LangSmith Walkthrough

LangChain makes it easy to prototype LLM applications and Agents. However, delivering LLM applications to production can be deceptively difficult. You will likely have to heavily customize and iterate on your prompts, chains, and other components to create a high-quality product.

To aid in this process, we've launched LangSmith, a unified platform for debugging, testing, and monitoring your LLM applications.

When might this come in handy? You may find it useful when you want to:

- Quickly debug a new chain, agent, or set of tools
- Visualize how components (chains, llms, retrievers, etc.) relate and are used
- Evaluate different prompts and LLMs for a single component
- Run a given chain several times over a dataset to ensure it consistently meets a quality bar
- Capture usage traces and using LLMs or analytics pipelines to generate insights

Prerequisites

Create a LangSmith account and create an API key (see bottom left corner). Familiarize yourself with the platform by looking through the docs

Note LangSmith is in closed beta; we're in the process of rolling it out to more users. However, you can fill out the form on the website for expedited access.

Now, let's get started!

Log runs to LangSmith

First, configure your environment variables to tell LangChain to log traces. This is done by setting the `LANGCHAIN_TRACING_V2` environment variable to true. You can tell LangChain which project to log to by setting the `LANGCHAIN_PROJECT` environment variable (if this isn't set, runs will be logged to the `default` project). This will automatically create the project for you if it doesn't exist. You must also set the `LANGCHAIN_ENDPOINT` and `LANGCHAIN_API_KEY` environment variables.

For more information on other ways to set up tracing, please reference the LangSmith documentation

NOTE: You must also set your `OPENAI_API_KEY` and `SERPAPI_API_KEY` environment variables in order to run the following tutorial.

NOTE: You can only access an API key when you first create it. Keep it somewhere safe.

NOTE: You can also use a context manager in python to log traces using

```
from langchain.callbacks.manager import tracing_v2_enabled

with tracing_v2_enabled(project_name="My Project"):
    agent.run("How many people live in canada as of 2023?")
```

API Reference:

- tracing_v2_enabled from `langchain.callbacks.manager`

However, in this example, we will use environment variables.

```
import os
from uuid import uuid4

unique_id = uuid4().hex[0:8]
os.environ["LANGCHAIN_TRACING_V2"] = "true"
os.environ["LANGCHAIN_PROJECT"] = f"Tracing Walkthrough - {unique_id}"
os.environ["LANGCHAIN_ENDPOINT"] = "https://api.smith.langchain.com"
os.environ["LANGCHAIN_API_KEY"] = ""  # Update to your API key

# Used by the agent in this tutorial
# os.environ["OPENAI_API_KEY"] = "<YOUR-OPENAI-API-KEY>"
# os.environ["SERPAPI_API_KEY"] = "<YOUR-SERPAPI-API-KEY>"
```

Create the langsmith client to interact with the API

```
from langsmith import Client

client = Client()
```

Create a LangChain component and log runs to the platform. In this example, we will create a ReAct-style agent with access to Search and Calculator as tools. However, LangSmith works regardless of which type of LangChain component you use (LLMs, Chat Models, Tools, Retrievers, Agents are all supported).

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import AgentType, initialize_agent, load_tools

llm = ChatOpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm=llm)
agent = initialize_agent(
    tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
verbose=False
)
```

API Reference:

- ChatOpenAI from `langchain.chat_models`
- AgentType from `langchain.agents`
- initialize_agent from `langchain.agents`
- load_tools from `langchain.agents`

We are running the agent concurrently on multiple inputs to reduce latency. Runs get logged to LangSmith in the background so execution latency is unaffected.

```
import asyncio

inputs = [
    "How many people live in canada as of 2023?",
    "who is dua lipa's boyfriend? what is his age raised to the .43
power?",
    "what is dua lipa's boyfriend age raised to the .43 power?",
    "how far is it from paris to boston in miles",
    "what was the total number of points scored in the 2023 super bowl?
what is that number raised to the .23 power?",
    "what was the total number of points scored in the 2023 super bowl
raised to the .23 power?",
    "how many more points were scored in the 2023 super bowl than in
the 2022 super bowl?",
    "what is 153 raised to .1312 power?",
    "who is kendall jenner's boyfriend? what is his height (in inches)
raised to .13 power?",
    "what is 1213 divided by 4345?",
]
results = []
```

```
async def arun(agent, input_example):
    try:
        return await agent.arun(input_example)
    except Exception as e:
        # The agent sometimes makes mistakes! These will be captured by
the tracing.
        return e


for input_example in inputs:
    results.append(arun(agent, input_example))
results = await asyncio.gather(*results)
```

```
from langchain.callbacks.tracers.langchain import wait_for_all_tracers

# Logs are submitted in a background thread to avoid blocking
execution.
# For the sake of this tutorial, we want to make sure
# they've been submitted before moving on. This is also
# useful for serverless deployments.
wait_for_all_tracers()
```

API Reference:

- wait_for_all_tracers from `langchain.callbacks.tracers.langchain`

Assuming you've successfully set up your environment, your agent traces should show up in the `Projects` section in the app. Congrats!

Evaluate another agent implementation

In addition to logging runs, LangSmith also allows you to test and evaluate your LLM applications.

In this section, you will leverage LangSmith to create a benchmark dataset and run AI-assisted evaluators on an agent. You will do so in a few steps:

1. Create a dataset from pre-existing run inputs and outputs
2. Initialize a new agent to benchmark
3. Configure evaluators to grade an agent's output

4. Run the agent over the dataset and evaluate the results

1. Create a LangSmith dataset

Below, we use the LangSmith client to create a dataset from the agent runs you just logged above. You will use these later to measure performance for a new agent. This is simply taking the inputs and outputs of the runs and saving them as examples to a dataset. A dataset is a collection of examples, which are nothing more than input-output pairs you can use as test cases to your application.

Note: this is a simple, walkthrough example. In a real-world setting, you'd ideally first validate the outputs before adding them to a benchmark dataset to be used for evaluating other agents.

For more information on datasets, including how to create them from CSVs or other files or how to create them in the platform, please refer to the LangSmith documentation.

```
dataset_name = f"calculator-example-dataset-{unique_id}"

dataset = client.create_dataset(
    dataset_name, description="A calculator example dataset"
)

runs = client.list_runs(
    project_name=os.environ["LANGCHAIN_PROJECT"],
    execution_order=1,  # Only return the top-level runs
    error=False,  # Only runs that succeed
)
for run in runs:
    client.create_example(inputs=run.inputs, outputs=run.outputs,
dataset_id=dataset.id)
```

2. Initialize a new agent to benchmark

You can evaluate any LLM, chain, or agent. Since chains can have memory, we will pass in a `chain_factory` (aka a `constructor`) function to initialize for each call.

In this case, we will test an agent that uses OpenAI's function calling endpoints.

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import AgentType, initialize_agent, load_tools

llm = ChatOpenAI(model="gpt-3.5-turbo-0613", temperature=0)
```

```
tools = load_tools(["serpapi", "llm-math"], llm=llm)


# Since chains can be stateful (e.g. they can have memory), we provide
# a way to initialize a new chain for each row in the dataset. This is
done
# by passing in a factory function that returns a new chain for each
row.
def agent_factory():
    return initialize_agent(tools, llm,
agent=AgentType.OPENAI_FUNCTIONS, verbose=False)


# If your chain is NOT stateful, your factory can return the object
directly
# to improve runtime performance. For example:
# chain_factory = lambda: agent
```

API Reference:

- ChatOpenAI from `langchain.chat_models`
- AgentType from `langchain.agents`
- initialize_agent from `langchain.agents`
- load_tools from `langchain.agents`

3. Configure evaluation

Manually comparing the results of chains in the UI is effective, but it can be time consuming. It can be helpful to use automated metrics and AI-assisted feedback to evaluate your component's performance.

Below, we will create some pre-implemented run evaluators that do the following:

- Compare results against ground truth labels. (You used the debug outputs above for this)
- Measure semantic (dis)similarity using embedding distance
- Evaluate 'aspects' of the agent's response in a reference-free manner using custom criteria

For a longer discussion of how to select an appropriate evaluator for your use case and how to create your own custom evaluators, please refer to the LangSmith documentation.

```
from langchain.evaluation import EvaluatorType
from langchain.smith import RunEvalConfig
```

```
evaluation_config = RunEvalConfig(
    # Evaluators can either be an evaluator type (e.g., "qa",
"criteria", "embedding_distance", etc.) or a configuration for that
evaluator
    evaluators=[
        # Measures whether a QA response is "Correct", based on a
reference answer
        # You can also select via the raw string "qa"
        EvaluatorType.QA,
        # Measure the embedding distance between the output and the
reference answer
        # Equivalent to:
EvalConfig.EmbeddingDistance(embeddings=OpenAIEmbeddings())
        EvaluatorType.EMBEDDING_DISTANCE,
        # Grade whether the output satisfies the stated criteria. You
can select a default one such as "helpfulness" or provide your own.
        RunEvalConfig.LabeledCriteria("helpfulness"),
        # Both the Criteria and LabeledCriteria evaluators can be
configured with a dictionary of custom criteria.
        RunEvalConfig.Criteria(
            {
                "fifth-grader-score": "Do you have to be smarter than a
fifth grader to answer this question?"
            }
        ),
    ],
    # You can add custom StringEvaluator or RunEvaluator objects here
as well, which will automatically be
    # applied to each prediction. Check out the docs for examples.
    custom_evaluators=[],
)
```

API Reference:

- EvaluatorType from `langchain.evaluation`
- RunEvalConfig from `langchain.smith`

4. Run the agent and evaluators

Use the arun_on_dataset (or synchronous run_on_dataset) function to evaluate your model. This will:

1. Fetch example rows from the specified dataset
2. Run your llm or chain on each example.

3. Apply evalutors to the resulting run traces and corresponding reference examples to generate automated feedback.

The results will be visible in the LangSmith app.

```
from langchain.smith import (
    arun_on_dataset,
    run_on_dataset,  # Available if your chain doesn't support async
calls.
)

chain_results = await arun_on_dataset(
    client=client,
    dataset_name=dataset_name,
    llm_or_chain_factory=agent_factory,
    evaluation=evaluation_config,
    verbose=True,
    tags=["testing-notebook"],  # Optional, adds a tag to the resulting
chain runs
)

# Sometimes, the agent will error due to parsing issues, incompatible
tool inputs, etc.
# These are logged as warnings here and captured as errors in the
tracing UI.
```

API Reference:

- arun_on_dataset from `langchain.smith`
- run_on_dataset from `langchain.smith`

```
    View the evaluation results for project '2023-07-17-11-25-20-
AgentExecutor' at:
    https://dev.smith.langchain.com/projects/p/1c9baec3-ae86-4fac-9e99-
e1b9f8e7818c?eval=true
    Processed examples: 1

    Chain failed for example 5a2ac8da-8c2b-4d12-acb9-5c4b0f47fe8a.
Error: LLMMathChain._evaluate("
    age_of_Dua_Lipa_boyfriend ** 0.43
    ") raised error: 'age_of_Dua_Lipa_boyfriend'. Please try again with
a valid numerical expression
```

```
    Processed examples: 4

    Chain failed for example 91439261-1c86-4198-868b-a6c1cc8a051b.
Error: Too many arguments to single-input tool Calculator. Args:
['height ^ 0.13', {'height': 68}]


    Processed examples: 9
```

Review the test results

You can review the test results tracing UI below by navigating to the "Datasets & Testing" page and selecting the **"calculator-example-dataset-*"** dataset, clicking on the `Test Runs` tab, then inspecting the runs in the corresponding project.

This will show the new runs and the feedback logged from the selected evaluators. Note that runs that error out will not have feedback.

Exporting datasets and runs

LangSmith lets you export data to common formats such as CSV or JSONL directly in the web app. You can also use the client to fetch runs for further analysis, to store in your own database, or to share with others. Let's fetch the run traces from the evaluation run.

```
runs = list(client.list_runs(dataset_name=dataset_name))
runs[0]
```

```
    Run(id=UUID('e39f310b-c5a8-4192-8a59-6a9498e1cb85'),
name='AgentExecutor', start_time=datetime.datetime(2023, 7, 17, 18, 25,
30, 653872), run_type=<RunTypeEnum.chain: 'chain'>,
end_time=datetime.datetime(2023, 7, 17, 18, 25, 35, 359642), extra=
{'runtime': {'library': 'langchain', 'runtime': 'python', 'platform':
'macOS-13.4.1-arm64-arm-64bit', 'sdk_version': '0.0.8',
'library_version': '0.0.231', 'runtime_version': '3.11.2'},
'total_tokens': 512, 'prompt_tokens': 451, 'completion_tokens': 61},
error=None, serialized=None, events=[{'name': 'start', 'time': '2023-
07-17T18:25:30.653872'}, {'name': 'end', 'time': '2023-07-
17T18:25:35.359642'}], inputs={'input': 'what is 1213 divided by
4345?'}, outputs={'output': '1213 divided by 4345 is approximately
0.2792.'}, reference_example_id=UUID('a75cf754-4f73-46fd-b126-
9bcd0695e463'), parent_run_id=None, tags=['openai-functions', 'testing-
notebook'], execution_order=1, session_id=UUID('1c9baec3-ae86-4fac-
9e99-e1b9f8e7818c'), child_run_ids=[UUID('40d0fdca-0b2b-47f4-a9da-
f2b229aa4ed5'), UUID('cfa5130f-264c-4126-8950-ec1c4c31b800'),
UUID('ba638a2f-2a57-45db-91e8-9a7a66a42c5a'), UUID('fcc29b5a-cdb7-4bcc-
8194-47729bbdf5fb'), UUID('a6f92bf5-cfba-4747-9336-370cb00c928a'),
UUID('65312576-5a39-4250-b820-4dfae7d73945')], child_runs=None,
feedback_stats={'correctness': {'n': 1, 'avg': 1.0, 'mode': 1},
'helpfulness': {'n': 1, 'avg': 1.0, 'mode': 1}, 'fifth-grader-score':
{'n': 1, 'avg': 1.0, 'mode': 1}, 'embedding_cosine_distance': {'n': 1,
'avg': 0.144522385071361, 'mode': 0.144522385071361}})
```

```
client.read_project(project_id=runs[0].session_id).feedback_stats
```

```
    {'correctness': {'n': 7, 'avg': 0.5714285714285714, 'mode': 1},
     'helpfulness': {'n': 7, 'avg': 0.7142857142857143, 'mode': 1},
     'fifth-grader-score': {'n': 7, 'avg': 0.7142857142857143, 'mode':
1},
     'embedding_cosine_distance': {'n': 7,
      'avg': 0.11462010799473926,
      'mode': 0.0130477459560272}}
```

Conclusion

Congratulations! You have succesfully traced and evaluated an agent using LangSmith!

This was a quick guide to get started, but there are many more ways to use LangSmith to speed up your developer flow and produce better results.

For more information on how you can get the most out of LangSmith, check out LangSmith documentation, and please reach out with questions, feature requests, or feedback at support@langchain.dev.