answer may contain false information. You must
use the reference text to determine if the answer to the question
contains false information, if the answer is a hallucination of facts.
A 'hallucination' refers to
an answer that is not based on the reference text or assumes
information that is not available in
the reference text. Your response should be a single word: either
"factual" or "hallucinated", and
it should not include any other text or characters. "hallucinated"
indicates that the answer
provides factually inaccurate information to the query based on the
reference text. "factual"
indicates that the answer to the question is correct relative to the
reference text, and does not contain made up information.

Use the examples below for reference:
{examples}

Here is the query, reference, and answer.
   # Query: {query}
   # Reference text: {reference}
   # Answer: {response}

Is the answer above factual or hallucinated based on the query and
reference text?
```

4. Fine tuning the evaluation model
The last step is fine-tuning the evaluator. Fine tuning the evaluator model is similar to fine tuning the LLM used for the application, and can be done using the data points collected earlier.

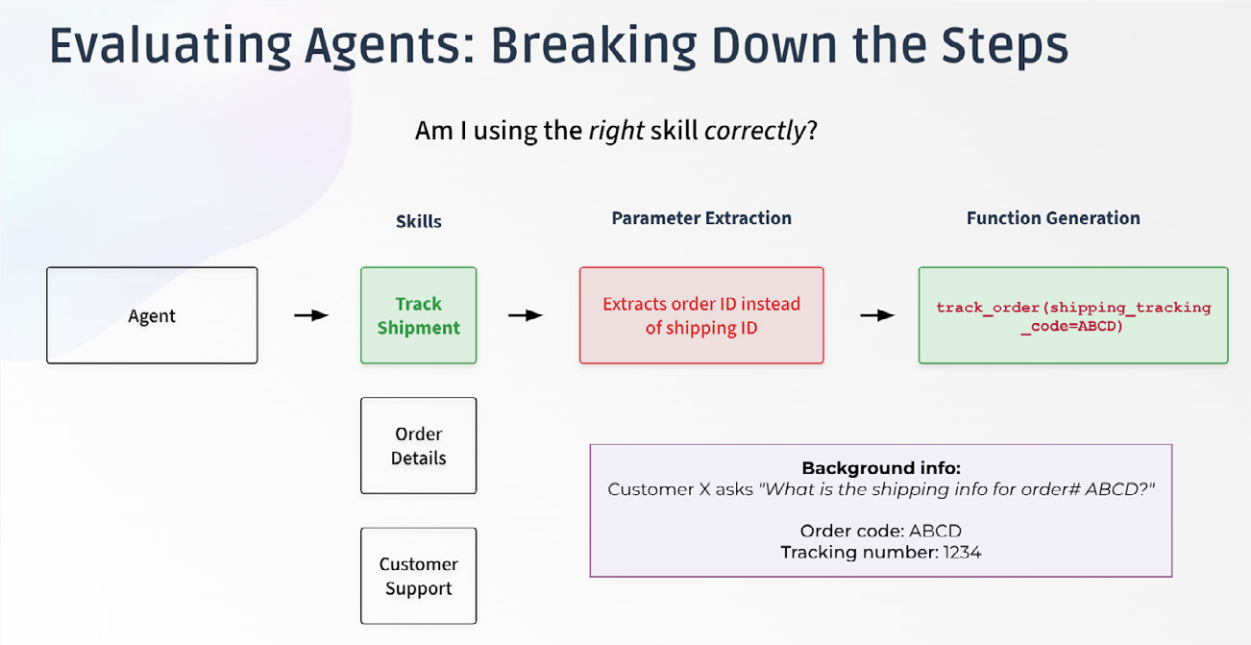
This also allows teams to use smaller language models, which reduces latency and cost while maintaining similar levels of performance. As the dataset of corrections increase, AI engineers can connect their evaluator to the CI/CD pipeline and continuously run fine tuning jobs to increase the precision of their evaluator.

Use Cases

Evaluating Agents

BUILDING EVALUATORS FOR EACH STEP

Evaluating the skill steps of an agent is similar to evaluating those skills outside of the agent. If your agent has a RAG skill, for example, you would still evaluate both the retrieval and response generation steps, calculating metrics like document relevance and hallucinations in the response.



UNIQUE CONSIDERATIONS WHEN EVALUATING AGENTS

Beyond skills, agent evaluation becomes more unique.

In addition to evaluating the agent's skills, you need to evaluate the router and the path the agent takes.

The router should be evaluated on two axes: first, its ability to choose the right skill or function for a given input; second, its ability to extract the right parameters from the input to populate the function call.

Choosing the right skill is perhaps the most important task and one of the most difficult. This is where your router prompt (if you have one) will be put to the test. Low scores at this stage usually stem from a poor router prompt or unclear function descriptions, both of which are challenging to improve.

Extracting the right parameters is also tricky, especially when parameters overlap. Consider adding some curveballs into your test cases, like a user asking for an order status while providing a shipping tracking number, to stress-test your agent.

Arize provides built-in evaluators to measure tool call accuracy using an LLM as a judge, which can assist at this stage.

```
TOOL_CALLING_PROMPT_TEMPLATE = """
You are an evaluation assistant evaluating questions and tool calls to
determine whether the tool called would answer the question. The tool
calls have been generated by a separate agent, and chosen from the
list of tools provided below. It is your job to decide whether that
agent chose the right tool to call.

    [BEGIN DATA]
    ************
    [Question]: {question}
    ************
    [Tool Called]: {tool_call}
    [END DATA]

Your response must be single word, either "correct" or "incorrect",
and should not contain any text or characters aside from that word.
"incorrect" means that the chosen tool would not answer the question,
the tool includes information that is not presented in the question,
or that the tool signature includes parameter values that don't match
the formats specified in the tool signatures below.

"correct" means the correct tool call was chosen, the correct
parameters were extracted from the question, the tool call generated
is runnable and correct, and that no outside information not present
in the question was used in the generated question.

    [Tool Definitions]: {tool_definitions}
"""
```

Lastly, evaluate the path the agent takes during execution. Does it repeat steps? Get stuck in loops? Return to the router unnecessarily? These “path errors” can cause the worst bugs in agents.

To evaluate the path, we recommend adding an iteration counter as an evaluation. Tracking the number of steps it takes for the agent to complete different types of queries can provide a useful statistic.

However, the best way to debug agent paths is by manually inspecting traces. Especially early in development, using an observability platform and manually reviewing agent executions will provide valuable insights for improvement.

Retrieval Augmented Generation (RAG) Evaluation

RAG applications need to be evaluated on two critical aspects.

- **Retrieval Evaluation:** To assess the accuracy and relevance of the documents that were retrieved. Examples:

Groundedness or Faithfulness	The extent or faithfulness to which the LLM's response aligns with the retrieved context.	Binary classification (fiathful/unfaithful)
Context relevance	Guages how relevant the retirieved context supports the user's query.	Binary classification (fiathful/unfaithful). Ranking metrics: Mean Reciprocal Rank (MRR), Precision @ K, Mean Average Precision (MAP), Hit rate, Normalized Dicounted Cumulative Gain (NDCG)

```
You are comparing a reference text to a question and trying to
determine if the reference text contains information relevant to
answering the question. Here is the data:

   [BEGIN DATA]
   ************
   [Question]: {query}
   ************
   [Reference text]: {reference}
   [END DATA]

Compare the Question above to the Reference text. You must determine
whether the Reference text contains information that can answer the
Question. Please focus on whether the very specific question can be
answered by the information in the Reference text.

Your response must be single word, either "relevant" or "unrelated",
and should not contain any text or characters aside from that word.

"unrelated" means that the reference text does not contain an answer to
the Question.

"relevant" means the reference text contains an answer to the Question.
```

- **Response Evaluation:** To measure the appropriateness of the response generated by the system when the context was provided. Examples:

Ground Truth-Based Metrics	The extent or faithfulness to which the LLM's response aligns with the retrieved context.	Accuracy, Precision, Recall, F1 score
Answer Relevance	Guages how relevant the retirieved context supports the user's query.	Binary classification (Relevant/Irrelevant
QA Correctness	Detectsw hether a question was correctlly answered by the system based on the retrieved data.	Binary classification (Correct/Incorrect)
Hallucinations	To detect LLM hallucinations relative to retrieved context.	Binary classification (Factual/ Hallucinated)

```
You are given a question, an answer and reference text. You must
determine whether the given answer correctly answers the question
based on the reference text. Here is the data:

   [BEGIN DATA]
   ************
   [Question]: {question}
   ************
   [Reference]: {context}
   ************
   [Answer]: {sampled_answer}
   [END DATA]

Your response must be a single word, either "correct" or
"incorrect", and should not contain any text or characters aside
from that word.

"correct" means that the question is correctly and fully answered
by the answer.

"incorrect" means that the question is not correctly or only
partially answered by the answer.
```

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 Community

Product Documentation

arize

To start your AI Observability and Evaluation journey, **sign up for a free trial** or **schedule a demo.**

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