

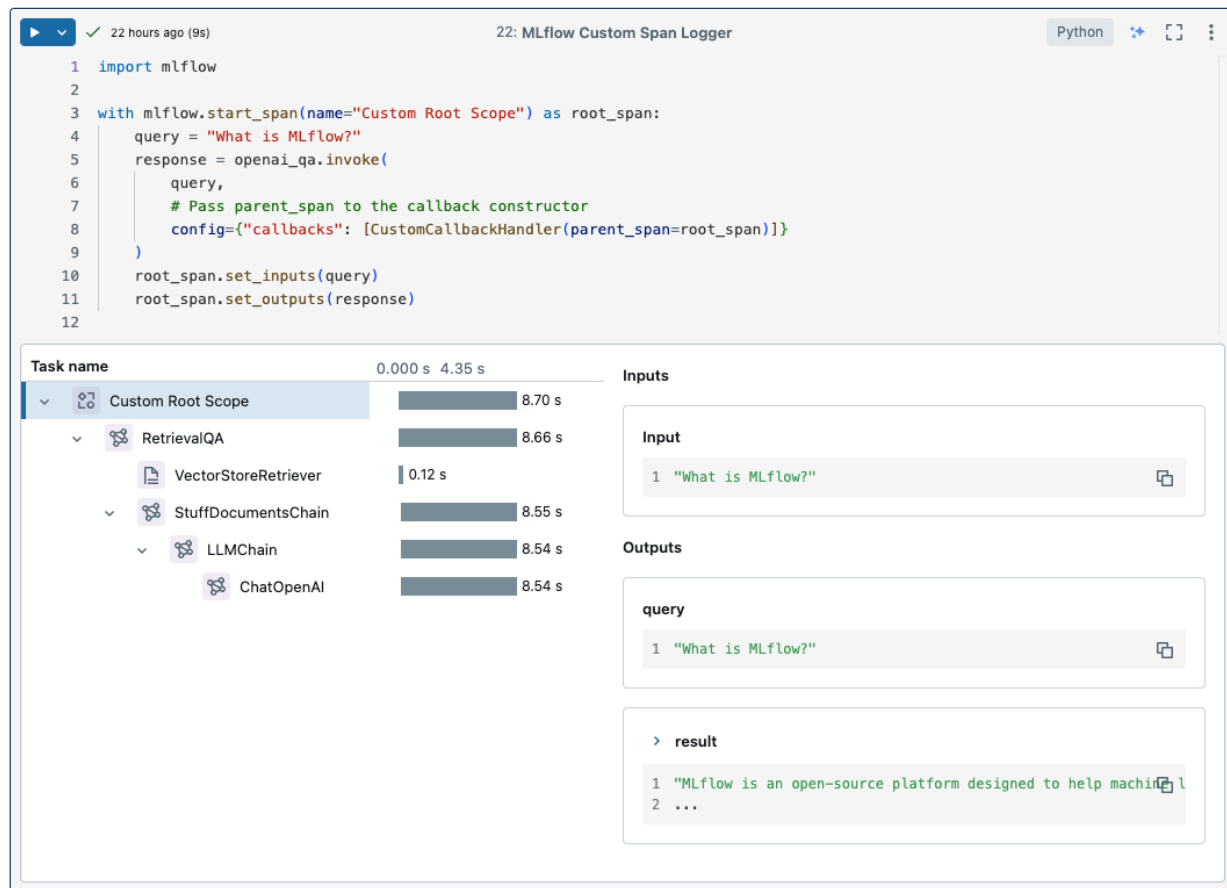
MLflow Tracing – Private Preview Docs for Customers

This article introduces MLflow Tracing and shows you how to get started with tracing in your AI systems.

What is MLflow Tracing?

Tracing, a well-established concept in software engineering, involves recording sequences of events like user sessions or request flows. In the context of AI systems, tracing often refers to interactions you have with an AI system. For example, a user message (prompt), followed by a vector lookup, followed by an AI response, example of a trace.

With MLflow Tracing, developers can now log, analyze, and compare traces across different versions of their AI systems. Users can now easily debug in their GenAI Python code and keep track of inputs & responses where their AI systems perform well or perform poorly. MLflow Tracing is tightly integrated with Databricks tools and infrastructure, allowing you to store and display all your traces in Databricks Notebooks or the Experimentation UI.



Why use MLflow Tracing?

Here are some possible benefits to using MLflow tracking your development workflow.

- If you identify a **possible hallucination** during development, you can now view the trace to **find the root cause of the problem**.
- You can verify that **prompt templates and guardrails are working correctly** and producing reasonable results.
- You can understand **how different frameworks, models, chunk sizes and techniques impact latency**.
- You can understand **how many tokens are used by different model calls** to measure application cost.
- You can use the trace data to **establish benchmark ("golden") datasets** that can be used to evaluate the performance of each new version of your AI system to ensure that it meets the criteria for deployment to production.

Limitations

- Only available in Databricks Notebooks and Notebook Jobs. We will support additional Databricks environments (e.g. DLT) and MLflow OSS soon.
- Not yet available in GovCloud regions. We will add support for them soon.
- [LangChain autologging](#) only supports the `invoke()` API. We will support additional LangChain inference APIs soon.

Requirements

- Turn on Preview in Databricks Workspace.
- MLflow 2.11.3 (Private Preview .whl)
- MLflow-Skinny 2.11.3 (Private Preview .whl)

Quickstart

Overview of MLflow Tracing: [Video](#)

Install MLflow Tracing

```
Python
# Install external dependencies for this demo
%pip install opentelemetry-api opentelemetry-sdk databricks-vectorsearch
tiktoken langchain -q

# Uninstall existing mlflow to avoid installation issues
%pip uninstall mlflow mlflow-skinny -y

# Install the MLflow trace prototype .whl
%pip install
"https://mlflow-snapshots.s3-us-west-2.amazonaws.com/mlflow-2.11.3-0.fd95310a50
de7df2-py3-none-any.whl" -U
%pip install
"https://mlflow-snapshots.s3-us-west-2.amazonaws.com/mlflow_skinny-2.11.3-0.fd9
5310a50de7df2-py3-none-any.whl" -U
```

```
dbutils.library.restartPython()
```

Add Traces to your API

Python

```
import mlflow
from mlflow.deployments import get_deploy_client

class QAChain(mlflow.pyfunc.PythonModel):
    def __init__(self):
        self.client = get_deploy_client("databricks")

    @mlflow.trace(name="quickstart-chain")
    def predict(self, model_input, system_prompt, params):
        messages = [
            {
                "role": "system",
                "content": system_prompt,
            },
            {
                "role": "user",
                "content": model_input[0]["query"]
            }
        ]

        traced_predict = mlflow.trace(self.client.predict)
        output = traced_predict(
            endpoint=params["model_name"],
            inputs={
                "temperature": params["temperature"],
                "max_tokens": params["max_tokens"],
                "messages": messages,
            },
        )

        with mlflow.start_span(name="_final_answer") as span:
            span.set_inputs({"query": model_input[0]["query"]})

            answer = output["choices"][0]["message"]["content"]

            span.set_outputs({"generated_text": answer})
```

```

    # Attributes computed at runtime can be set using the set_attributes()
    method.
    span.set_attributes({
        "model_name": params["model_name"],
        "prompt_tokens": output["usage"]["prompt_tokens"],
        "completion_tokens": output["usage"]["completion_tokens"],
        "total_tokens": output["usage"]["total_tokens"]
    })
    return answer

```

Perform inference

```

Python
SYSTEM_PROMPT = """
You are an assistant for Databricks users. You are answering python, coding,
SQL, data engineering, spark, data science, DW and platform, API or
infrastructure administration question related to Databricks. If the question
is not related to one of these topics, kindly decline to answer. If you don't
know the answer, just say that you don't know, don't try to make up an answer.
Keep the answer as concise as possible. Use the following pieces of context to
answer the question at the end:
"""

model = QChain()

prediction = model.predict(
    [
        {"query": "What is in MLflow 5.0"},
    ],
    SYSTEM_PROMPT,
    {
        # Using Databricks Foundation Model for easier testing, feel free to
        # replace it.
        "model_name": "databricks-dbrx-instruct",
        "temperature": 0.1,
        "max_tokens": 1000,
    }
)

```

The trace will be automatically shown when you run the cell

Task name	0.000 s	0.45 s	0.89 s
quickstart-chain	1.33 s		
predict	1.33 s		
_final_answer	0.001 s		

Attributes
function_name predict

Inputs

```

1 {
2   "model_input": [
3     {
4       "query": "What is in MLflow 5.0"
5     }
6   ],
7   "SYSTEM_PROMPT": "\nYou are an assistant for Databricks users. You
are answering python, coding, SQL, data engineering, spark, data science,
DW and platform, API or infrastructure administration question related to
Databricks. If the question is not related to one of these topics, kindly
decline to answer. If you don't know the answer, just say that you don't
know, don't try to make up an answer. Keep the answer as concise as
possible. Use the following pieces of context to answer the question at
the end:\n",
8   "params": {
9     "model_name": "databricks-dbrx-instruct",
10    "temperature": 0.1,
11    "max_tokens": 1000
12  }
13 }

```

Outputs

```

1 {
2   "output": "I'm sorry for any confusion, but MLflow is an open-source
platform for managing machine learning workflows, and it is not versioned
in the same way that Databricks Runtime is. The latest version of MLflow
as of my knowledge up to 2021 is MLflow 1.23.0. It would be best to refer
to the official MLflow documentation or GitHub repository for the most
up-to-date information on its features and versions."
3 }

```

Sample Notebook

1. In any Databricks Notebook. Click **File > Import Notebook**
2. Click the **URL** option
3. **Paste the Sample Notebook URL**

Unset

<https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173b9cf/2830662238121329/704769777646422/8538262732615206/latest.html>

4. Enjoy!

Reference Sample Notebook: [Sample Code here](#)

Trace your custom GenAI model

MLflow Tracing provides two different ways of instrumenting your custom GenAI model:

- **Fluent APIs** – Low-code APIs for instrumenting AI systems without worrying about the tree structure of the trace. MLflow will determine the appropriate parent-child tree structure (spans) based on the Python stack.
- **MLflow Client APIs** – MLflowClient implements more granular, thread safe APIs for advanced use cases. These APIs don't manage the parent-child relationship of the spans, so you need to manually specify it to construct the desired trace structure. This requires more boilerplate code but gives you better control over the trace lifecycle, particularly for multithreaded use cases.

The general recommendation is to use the fluent APIs as much as possible where it satisfies your needs because they produce cleaner code and are less error prone. However, there are certain use cases that require more control, such as multi-threaded applications, callback-based instrumentation, etc. MLflow Client APIs are suitable for such use cases.

Fluent APIs

There are 3 ways of instrumenting your code using fluent APIs. MLflow automatically constructs the trace hierarchy based on where/when the code is executed. The following sections describe the usage of each method, but please refer to [examples](#) and source code in the [MLflow repository](#) for more detailed information.

1. Decorate a function

You can decorate your function with the [@mlflow.trace](#) decorator to create a span for the scope of the decorated function. The span starts when the function is invoked and ends when it returns. MLflow will automatically record the input and output of the function, as well as any exceptions raised from the function. For example, running the following code will create a span with the name "my_function", capturing the input arguments x and y, as well as the output of the function.

Python

```
@mlflow.trace(attributes={"key": "value"})
def my_function(x, y):
    return x + y
```

2. Use the tracing context manager

If you want to create a span for an arbitrary block of code, not just a function, you can use `mlflow.start_span()` as a context manager that wraps the code block. The span starts when the context is entered and ends when the context is exited. The span input and outputs should be provided manually via setter methods of the span object that is yielded from the context manager. Please refer to the [source code](#) for the full list of setter methods and properties.

Python

```
with mlflow.start_span("my_span") as span:
    span.set_inputs({"x": x, "y": y})
    result = x + y
    span.set_outputs(result)
    span.set_attribute("key", "value")
```

3. Wrap an external function

The `mlflow.trace` function can be used as a wrapper to trace a function of your choice. This is useful when you want to trace functions imported from external libraries. It will generate the same span as you would get by decorating that function.

Python

```
from external.library import func

traced_func = mlflow.trace(func)
traced_func(x, y, ...)
```

MLflow Client APIs

[MlflowClient](#) exposes granular, thread safe APIs to start and end traces, manage spans, and set span fields, providing full control of the trace lifecycle and structure. These APIs are useful when the fluent APIs are not sufficient for your requirements, such as multi-threaded applications, callbacks, etc. Here are the steps to create a complete trace with MLflow Client.

1. Create an instance of MlflowClient by `client = MlflowClient()`

2. Start a trace with `client.start_trace()` method. This initiates the trace context and starts an absolute root span. Here you can set your attributes, inputs, and outputs for the Trace. This will return the root span object.

⚠ You may notice we don't have an equivalent method in the fluent APIs. This is because fluent APIs automatically initialize the trace context and determine if it is the root span or not based on the managed state.

3. Get the request id (an unique identifier of the trace, synonym to "trace_id") and the id of the returned span by `span.request_id` and `span.span_id`.
4. Start a child span by doing `client.start_span(request_id, parent_span_id=span_id)`. Here you can set your attributes, inputs, and outputs for the Span. This method requires request id and parent span id so it can associate the span to the correct position in the trace hierarchy. This will return another span object.
5. End the child span by calling `client.end_span(request_id, span_id)`.
6. Repeat 3 – 5 for any children spans you want to create.
7. Finally, once all the children spans are ended, call `client.end_trace(request_id)`. This will close the entire trace and record it.

Example Code:

```
Python
from mlflow.client import MlflowClient

mlflow_client = MlflowClient()

root_span = mlflow_client.start_trace(name="simple-rag-chain",
    inputs={"query": "Demo",
            "model_name": "DBRX",
            "temperature": 0,
            "max_tokens": 200})

request_id = root_span.request_id

# Retrieve documents that simliar to the query
similarity_search_input = dict(query_text="demo", num_results=3)

span_ss = mlflow_client.start_span(
    "search",
    # Specify request_id and parent_span_id to create the span at the
    # right position in the trace
```

```

        request_id=request_id,
        parent_span_id=root_span.span_id,
        inputs=similarity_search_input
    )
    retrieved = ["Test Result"]
    # Span has to be ended explicitly
    mlflow_client.end_span(request_id, span_id=span_ss.span_id,
        outputs=retrieved)

    root_span.end_trace(request_id, outputs={"output": retrieved})

```

Please refer to [the example](#) and source code in the [MLflow repository](#) for more detailed information.

Use with RAG Studio

Start with the [RAG Studio Sample Repo](#)

1. Update the [wheel_installer](#) file by adding the code below

```

Python
# External dependencies
%pip install opentelemetry-api opentelemetry-sdk databricks-vectorsearch
tiktoken langchain langchainhub faiss-cpu -U -q
%pip uninstall mlflow mlflow-skinny -y # uninstall existing mlflow to avoid
installation issues

%pip install
"https://mlflow-snapshots.s3-us-west-2.amazonaws.com/mlflow-2.11.3-0.fd95310a50
de7df2-py3-none-any.whl" -U

%pip install
"https://mlflow-snapshots.s3-us-west-2.amazonaws.com/mlflow_skinny-2.11.3-0.fd9
5310a50de7df2-py3-none-any.whl" -U

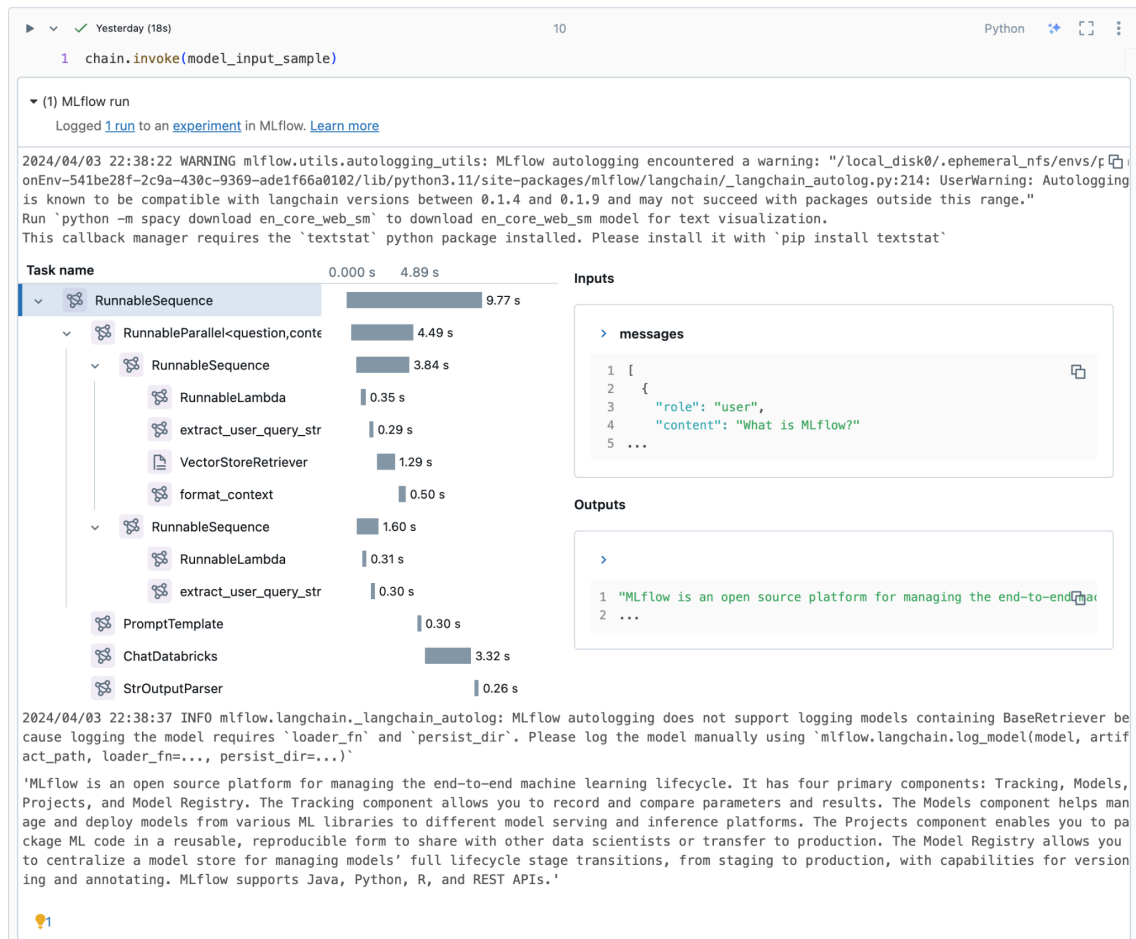
```

2. Simply add the following lines to your RAG Databricks notebook(s) to see the Traces

Python

```
import mlflow
mlflow.langchain.autolog()
```

3. You should see a Trace when you run. `chain.invoke(...)`



The screenshot displays a Databricks notebook interface with a Python cell containing the code `chain.invoke(model_input_sample)`. Below the code, an MLflow run trace is visible, showing a warning about autologging compatibility and a list of tasks with their durations. The tasks include `RunnableSequence` (9.77 s), `RunnableParallel<question,cont` (4.49 s), `RunnableSequence` (3.84 s), `RunnableLambda` (0.35 s), `extract_user_query_str` (0.29 s), `VectorStoreRetriever` (1.29 s), `format_context` (0.50 s), `RunnableSequence` (1.60 s), `RunnableLambda` (0.31 s), `extract_user_query_str` (0.30 s), `PromptTemplate` (0.30 s), `ChatDatabricks` (3.32 s), and `StrOutputParser` (0.26 s). The input messages are shown as a list containing a user query: "What is MLflow?". The output is a single string: "MLflow is an open source platform for managing the end-to-end machine learning lifecycle. It has four primary components: Tracking, Models, Projects, and Model Registry. The Tracking component allows you to record and compare parameters and results. The Models component helps manage and deploy models from various ML libraries to different model serving and inference platforms. The Projects component enables you to package ML code in a reusable, reproducible form to share with other data scientists or transfer to production. The Model Registry allows you to centralize a model store for managing models' full lifecycle stage transitions, from staging to production, with capabilities for versioning and annotating. MLflow supports Java, Python, R, and REST APIs."

Use with your LangChain application

1. Make sure you have installed all the [dependencies](#).
2. Simply add the following lines to any Databricks notebook where LangChain is used to see the Traces

Python

```
import mlflow
mlflow.langchain.autolog()
```

3. You should see a Trace when you run. `chain.invoke(...)`

Example Code:

Python

```
import os
import mlflow
import mlflow.deployments

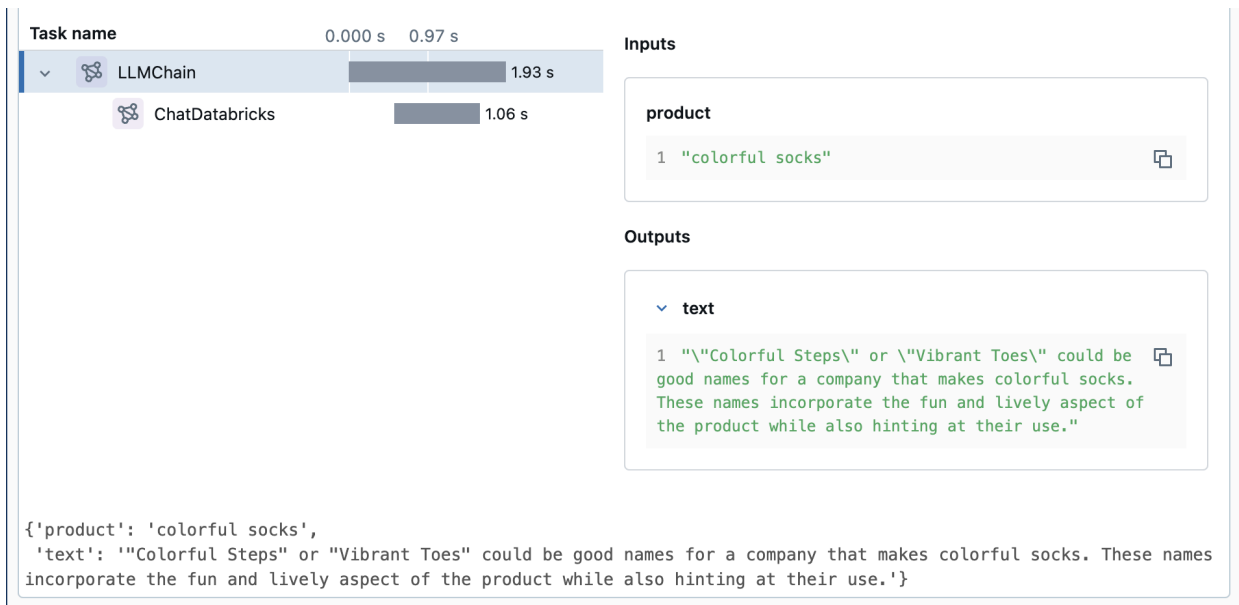
## Set up Autologging with MLflow and Set Deployment server to Databricks in
order to use DBRX
mlflow.langchain.autolog()
mlflow.deployments.get_deploy_client("databricks")

from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain_community.chat_models import ChatDatabricks

# Initialize the OpenAI model and the prompt template
llm = ChatDatabricks(endpoint="databricks-dbrx-instruct",
extra_params={"temperature": 0}, allow_dangerous_deserialization=True)
prompt = PromptTemplate(
    input_variables=["product"],
    template="What is a good name for a company that makes {product}?",
)

# Create the LLMChain with the specified model and prompt
chain = LLMChain(llm=llm, prompt=prompt)

chain.invoke({"product": "colorful socks"})
```



Examples using Trace SDK

Fluent API Example

The following is an example of instrumenting a simple RAG model using MLflow Tracing Fluent APIs.

```
Python
import mlflow
from mlflow.deployments import get_deploy_client
import pandas as pd
import tiktoken
from typing import Dict, List
from databricks.vector_search.client import VectorSearchClient

QUERY_SYSTEM_PROMPT = "I will ask you to assess whether a particular scientific claim is True or False. Answer 'NEE' if there's not enough evidence."
QUERY_USER_PROMPT = "The evidence is the following:\n{context}\n\nClaim:\n{claim}\n\nAssessment:\n"

class MockVectorSearchClient:
    class _Index:
```

```

def similarity_search(self, query_text, columns, filters, num_results):
    return {
        "result": {
            "data_array": [
                (
                    f"This is {i}th dummy content for query {query_text}",
                    f"doc_{i}",
                    f"https://SOME_URL_{i}",
                ) for i in range(num_results)
            ]
        }
    }

def get_index(self, *args, **kwargs):
    return self._Index()

class ExampleRAGChain:
    def __init__(
        self,
        index_name: str,
        vs_endpoint_name: str,
    ):
        vsc = MockVectorSearchClient() # Replace with VectorSearchClient() if
you want to use the real VS index.
        self.index = vsc.get_index(index_name, vs_endpoint_name)
        self.client = get_deploy_client("databricks")

    @mlflow.trace(name="simple-rag-chain")
    def predict(
        self,
        claim: str,
        model_name: str,
        temperature: float,
        max_tokens: int,
    ):
        # Retrieve documents that simliar to the query
        retrieved = mlflow.trace(self.index.similarity_search)(
            query_text=claim, columns=["content", "id", "url"], filters=[],
num_results=3,
        )["result"]["data_array"]

        # Query LLM with the query and contexts. (This can be a single method
in practice)
        with mlflow.start_span(name="final_answer") as span:

```

```

        context = " ".join(doc[0] for doc in retrieved)
        messages = [
            {"role": "system", "content": QUERY_SYSTEM_PROMPT},
            {"role": "user", "content":
QUERY_USER_PROMPT.format(claim=claim, context=context)}
        ]
        output = self.query_llm(messages, model_name, temperature,
max_tokens)

        span.set_inputs({"claim": claim, "retrieved": retrieved})
        span.set_outputs({"generated_text": output})

        # Attributes computed at runtime can be set using the
set_attributes() method.
        token_counts = self.count_tokens(messages, output)
        span.set_attributes({
            "input_token": token_counts["input"],
            "output_token": token_counts["output"]
        })

    return output

    @mlflow.trace()
    def query_llm(self, messages, model_name, temperature: float=0.1,
max_tokens: int=200) -> str:
        return self.client.predict(
            endpoint=model_name,
            inputs={
                "temperature": temperature,
                "max_tokens": max_tokens,
                "messages": messages,
            },
        )["choices"][0]["message"]["content"]

    @mlflow.trace(attributes={"tiktoken.encoding_for_model": "gpt-3.5-turbo"})
    def count_tokens(self, messages: List[Dict[str, str]], response: str) ->
Dict[str, int]:
        encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")
        return {
            "input": len(encoding.encode(" ".join(message["content"] for
message in messages))),
            "output": len(encoding.encode(response))
        }

```

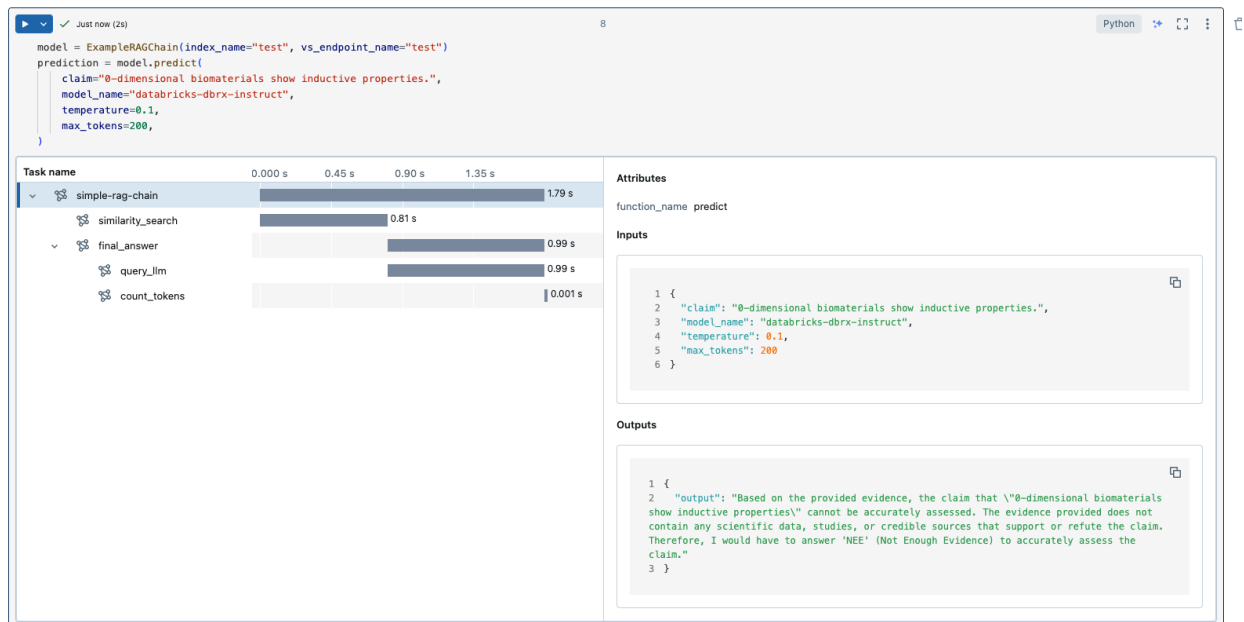
When running a prediction, the following trace will be generated and displayed.

Python

```
model = ExampleRAGChain(index_name="test", vs_endpoint_name="test")

params={
    # Using Databricks Foundation Model for easier testing, feel free to replace
    # it.
    "model_name": "databricks-dbrx-instruct",
    "temperature": 0.1,
    "max_tokens": 200,
}

output = model.predict(claim="What is ML?", model_name= params["model_name"],
    temperature= params["temperature"], max_tokens=params["max_tokens"])
```



MLflow Client API Example

The same can be achieved with MLflow Client APIs as follows.

Python

```
from mlflow.client import MlflowClient

class ExampleRAGChain:
    def __init__(
        self,
        index_name: str,
        vs_endpoint_name: str,
    ):
        vsc = VectorSearchClient()
        self.index = vsc.get_index(index_name, vs_endpoint_name)
        self.deploy_client = get_deploy_client("databricks")

        self.mlflow_client = MlflowClient()

    def predict(
        self,
        claim: str,
        model_name: str,
        temperature: float,
        max_tokens: int,
    ):
        # Call start_trace() at the beginning of the trace (and the root span)
        root_span = self.mlflow_client.start_trace(name="simple-rag-chain",
inputs={"claim": claim, "model_name": model_name, "temperature": temperature,
"max_tokens": max_tokens})
        request_id = root_span.request_id

        # Retrieve documents that simliar to the query
        similarity_search_input = dict(query_text=claim, columns=["content",
"id", "url"], filters=[], num_results=3)
        span_ss = self.mlflow_client.start_span(
            "similarity_search",
            # Specify request_id and parent_span_id to create the span at the
right position in the trace
            request_id=request_id,
            parent_span_id=root_span.span_id,
            inputs=similarity_search_input
        )
        retrieved =
self.index.similarity_search(**similarity_search_input)["result"]["data_array"]
```

```

        # Span has to be ended explicitly
        self.mlflow_client.end_span(request_id, span_id=span_ss.span_id,
outputs=retrieved)

        # Query LLM with the query and contexts. (This can be a single method
in practice)
        span_llm = self.mlflow_client.start_span(
            name="final_answer",
            request_id=request_id,
            parent_span_id=root_span.span_id,
            inputs={"claim": claim, "retrieved": retrieved},
        )
        context = " ".join(doc[0] for doc in retrieved)
        messages = [
            {"role": "system", "content": QUERY_SYSTEM_PROMPT},
            {"role": "user", "content": QUERY_USER_PROMPT.format(claim=claim,
context=context)}
        ]
        output = self.query_llm(messages, model_name, temperature, max_tokens,
parent_span=span_llm)

        # Attributes computed at runtime can be set using the set_attributes()
method.
        token_counts = self.count_tokens(messages, output,
parent_span=span_llm)
        self.mlflow_client.end_span(
            request_id=request_id,
            span_id=span_llm.span_id,
            outputs={"generated_text": output},
            attributes={
                "input_token": token_counts["input"],
                "output_token": token_counts["output"]
            }
        )

        # end_trace() needs to be called to close the trace and log it
        self.mlflow_client.end_trace(request_id, outputs={"output": output})
        return output

    def query_llm(self, messages, model_name, temperature: float=0.1,
max_tokens: int=200, parent_span=None) -> str:
        llm_inputs = dict(temperature=temperature, max_tokens=max_tokens,
messages=messages)
        span = self.mlflow_client.start_span(

```

```

        name="query_llm",
        request_id=parent_span.request_id,
        parent_span_id=parent_span.span_id,
        inputs=llm_inputs,
    )
    generated_text = self.deploy_client.predict(
        endpoint=model_name,
        inputs=llm_inputs
    )["choices"][0]["message"]["content"]
    self.mlflow_client.end_span(
        request_id=parent_span.request_id,
        span_id=span.span_id,
        outputs=generated_text,
    )
    return generated_text

def count_tokens(self, messages: List[Dict[str, str]], response: str,
parent_span=None) -> Dict[str, int]:
    span = self.mlflow_client.start_span(
        name="count_tokens",
        request_id=parent_span.request_id,
        parent_span_id=parent_span.span_id,
        inputs=messages,
        attributes={"tiktoken.encoding_for_model": "gpt-3.5-turbo"}
    )
    encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")
    counts = {
        "input": len(encoding.encode(" ".join(message["content"] for
message in messages))),
        "output": len(encoding.encode(response))
    }
    self.mlflow_client.end_span(
        request_id=parent_span.request_id,
        span_id=span.span_id,
        outputs=counts,
    )
    return counts

```

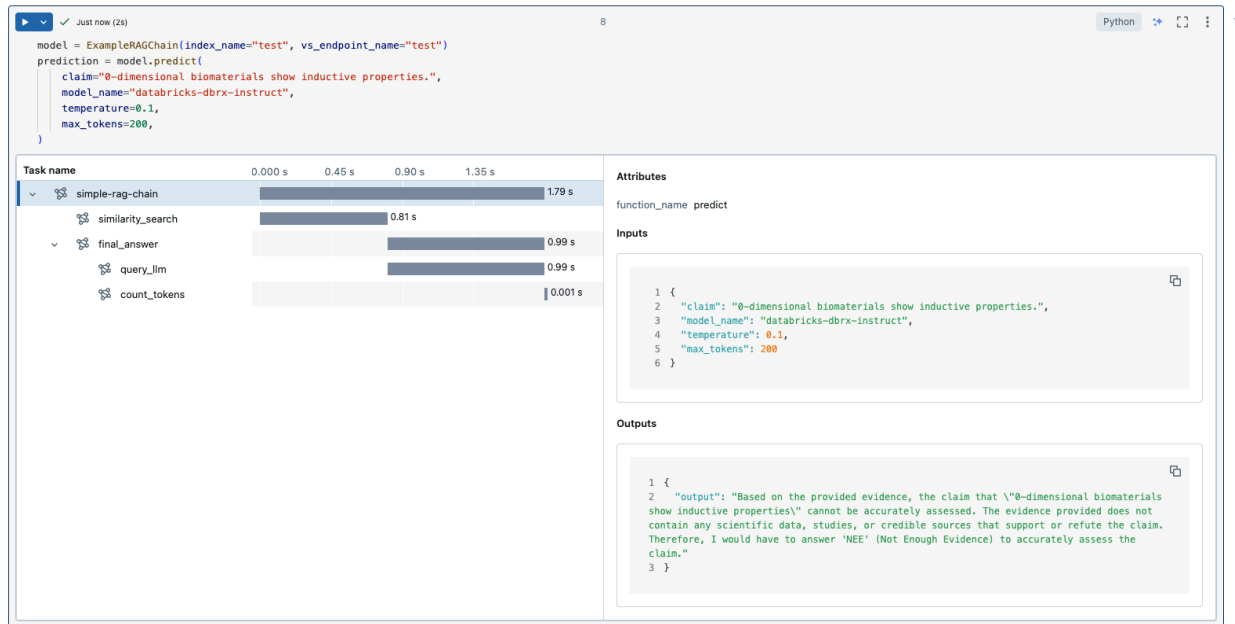
When running a prediction, the following trace will be generated and displayed.

Python

```
model = ExampleRAGChain(index_name="test", vs_endpoint_name="test")
```

```
params={  
    # Using Databricks Foundation Model for easier testing, feel free to replace  
    # it.  
    "model_name": "databricks-dbrx-instruct",  
    "temperature": 0.1,  
    "max_tokens": 200,  
}
```

```
output = model.predict(claim="What is ML?", model_name= params["model_name"],  
    temperature= params["temperature"], max_tokens=params["max_tokens"])
```



FAQs

Q. What happens when a trace is interrupted with an exception?

Traces will still be recorded with status set to ERROR. MLflow will record detailed information about the exception, such as the exception message and stack trace, in the [attributes](#) field of the span where the error occurred.

Q. What data types can be recorded as inputs/outputs/attributes of a span?

Any object can be set as a span input, output, or attribute. All objects are converted to strings for rendering in the UI.

Q. Can I use the fluent APIs and MLflow Client APIs together?

It is not recommended to mix them in general because it may confuse the automatic state management of the fluent APIs and result in broken trace structures. However, there is one particular case that is useful: creating a parent span with the fluent API and using the client API to create child spans underneath. You can achieve this by first creating the parent span using `mlflow.start_span()` context manager, getting the `span_id` and `request_id` from the yielded span, and passing them to the MLflow Client APIs as demonstrated [above](#).

- What does work: [parent span (fluent API)] -> [child span (client API)]
- What does NOT work: [parent span (client API)] -> [child span (fluent API)]

Feedback

[Please schedule a **feedback call**](#). The Product Team wants to learn about your experiences with MLflow Tracing and what could be improved. A good time for the call is 1 – 2 weeks after the feature is enabled in your workspace(s).

For any feedback or questions concerning the private preview, please contact your Databricks representative or abe@databricks.com