Debugging

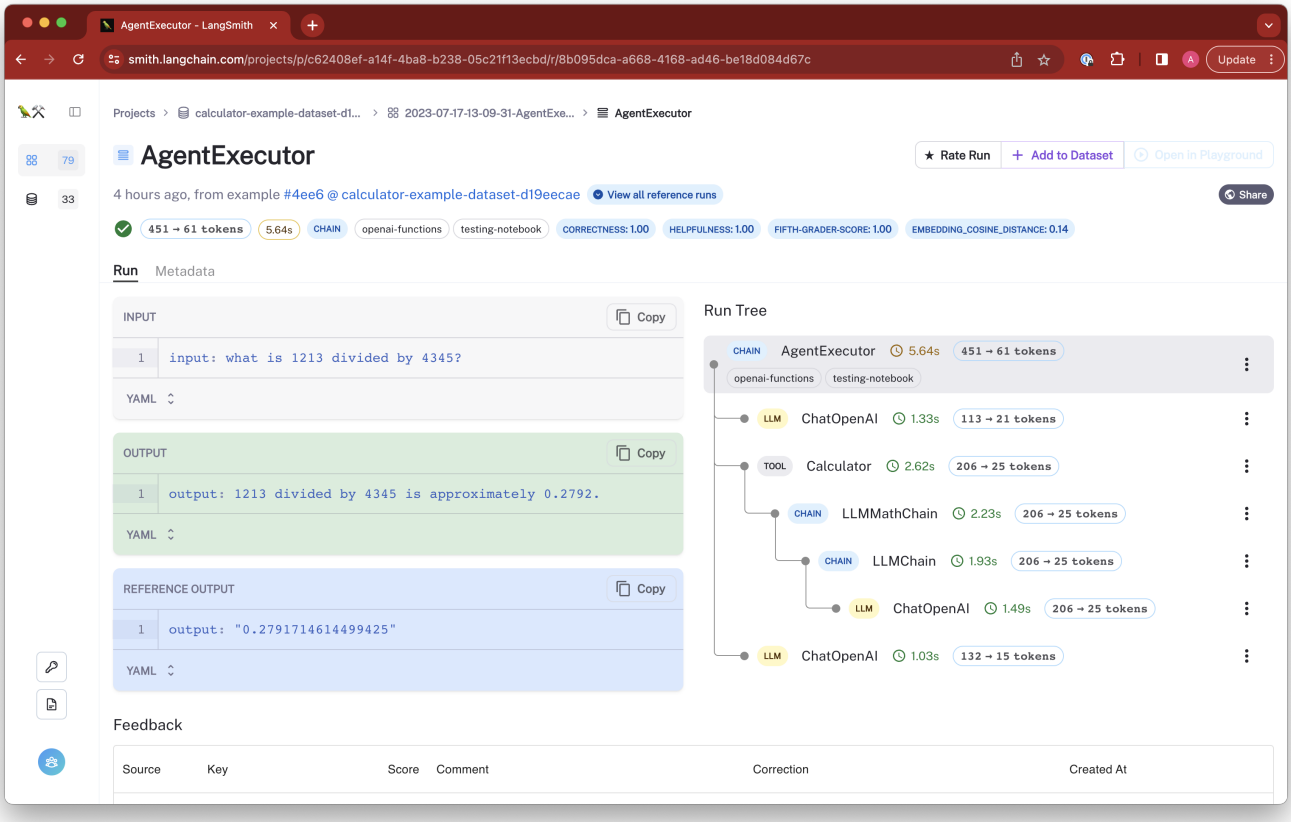
If you're building with LLMs, at some point something will break, and you'll need to debug. A model call will fail, or the model output will be misformatted, or there will be some nested model calls and it won't be clear where along the way an incorrect output was created.

Here's a few different tools and functionalities to aid in debugging.

Tracing

Platforms with tracing capabilities like LangSmith and WandB are the most comprehensive solutions for debugging. These platforms make it easy to not only log and visualize LLM apps, but also to actively debug, test and refine them.

For anyone building production-grade LLM applications, we highly recommend using a platform like this.



`langchain.debug` and `langchain.verbose`

If you're prototyping in Jupyter Notebooks or running Python scripts, it can be helpful to print out the intermediate steps of a Chain run.

There's a number of ways to enable printing at varying degrees of verbosity.

Let's suppose we have a simple agent and want to visualize the actions it takes and tool outputs it receives. Without any debugging, here's what we see:

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

llm = ChatOpenAI(model_name="gpt-4", temperature=0)
tools = load_tools(["ddg-search", "llm-math"], llm=llm)
agent = initialize_agent(tools, llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION)
```

API Reference:

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

```
agent.run("Who directed the 2023 film Oppenheimer and what is their
age? What is their age in days (assume 365 days per year)?")
```

```
    'The director of the 2023 film Oppenheimer is Christopher Nolan and
he is approximately 19345 days old in 2023.'
```

`langchain.debug = True`

Setting the global `debug` flag will cause all LangChain components with callback support (chains, models, agents, tools, retrievers) to print the inputs they receive and outputs they generate. This is the most verbose setting and will fully log raw inputs and outputs.

```
import langchain
```

```
langchain.debug = True

agent.run("Who directed the 2023 film Oppenheimer and what is their
age? What is their age in days (assume 365 days per year)?")
```

▸ Console output

`langchain.verbose = True`

Setting the `verbose` flag will print out inputs and outputs in a slightly more readable format and will skip logging certain raw outputs (like the token usage stats for an LLM call) so that you can focus on application logic.

```
import langchain

langchain.verbose = True

agent.run("Who directed the 2023 film Oppenheimer and what is their
age? What is their age in days (assume 365 days per year)?")
```

▸ Console output

`Chain(..., verbose=True)`

You can also scope verbosity down to a single object, in which case only the inputs and outputs to that object are printed (along with any additional callbacks calls made specifically by that object).

```
# Passing verbose=True to initialize_agent will pass that along to the
AgentExecutor (which is a Chain).
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True,
)

agent.run("Who directed the 2023 film Oppenheimer and what is their
age? What is their age in days (assume 365 days per year)?")
```

▸ Console output

Other callbacks

`Callbacks` are what we use to execute any functionality within a component outside the primary component logic. All of the above solutions use `Callbacks` under the hood to log intermediate steps of components. There's a number of `Callbacks` relevant for debugging that come with LangChain out of the box, like the FileCallbackHandler. You can also implement your own callbacks to execute custom functionality.

See here for more info on Callbacks, how to use them, and customize them.