Beginner's Guide to Agent Evaluations

https://www.youtube.com/watch?v=_QozKR9eQE8

Description

When companies deploy their agents into production, a key challenge emerges: how to evaluate whether the agent is performing as expected. You might find yourself asking: Is my agent on the right track? How can I ensure the final output is accurate?
In this video, we walk through how to build and evaluate a customer support agent, covering:

- The challenges of evaluating agents and practical approaches to overcome them
- How to create a golden dataset to evaluate against
- Evaluation strategies to assess agent performance

Transcript

Hi, I'm David from LangChain. Today, I'll be walking through how you can both build and evaluate a customer support agent.
Before filming this video, I built a customer support agent using LangGraph and ported it over to LangGraph Studio, which is where we are now. LangGraph Studio is a specialized tool where you can interact with, debug, and visualize your agent.
The customer support agent I built is designed to assist customers of a digital music store. This customer support agent has access to a SQL database that not only has information on the products and offerings of the digital music store but also details about customers who have shopped at the store in the past and their purchase history.
This customer support agent has two core functionalities:

1. The ability to answer questions about the products and offerings of the store, such as, "Do you have any songs by Amy Winehouse?" or "Do you have any albums by Pink Floyd?"
2. The ability to handle and process customer refunds.

Each of these functionalities is handled by a separate subgraph in our architecture. As we can see here, we have:

- A question-answering subgraph that handles queries related to the products and offerings of the music store.
- A refund subgraph that handles and processes customer refund requests.

These are routed by our supervisor or intent classifier node. When a query enters our graph, the intent classifier determines whether it should be routed to the refund subgraph or the question-answering subgraph.

The final node in this architecture is called `Compile Follow-Up` . All it does is clean up the state of our agent and return a neat final output to the user.

Let's actually ask our agent two questions and see it in action. I'll copy and paste a predefined message I have here:"My name is Mark Phillips. Here's my phone number so you can identify me in the SQL database. I want a refund on a past purchase."

When we click submit, our intent classifier node correctly routes it to the refund agent. Scrolling down, we see that the customer support agent responded:"All right, Mark. Which of the following purchases would you like to be refunded for?"

It returns a list of invoice IDs in the database associated with his name, first name, last name, and phone number. Mark can now follow up with the invoice IDs he'd like refunded, and the system will execute the refund for him.

For the sake of time, I won't ask that follow-up. Instead, I'll ask another question that will be handled by our question-answering subgraph:"Never mind. Do you have any songs for sale by Amy Winehouse?"

When I hit submit, the agent correctly routes the query to the question-answering subgraph. Scrolling down, we see the assistant's response:"Yes, we have several Amy Winehouse songs available from her albums 'Back to Black' and 'Frank.' There are 12 songs from 'Back to Black' and 11 from 'Frank.'"

Great! That's a quick overview of what our agent looks like and does. Now, let's dive into building and evaluating it.

Before I hop into a notebook and show you how to evaluate this customer support agent in code, I'll first touch on why evaluating an agent is difficult and important.

It's difficult to evaluate an agent because today, agents take a large number of steps before returning an output. These steps are not explicitly defined beforehand by a developer. Instead, they're determined dynamically by an LLM or multiple LLMs.

As a result, when you're evaluating an agent, you not only want to ensure it produces high-quality output but also that the path or trajectory it follows to construct that final output is optimal.

Let's use our customer support agent as an example. If I were evaluating this agent and it received a query like, "What songs do you have by Amy Winehouse?" I'd want to ensure that:

1. The agent produces a high-quality, accurate output.
2. The query is routed to the correct subgraph — in this case, the question-answering subgraph.
3. Within this subgraph, the right tools are called, in the right order.

It's important to evaluate both the final output and the trajectory or steps taken because you can have a situation where the output is high-quality, but the trajectory is inefficient.

For example, imagine the query about Amy Winehouse songs. The supervisor node routes it correctly to the question-answering subgraph. However, within that subgraph, unnecessary tools are called multiple times, or incorrect tools are used. While the

output might still be accurate, the inefficiency would cause unnecessary latency and token usage.

This highlights why evaluating an agent is so important. Let's say you make a change to your agent — switching to a new model provider, iterating on prompts, or making a substantial change to the architecture. You need to ensure this change doesn't degrade performance, whether in latency, token usage, or output quality. Catching regressions before pushing changes to production is critical.

Now let's evaluate this customer support agent using three strategies:

1. Evaluating the quality and accuracy of its final outputs.
2. Ensuring that a single step, such as the intent classification, behaves correctly.
3. Evaluating the entire trajectory or path that the agent follows to ensure it aligns with an optimal trajectory.

We'll use the LangSmith SDK for these evaluations.

To start, we'll define three key components for our evaluations:

1. **A golden dataset**: This contains example inputs and their corresponding expected outputs or behaviors.
2. **The application logic**: This refers to the customer support agent we built with LangGraph.
3. **Evaluators**: These assess the agent's outputs against the golden dataset to determine correctness and quality.

Strategy 1: Evaluating Final Output Accuracy

The first evaluation strategy assesses whether the agent produces high-quality, accurate outputs.

Step 1: Define the golden dataset. In this case, the golden dataset includes inputs — example queries that customers might ask — and outputs — high-quality responses we'd expect the agent to produce in production.

For instance, one input query might be:*"How many songs do you have by James Brown?"*

The corresponding output would be:*"We have 20 songs by James Brown."*

Once the golden dataset is defined, we send it to LangSmith for use in the evaluation process.

Step 2: Define the application logic.We'll create a target function that passes each input from the golden dataset to our customer support agent. The agent processes the input and generates a response, which we then capture for evaluation.

Step 3: Define the evaluator.The evaluator compares the generated output with the expected output from the golden dataset. It uses an LLM judge with a system prompt that instructs it to act as a teacher grading a quiz.

The LLM judge evaluates whether the agent's response is correct and provides reasoning for its decision. It returns a Boolean value (`true` for correct, `false` for incorrect) and an explanation.

Step 4: Run the evaluation.Using LangSmith's `evaluate` function, we specify:

- The target function (application logic).
- The golden dataset.
- The evaluator.
- The experiment name, which will appear in the LangSmith UI.

Switching to the LangSmith UI, we can see the results. For each input, the generated output is compared against the golden dataset. The LLM judge scores the output, providing a detailed rationale for each evaluation.
For example:

- Input: *"How many songs do you have by James Brown?"*
- Generated output: *"We have 20 songs by James Brown."*
- Result: Correct (scored as `1`).

However, if the agent struggles with a query, like:*"Who recorded 'Wish You Were Here'?* The LLM judge might mark the output as incorrect and provide an explanation, helping us pinpoint areas for improvement.

Strategy 2: Evaluating Single Steps

The second evaluation strategy focuses on a single step in the agent's process. For this demo, we'll evaluate the **intent classification step**, ensuring that the agent routes queries to the correct subgraph.
Step 1: Define the golden dataset.The inputs include messages or conversation histories, and the outputs specify the correct subgraph the query should be routed to.
For instance:

- Input: *"I bought some tracks recently and didn't like them."*
- Expected output: *Route to the refund subgraph.*

Step 2: Define the application logic.In this case, the target function bypasses the full graph and directly evaluates the intent classifier node. The function determines which subgraph the classifier routes the query to.
Step 3: Define the evaluator.The evaluator checks whether the intent classifier correctly routes the input to the expected subgraph. If the classifier's choice matches the golden dataset's output, it's marked as correct. Otherwise, it's flagged as incorrect.
Step 4: Run the evaluation.We use the same `evaluate` function, specifying the golden dataset, application logic, evaluator, and experiment name.
In the LangSmith UI, we can view the results, confirming whether the intent classifier consistently routes queries to the correct subgraph.

Strategy 3: Evaluating Trajectories

The third strategy evaluates the entire trajectory or path the agent follows when processing a query.
Step 1: Define the golden dataset.Each input includes a customer query, and the output specifies the optimal trajectory the agent should follow.
For example:

- Input: *"How many songs do you have by James Brown?"*
- Expected trajectory:
  1. Route to the question-answering subgraph.
  2. Use the `lookup track` tool.
  3. Route to the `Compile Follow-Up` node.

Step 2: Define the application logic.We modify the target function to capture and return the agent's trajectory for each input. The trajectory includes all nodes and tools the agent uses while processing the query.

Step 3: Define the evaluators.We use two evaluators:

- **Extra steps evaluator**: Flags any unnecessary steps taken by the agent.
- **Unmatched steps evaluator**: Ensures the steps followed match the expected trajectory in both content and order.

Step 4: Run the evaluation.After running the evaluation, the LangSmith UI displays the results, showing where the agent followed the optimal trajectory and where it deviated. For example:

- Input: *"Who recorded 'Wish You Were Here'? What other albums by them do you have?"*
- Observed trajectory: The agent took extra steps and mismatched steps, calling tools in the wrong order.

This highlights areas for optimization to ensure the agent processes queries efficiently and correctly.

Wrapping Up

These three evaluation strategies — final output accuracy, single-step performance, and trajectory evaluation — provide a comprehensive way to assess and refine your agent.

If you want to learn more about building and evaluating agents, I recommend checking out the Introduction to LangSmith and Introduction to LangGraph courses at academy.langchain.com/collections.

These resources provide a deeper dive into LangSmith for running experiments and managing datasets, as well as LangGraph, our open-source framework for building agentic applications.

I hope this video was helpful, and I'll see you in the next one!