LangChain

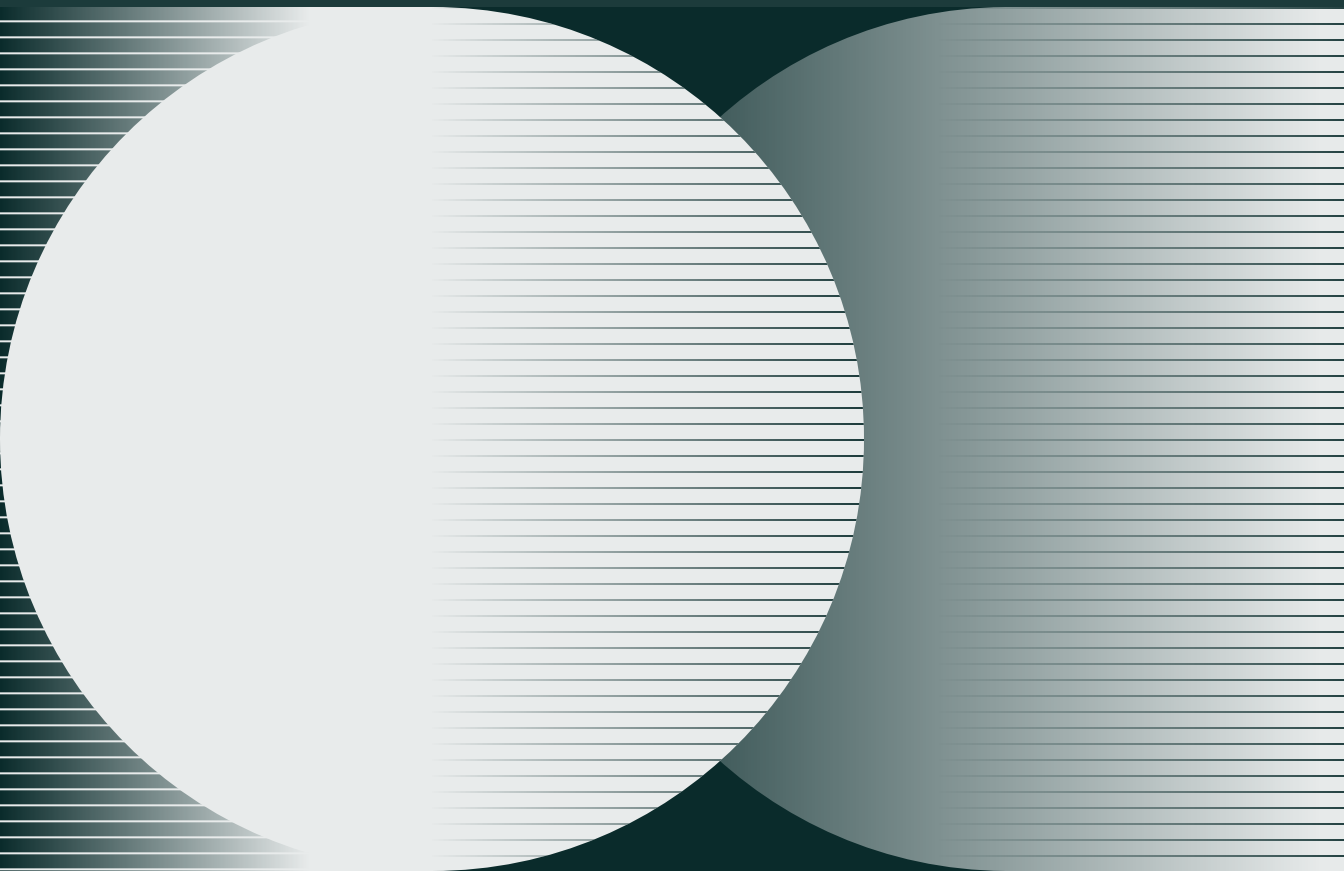
The Definitive Guide to Testing LLM Applications

Table of Contents

LangChain

INTRODUCTION

Why testing matters for LLM applications



Why testing matters for LLM applications

While Large Language Models (LLMs) promise to solve previously unthinkable tasks, they have also introduced new challenges for successful deployment. First, LLMs are non-deterministic, generating a distribution of possible outcomes from a single input. This can lead to inconsistent or hallucinated outputs. Additionally, LLMs ingest arbitrary text, forcing developers to grapple with a broad domain of possible user inputs. And, finally, LLMs output natural language, often requiring new metrics to assess style or accuracy.

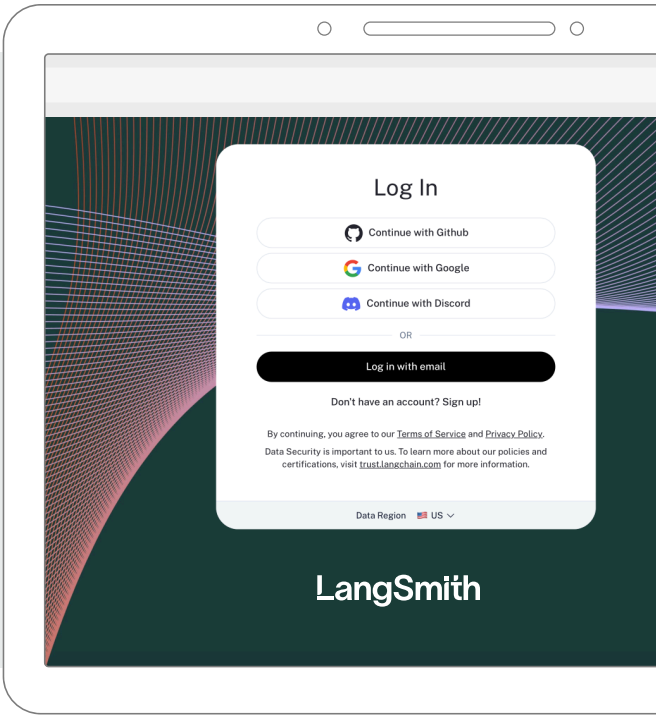
With these challenges in mind, we find that companies must develop new testing approaches to guard against harmful / misleading responses or even embarrassing brand moments caused by their LLM app.

Follow along with LangSmith

Throughout this guide, we'll refer to LangSmith, our platform for testing and monitoring LLM applications.

If you'd like to follow along on your own account, sign up for free at

smith.langchain.com



At LangChain, we think of testing as the way to measure your LLM application's performance. The goal of testing is to help you iterate faster on your LLM app, enabling you to make quick decisions amidst a vast sea of choices – including what models, prompts, tools, or retrieval strategies to use.

We've crafted this short guide to help you add rigor to your LLM app testing. In this guide, you'll learn various testing techniques across phases of the LLM app development lifecycle — from application design, to pre-production, to post-production. We also give advice on evaluating specific use cases such as Retrieval Augmented Generation (RAG) applications and agents.

Before LangSmith, we didn't have a systematic way to improve the quality of our LLM applications. By integrating LangSmith into our application framework, we now have a cohesive approach to benchmark prompts and models for 200+ of our applications. This supports our data-driven culture at Grab and allows us to drive continuous refinement of our LLM-powered solutions.

Padarn Wilson
Head of Engineering, ML Platform at Grab

LangChain

Testing techniques across the app development cycle

1.0

Tests can be applied at three different stages in your application development cycle:

Design

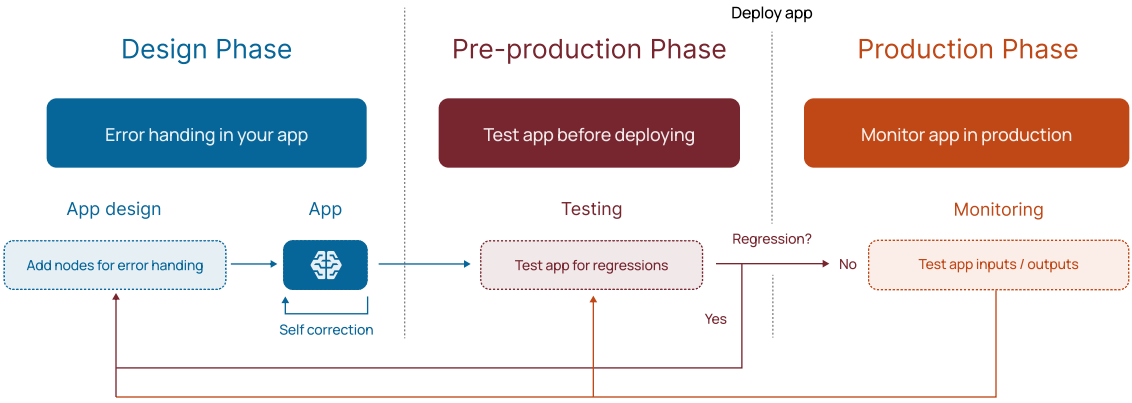
Tests can be added directly to your application. These are typically simple assertions fast enough to be executed at runtime. Using an orchestrator system can help execute these tests and feed failures back to the LLM. The goal is to catch and self-correct errors *within your app* itself before they affect users.

Pre-Production

Tests can be run before deploying your application into production. These typically cover key scenarios your application must pass, based on data you've collected. The goal is to catch and fix any regressions before the app is released to real users.

Post-Production

Tests can be run on your application in production. These help monitor and catch errors or undesirable behavior affecting real users. The goal is to identify issues and feed them back into the design or pre-production phases, creating a continuous cycle of "design, test, deploy, monitor, fix, and redesign".



Together, these testing phases form a flywheel for continuous improvement of your LLM system, helping you identify and fix production issues to prevent regressions in new versions of your app. Now, let's dive into techniques and tips for each phase.

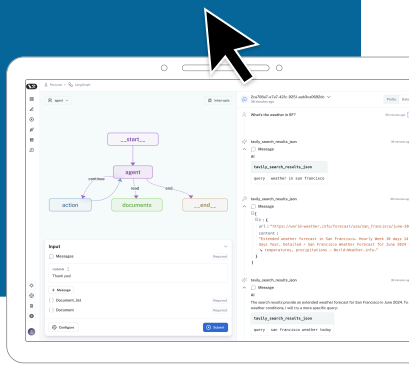
LangChain

Design Phase 2.0

When designing your application, you may want to consider adding checks within your application logic to prevent unwanted outputs. Applications with built-in error handling can leverage the ability for LLMs to self-correct by executing tests within the application and feeding errors back to the LLM.

Below we discuss a few design patterns that can be used for self-correction. In each case discussed below, we use LangGraph as the framework to orchestrate error handling.

To learn more about LangGraph, check out our documentation and tutorials.



Use Case:

Self-corrective code generation

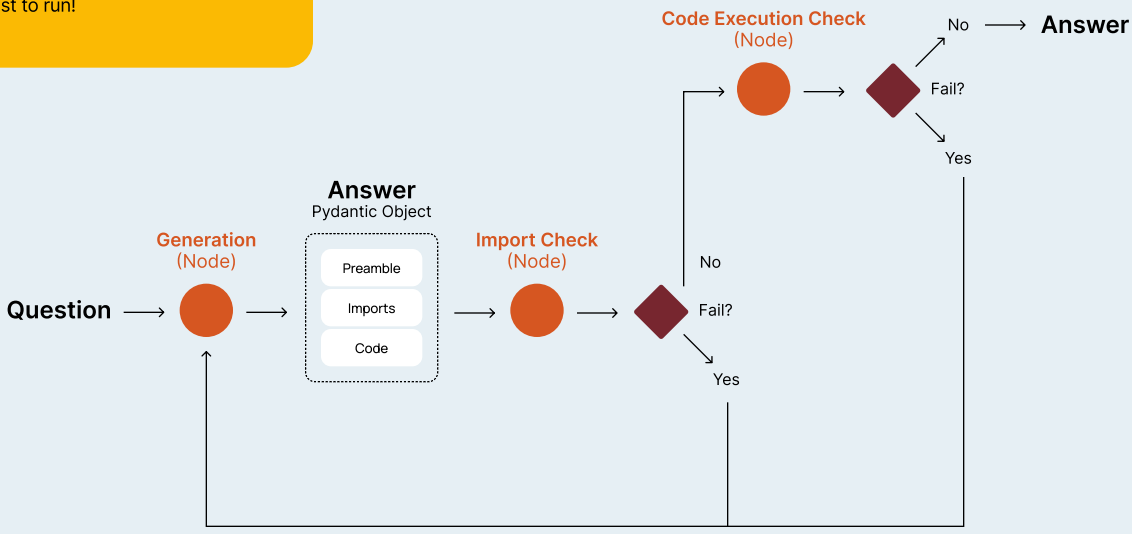
In traditional systems, incorrect or incomplete code often arises from misunderstandings or lack of context, such as incorrect import statements or syntax mistakes. To identify and correct these errors, you would typically manually review code, analyze error messages, correct the issues, and re-run tests until the code functions correctly.

Encoding these checks into a system like LangGraph and having it make fixes can greatly speed up this process of code generation. Error handling should ideally focus on lightweight, deterministic checks—such as simple assertions or Guardrails—to minimize cost and latency, while LangGraph can orchestrate retries or remediation when those checks fail. In this example, ChatLangChain can sometimes hallucinate import statements, which degrades the user experience. This approach reduced those failures on our evaluation set, but does not eliminate them entirely.

To address this, we can design a simple LangGraph control flow that tests the imports and, if they fail, passes back any errors to the LLM for correction. This simple assertion improved performance considerably on our evaluation set.

Tip

Always start with the simplest possible tests (e.g., assertions that can be hard-coded) because they are cheap and fast to run!



Use Case:

Self-corrective RAG

Sometimes, simple assertions are not a sufficient test. For example, RAG systems can suffer from hallucinations and low-quality retrieval (e.g., if a user's question falls outside the domain of the indexed documents).

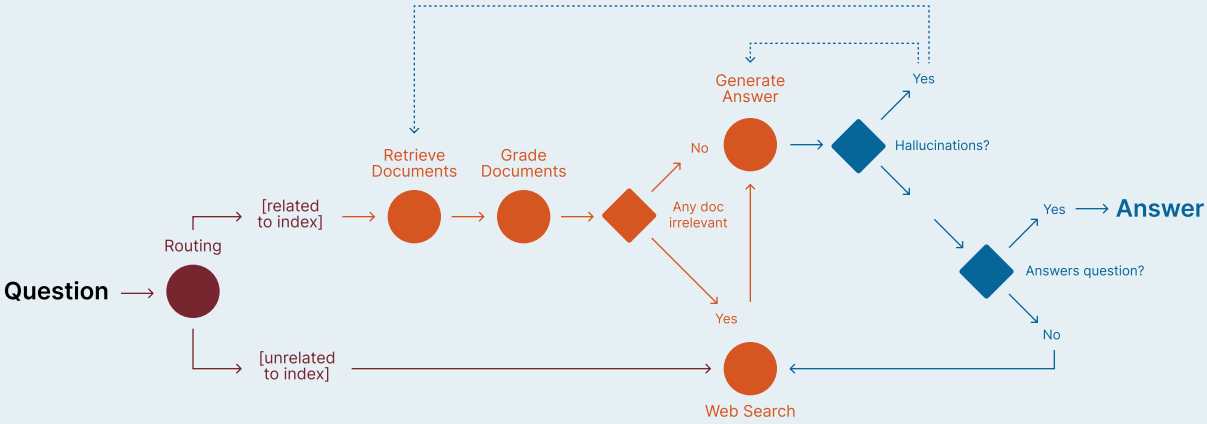
In both cases, tests that can reason over text are needed and cannot be easily captured as simple assertions. In these cases, we can use an LLM to perform the testing.

> **Tip**
>
> Use of LLM-as-judge evaluators for error handling should be done with care to minimize latency and cost. See the "Pre-Production" section for more on LLM-as-judge evaluators.

As an example, here is a self-corrective RAG application with several stages of error handling that uses an LLM to grade retrieval relevance, answer hallucination, and answer usefulness.

The application control flow is laid out explicitly using LangGraph:

- Each node should be assigned a specific task.
- The control flow performs assertions in logical order, starting with retrieval.
- If the retrieval documents are relevant, it then proceeds with answer generation.
- It checks if the answer contains hallucinations related to the retrieved documents.
- Finally, it checks whether the answer addresses the questions.



Using LangGraph, this control flow is highly reliable. We've even run it using strictly local LLMs (e.g., 8b parameter model).

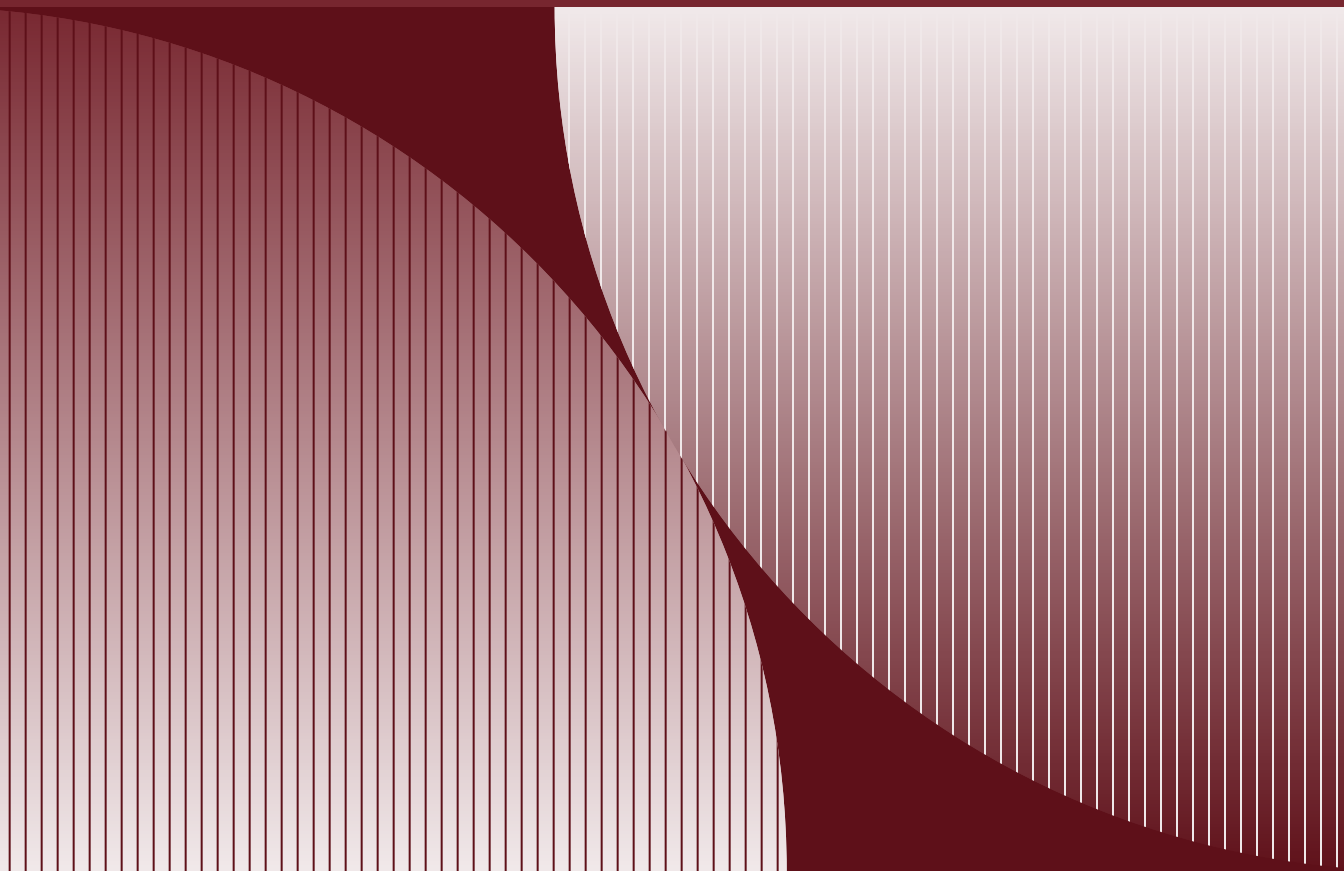
To see step-by-step how to implement corrective RAG, check out these resources:

Building reliable, fully-local RAG agents with Llama3

Conceptual guide on self-corrective generation in RAG

LangChain

Pre-Production 3.0



Pre-Production

The goal of pre-production testing is to measure the performance of your application, ensuring continuous improvement and catching any regressions on scenarios that you expect to pass. To begin testing, you'll first need to define a set of scenarios that you want to test. Below, we'll walk through prepping a dataset and evaluation criteria, and how to then use regression testing to measure and improve LLM app performance over time.

Prerequisites:

Build a dataset

Datasets are collections of examples that serve as inputs and (optionally) expected outputs used to evaluate your LLM app.

Gathering good data is often the hardest part of testing, given the large amount of time and context needed to pull together a quality dataset to benchmark your system. However, you don't need many data points to get started.

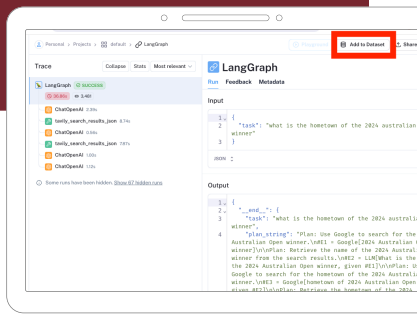
Below, we outline common methods for gathering data:

- **Manually curated examples:** These are hand-chosen / written examples. We often see teams start small with their volume of data for a new project, collecting 10-50 quality examples by hand. You can add to these manual data points over time to cover edge cases that emerge during production usage.

- **Traces:** These are traces of past interactions with your application. Adding in production traces from older app versions ensures your dataset is realistic and can cover commonly-recurring user questions as you iterate on your application.

- **Synthetic data:** These are artificially-generated examples that simulate various scenarios and / or edge cases. This approach can be useful for augmenting your dataset when you may not have enough real data to test on. You can generate new, plausible inputs by sampling from existing inputs — or you can paraphrase existing inputs to diversify your dataset without changing the underlying context.

If you're using LangSmith, it's easy to save debugging and production traces to datasets, or to go back and edit your dataset as you gather more information.



Define evaluation criteria

After creating your dataset, you'll want to define evaluation metrics to assess your application's responses before shipping to production. This batch evaluation on a predetermined test suite is often referred to as **Offline Evaluation**.

For offline evaluation, you can also optionally label expected outputs (i.e. ground truth references) for the data points you are testing on. This lets you compare your application's response with the ground truth references.

There are a few ways to score your LLM app performance:

- **Heuristic evaluators:** These allow you to define assertions and hard-coded rules on your outputs to score their quality. You can use reference-free heuristics (e.g. checking if output is valid JSON) or reference-based heuristics like accuracy. Reference-based evaluation compares an output to a predefined ground truth, whereas reference-free evaluation assesses qualitative characteristics without a ground truth.

  Custom heuristic evaluators are useful for code generation tasks like schema checking and unit testing with hard-coded evaluation logic; in contrast, evaluations on natural language cannot be hardcoded as rules, requiring human or LLM-as-Judge evaluators.

Tip

Start with simple (e.g. heuristic) evaluations as much as possible. Then, perform human review. Finally, use LLM-as-Judge to automate your human review. This order of operations lets you add depth and scale once your criteria are well-defined.

- **Human evaluators:** Using human feedback is a good starting point if you can't express your testing requirements as code and an LLM evaluator is not consistent enough. When looking at qualitative characteristics, humans can label app responses with scores. LangSmith speeds up the process of collecting and incorporating human feedback with annotation queues.

- **LLM-as-Judge evaluators:** These capture human grading rules into an LLM prompt, in which you use the LLM to judge whether the output is correct (e.g. relative to the reference answer) or whether it adheres to specific criteria (e.g. if it's reference-free and you'd like to check if the output contains offensive content). Answer correctness is a common LLM-as-Judge evaluator for offline evaluation, in which the reference is a correct answer supplied from the dataset output. As you iterate in pre-production, you'll want to audit the scores and tune the LLM-as-Judge to produce reliable scores.

Tip

For LLM-as-Judge evaluators, you can:

1. Minimize cognitive load by using binary (yes, no) or simple (1, 0) scores versus a continuous or more complex (e.g., Likert scale) possible range of scores.
2. Use straightforward prompts that can easily be replicated and understood by a human. For example, avoid asking an LLM to produce scores on a range of 0-10 with vague distinctions between scores.

Advanced:

Few-shot feedback for improving LLM-as-Judge evaluators

How can you trust the results of LLM-as-Judge evaluation? Typically, this requires another round of prompt engineering to ensure accurate performance. But by leveraging few-shot learning, “self-improving” evaluation is now possible in LangSmith with minimal setup.

Few-shot learning can improve accuracy and align outputs with human preferences by providing examples of correct behavior. This is especially useful when it’s tough to explain in instructions how the LLM should behave, or if the output is expected to have a particular format. In LangSmith, self-improving evaluation looks like the following:

1. The LLM evaluator provides feedback on generated outputs, assessing factors like correctness, relevance, or other criteria
2. Add in human corrections to modify or correct the LLM evaluator’s feedback in LangSmith. This is where human preferences and judgment are captured.
3. These corrections are stored as few-shot examples in LangSmith, with an option to leave explanations for corrections.
4. The few-shot examples are incorporated into future prompts as subsequent evaluation runs.

Over time, the LLM-as-Judge evaluator becomes increasingly aligned with human preferences. This self-improving approach eliminates the need for time-consuming prompt engineering, while improving the accuracy and relevance of LLM-as-Judge evaluations. To learn more, read this blog post.

Advanced:

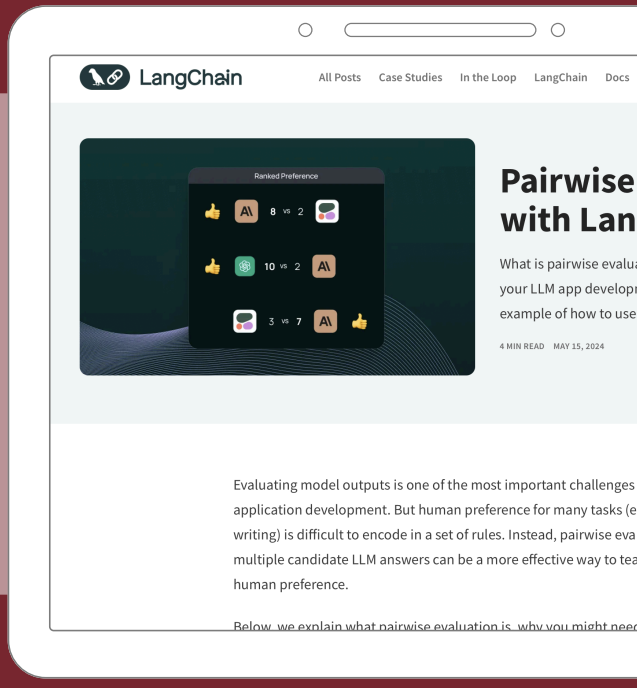
Pairwise evaluation

Evaluating LLM app outputs can be challenging in isolation, especially for tasks where human preference is hard to encode in a set of rules. Ranking outputs preferentially (for example, which one of these two outputs is more informative, vague, safe etc, versus grading each one individually) can be less cognitively-demanding for human or LLM-as-Judge evaluators.

This is where pairwise evaluation comes in handy. Pairwise evaluation compares two outputs simultaneously from different versions of an application to determine which version better meets evaluation criteria like creativity, etc.

LangSmith natively supports running and visualizing pairwise LLM app generations, highlighting preference for one generation over another based on guidelines set by the pairwise evaluator.

Read more in this blog post

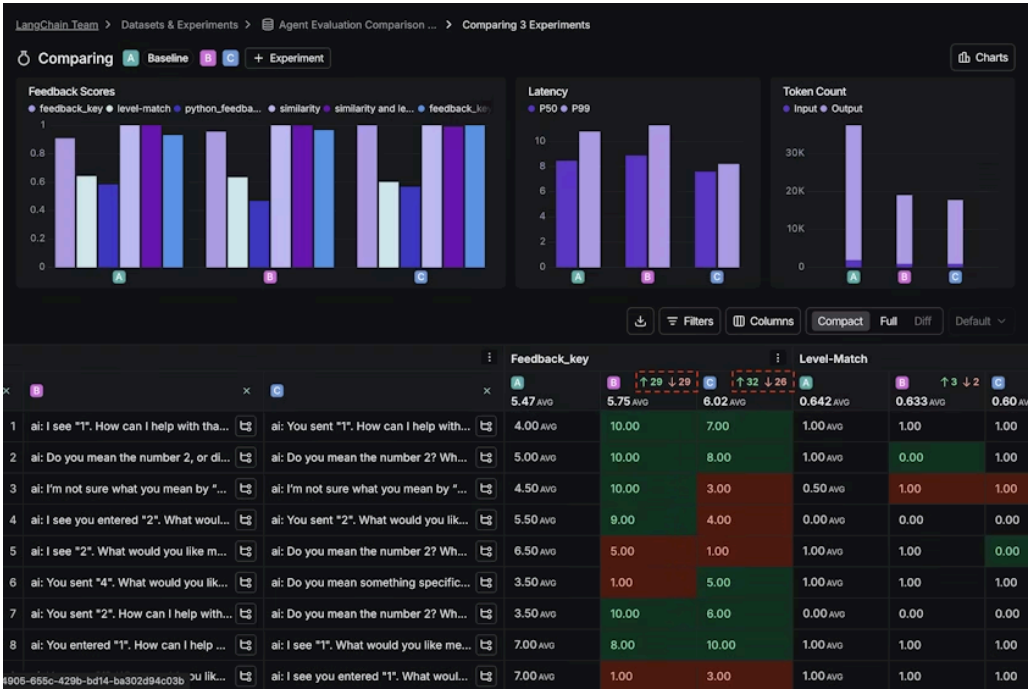


Regression testing

In traditional software, tests pass based on functional requirements, leading to more stable behavior once validated. In contrast, AI models show variable performance due to model drift (i.e. degradation due to changes in data distribution or updates to the model) and sensitivity. As such, regression testing for LLM applications is even more critical and should be done frequently.

Once you've defined a dataset and evaluator, you may want to:
(1) Measure your LLM application's performance across experiments to identify improved app versions to ship.
(2) Monitor app performance over time to prevent regressions.

LangSmith supports these needs with a built-in comparison view, as shown below.



For (1), you can select and compare experiments associated with a dataset. Runs that regressed are auto-highlighted in red, while runs that improved are in green — showing aggregate stats and allowing you to comb through specific examples. This is useful for migrating models or prompts, which may result in performance improvements or regression on specific examples.

For (2), you can set a baseline (e.g., the current deployed version of your app) and compare it against prior versions to detect unexpected regressions. If a regression occurs, you can isolate both the app version and the specific examples that contain performance changes.

LangSmith has made it easier than ever to curate and maintain high-signal LLM testing suites. Using LangSmith for testing, we've seen:

- *A 43% performance increase over production systems, bolstering executive confidence to invest millions in new opportunities.*
- *A 15% reduction in engineering time needed for regression testing, by eliminating the "whack-a-mole" effect of prompt changes*

Walker Ward
Software Engineer Architect at Podium

Use Case:

Testing agents

Agents show promise for autonomously performing tasks and automating workflows, but testing an agent can be challenging. Agents use an LLM to decide the control flow of the application, which means every agent run can look quite different. For example, different tools might be called, agents might get stuck in a loop, or the number of steps from start to finish can vary significantly.

Tip

For building agents, you can:

- Customize your agent: Build custom or domain-specific agents with a controllable agent orchestrator like LangGraph. This improves agent reliability over general purpose architectures.
- Diversify LLMs: Test multiple tool-calling LLMs to optimize performance and manage costs effectively, leveraging different strengths for specific applications.

For testing agents, you can:

- Reduce noise: Implement repetitions (i.e. run same test multiple times and aggregate the results) to reduce variability in tool selection and agent responses
- Isolate failures: Partition your dataset into different splits to identify specific subsets of data causing an issue. This also helps you save computational resources.

We recommend you test agents at three different levels of granularity:

- The agent's final response to strictly focus on end-to-end performance
- Any single, important step of the agent to drill into specific tool calls / decisions
- The trajectory of the agent to examine the full reasoning trajectory

To learn more about testing agents, check out these tutorials or explore this notebook.
You can also watch our workshop on building and testing reliable agents.

Testing an agent's final response

To assess the overall performance of an agent on a task, treat it as a black box and define success as whether or not it completes the task. Keep in mind that this method is hard to debug when failures occur.

Testing for the agent's final response involves:

- **Inputs:** User input and (optionally) predefined tools.
- **Output:** Agent's final response.
- **Evaluator:** LLM-as-Judge evaluators, which can assess task completion directly from a text response

Testing a single step of an agent

Testing an individual action (where the LLM call makes a decision) lets you zoom in to see where your application is failing. It can also be faster to run (with just one LLM call invoked). Keep in mind that testing a single step doesn't capture the scope of the full agent. Dataset creation may also be more difficult- as it's easier to generate a dataset for earlier steps in an agent trajectory, but harder to do so for later steps which must account for prior agent actions and responses.

Testing for a single step of an agent involves:

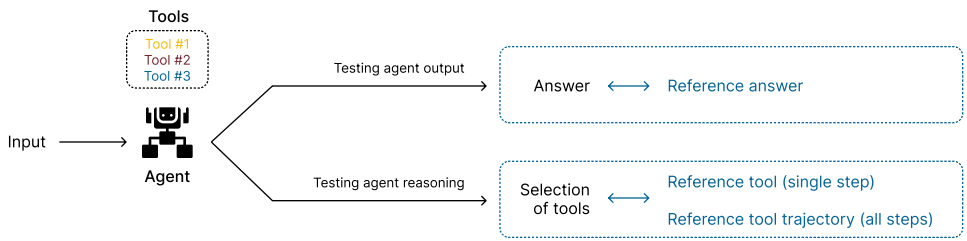
- **Inputs:** User input to a single step (e.g., user prompt, set of tools). Can also include previously completed steps.
- **Output:** LLM response from that step (often contains tool calls indicating what action the agent should take next)
- **Evaluator:** Binary score for correct tool selection and heuristic assessment of the tool input's accuracy.

Testing an agent's trajectory

Looking back on the steps an agent took (often referred to as the trajectory) lets you assess whether or not the trajectory lined up with expectations of the agent, e.g. the number of steps or sequence of steps taken.

Testing an agent's trajectory involves:

- **Inputs:** User input and (optionally) predefined tools.
- **Output:** Expected sequence of tool calls (in other words, the "exact" trajectory), or a list of tool calls in any order.
- **Evaluator:** Function over the steps taken. To test the outputs, you can look at an exact match binary score (note: may be multiple correct paths) or metrics that focus on the incorrect steps count. To test inputs to the tools, LLM-as-Judge may be more fruitful. You'd need to evaluate the full agent's trajectory against a reference trajectory, then compile as a set of messages to pass into the LLM-as-Judge.



Implementation:

Integrating into your CI flow

Integrating your LLM app testing into your Continuous Integration (CI) flow can help you automate testing each time changes are made to the codebase. But, there are a couple challenges with this workflow:

1. Tests on LLM apps tend to be flakey, as many tests also use an LLM to do the evaluation.
2. Running an experiment on every PR can be costly, as calls to LLMs are expensive.

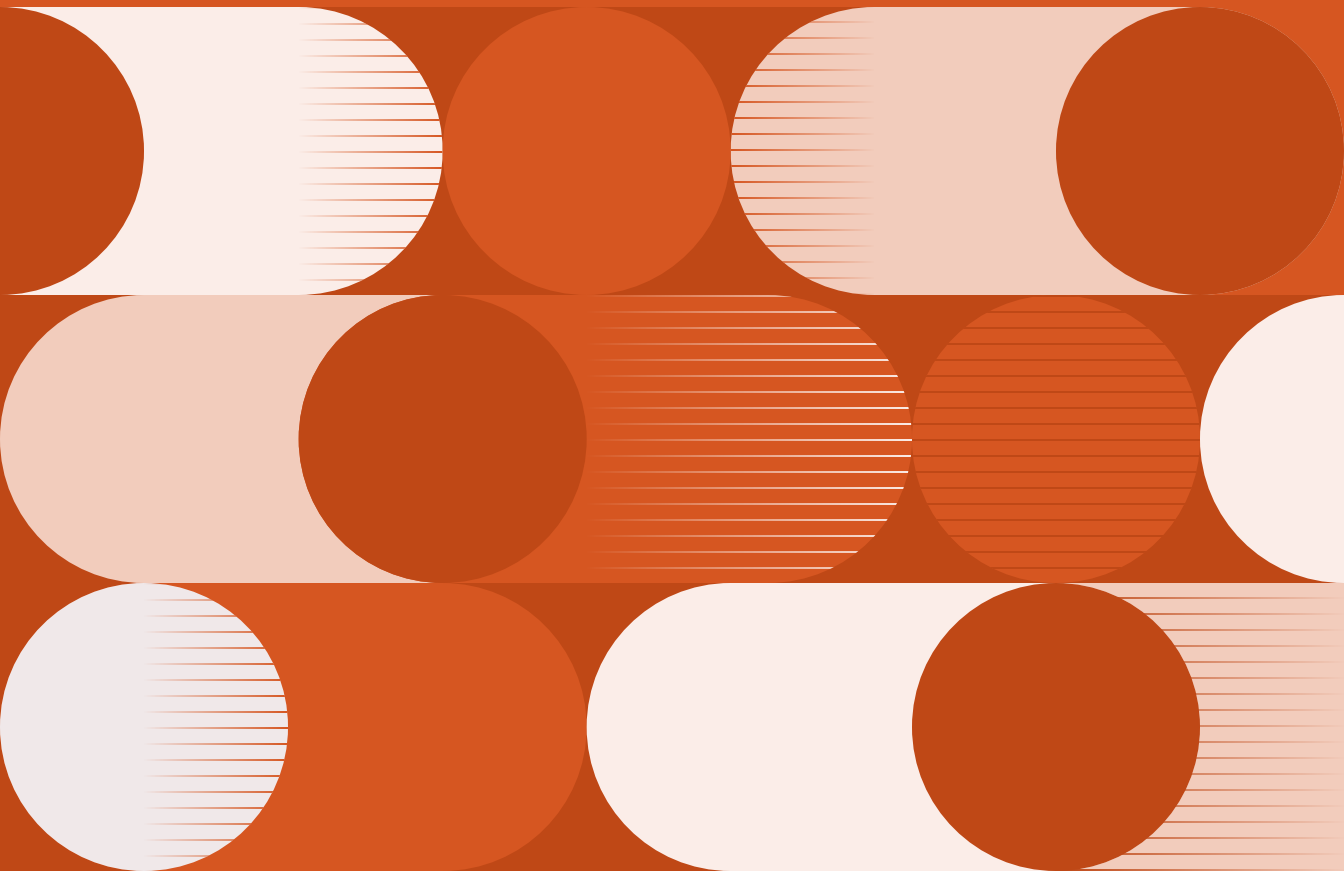
To adapt your CI workflow to deal with LLM application testing, we recommend the following:

- **Use a cache:** Instead of making a request to the LLM every time, pull from a cache if the input to the LLM hasn't changed from what's stored in the cache.
- **Isolate datasets for CI:** Instead of triggering your full experiment on each commit push, use a subset of the dataset that tests the most critical examples. Reserve running the experiment over the full dataset when you have more substantial changes.
- **Plan for human assistance:** Despite a desire for full automation, you'll likely need a workflow that allows a human to correct failing tests in order to avoid blocking merges on finicky LLM evaluators.

LangChain

Post-Production

4.0



Post-Production

Though essential, testing in the pre-production phase won't catch everything. Only after you've shipped to production can you get insights into how your LLM application fares under real user scenarios. Beyond checking for spikes in latency or errors, you'll need to assess characteristics such as relevance, accuracy, or toxicity. In post-production, a monitoring system can help detect when your LLM app performance veers off course, allowing you to isolate valuable failure cases.

Prerequisites:

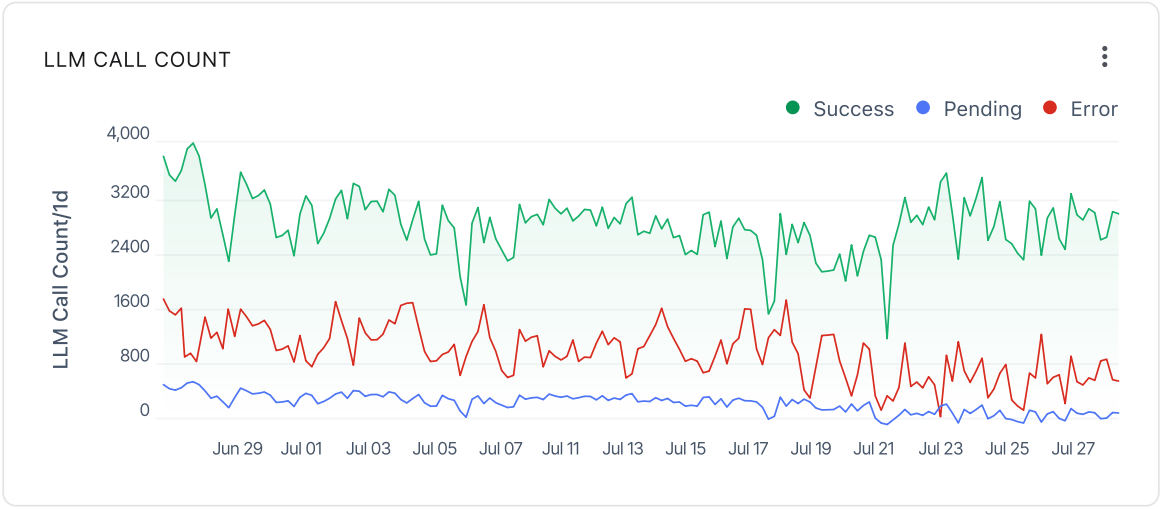
Set up tracing

If you haven't yet, you'll need to set up tracing to gain visibility into a meaningful portion of your production traffic. LangSmith makes this easy to do and offers insights into not only the user input and application response, but also all interim steps the application took to arrive at the response. This level of detail is helpful for writing specific step assertions or debugging issues later on.

LangSmith will additionally provide helpful metrics out-of-the-box, such as:

- Trace volume
- Success / failure rates
- Latency & time to first token
- LLM calls per trace
- Token count & cost

Check out our quickstart guide to set up tracing in LangSmith within minutes.



This can provide baseline insight into your LLM app performance, on top of which you'll track qualitative metrics (covered in the next section).

> *LangSmith has been instrumental in accelerating our AI adoption and enhancing our ability to identify and resolve issues that impact application reliability. With LangSmith, we've created custom feedback loops, improving our AI application accuracy by 40% and reducing deployment time by 50%.*

Varadarajan Srinivasan
VP of Data Science, AI and ML Engineering at Acxiom

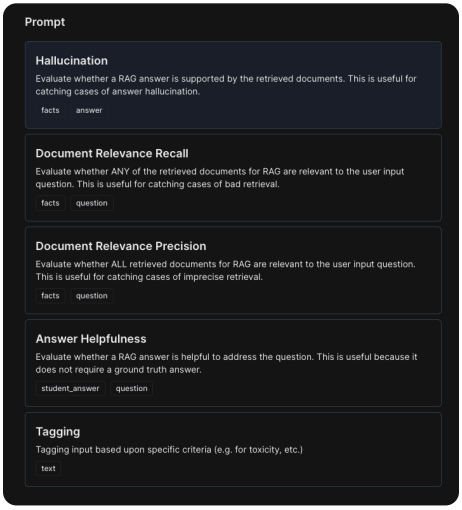
Collect feedback in production

Unlike in the pre-production phase, evaluators for post-production testing don't have grounded reference responses to compare against. Instead, your evaluators will score performance in real time as your application processes user inputs. This reference-free, real-time evaluation is often referred to as **Online Evaluation**.

There are at least two types of feedback you can collect in production to improve app performance:

- **Feedback from users:** You can directly collect user feedback, which can be explicit or implicit. Adding a 👍/ 👎 button on your application is an easy way to record user satisfaction with the application's response. You can also ask users to provide an additional explanation of why or why not their expectations were met. In LangSmith, you can attach user feedback to any trace or intermediate run (i.e. span) of a trace, including annotating traces inline or reviewing runs together in an annotation queue.

- **Feedback from LLM-as-Judge evaluators:** These can be appended to projects in LangSmith, giving you the ability to define LLM-as-judge evaluation prompts that operate directly on application inputs or outputs. LangSmith has a number of preexisting prompts for RAG as well as tagging (e.g., for toxicity).



Off-the-shelf online evaluator prompts in LangSmith

Use Case:

Evaluating a RAG application in production

Let's apply some of these concepts by adding online evaluation to a RAG application that handles common questions over a knowledge base. Typically, LLM-as-Judge evaluators are used for RAG to evaluate factual accuracy and consistency between texts.

To assess the RAG app's performance in real-time, you'll likely want to test:

1. If the application is hallucinating responses,
2. If the response is relevant and properly addresses the user's questions
3. What types of questions the users are asking

1 - Testing for hallucinations

Below is an example of an LLM-as-Judge online evaluator that takes as input both:

- `facts` which is a variable representing the raw source of information from the retrieval step, and
- `student answer` which is a variable representing the RAG application response

POST-PRODUCTION

This prompt checks to see if the application's response is grounded in the retrieved documents and prevents the introduction of unsupported information. We can represent the result of the test as a boolean (hallucination or not). This also further motivates why keeping track of interim trace steps, the retrieved documents, is important (and not just the input and final response).

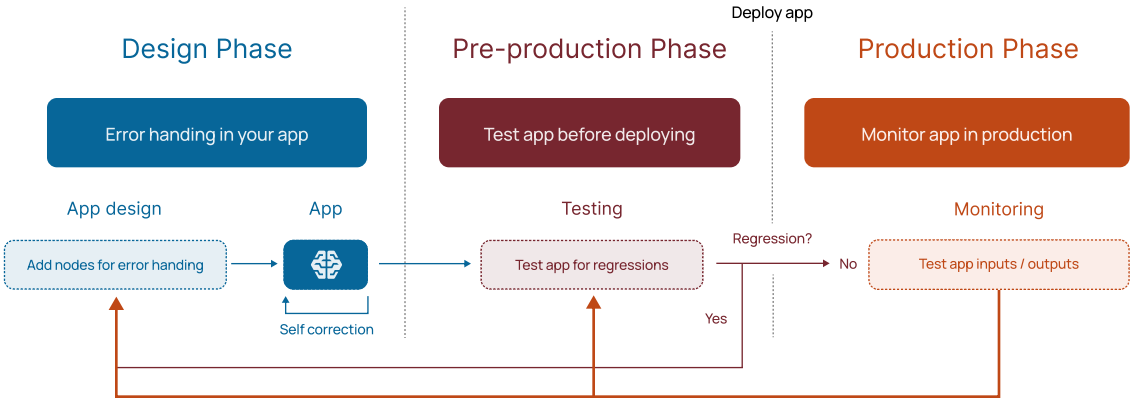
```
You are a teacher grading a quiz.
You will be given FACTS and a STUDENT ANSWER.
Here is the grade criteria to follow:
(1) Ensure the STUDENT ANSWER is grounded in the FACTS.
(2) Ensure the STUDENT ANSWER does not contain "hallucinated" information outside
the scope of the FACTS.

Score:
A score of 1 means that the student's answer meets all of the criteria. This is the
highest (best) score.
A score of 0 means that the student's answer does not meet all of the criteria.
This is the lowest possible score you can give.
Explain your reasoning in a step-by-step manner to ensure your reasoning and
conclusion are correct.
Avoid simply stating the correct answer at the outset.
```

To learn more about online evaluation, explore our educational videos:

- Online evaluation in our LangSmith Evaluation series
- Online evaluation with a focus on guardrails in our LangSmith Evaluation series
- Online evaluation with a focus on RAG in our LangSmith Evaluation series

Bootstrapping

After setting up tracing and online evaluators, you'll start to catch errors in your application in production. Ideally, you modify your application to fix these errors. You can also fold these errors back into your test dataset used for offline evaluation, in order to prevent the same mistakes in future releases of your application.



Bootstrapping to improve future versions

You can also do a phased release of your app on a smaller audience to gradually build up your dataset before doing a full cutover to the new version. Seeing a big jump in any of your monitoring charts in LangSmith should alert you to investigate further or do a rollback. This approach lets you spot tradeoffs between cost, latency, and quality.

LangChain

5.0

Conclusion

Conclusion

As large language models evolve rapidly, robust testing is needed to improve the system built around the model and provide lasting value. In this guide, we discussed three layers of testing that you can consider: (1) error handling within the application itself, (2) pre-production testing, and (3) post-production monitoring.

Together, these three layers of testing create a virtuous data flywheel. Production monitoring lets you identify application errors, informing the design process and pre-production (regression) testing. During the design phase, in-app error handling using frameworks like LangGraph can fix some of these errors and enable self-correction. Pre-production testing ensures each app version you ship avoids regressions and, ideally, improves performance on your collected examples.

LangChain products have helped over a million developers integrate generative AI into their software, and it's time to also integrate this flywheel of testing. With this guide, we hope you've gained a framework for robust LLM application testing, so you can iterate faster and systematically navigate decisions in the ever-changing LLM space.

LangChain

Glossary

Glossary

Agents: An agent is a system that uses an LLM to decide the control flow of an application.

Agent trajectory: The series of steps an agent took to complete a given task.

Experiment: An experiment is application code execution on all example inputs in a dataset and evaluated for the criteria you've defined.

LLM-as-Judge Evaluation: LLM-as-judge evaluators use LLMs to score your application's performance. To use them, you typically encode the grading rules / criteria in the LLM prompt. They can be reference-free (e.g., check if system output contains offensive content or adheres to specific criteria), or, they can compare task output to a reference (e.g., check if the output is factually accurate relative to the reference).

Offline evaluation: Offline evaluation is conducted prior to deployment of your LLM application. Usually you have a set of examples in the form of a dataset that you want to test your application on. Once you record the outputs of your application over all examples, you can evaluate performance based on tests you've created. We call this an experiment in LangSmith. These tests can be reference-free or rely on a grounded true response to compare your application's response against.

Glossary

Online evaluation: Online evaluation allows you to evaluate an application in production. This type of evaluation scores performance in real time, as your application handles user inputs. Notably, it does not rely on a grounded, true response for comparison.

RAG (Retrieval Augmented Generation): A technique for AI applications that leverages external knowledge from a knowledge base, retrieving relevant documents or other sources of information to generate more informed and context-aware responses.

Repetitions: Since LLM applications can exhibit considerable run-to-run variability, repetitions involve running the same evaluation multiple times and aggregating the results to smooth out run-to-run variability in order to examine the reproducibility of the AI application's performance.

LangChain

LangChain is the platform over a million developers choose for building AI apps from prototype to production. Created by the LangChain team, LangSmith is a unified platform for debugging, testing, deploying, and monitoring your LLM application. Thousands of organizations rely on LangSmith — including Rakuten, Home Depot, Elastic, and Grab — to build, run, and manage their LLM applications. Founded in 2022, LangChain is headquartered in San Francisco with customers worldwide.

Request Demo