RAG Studio & Evaluation Suite Documentation

Jump to table of contents

Note: This documentation will be moved to docs.databricks.com.

Terminology

Note: The products are in the process of being officially named. Please do not get attached to our temporary names!

- **RAG Studio:** A marketing name that encompasses the upgraded Mosaic AI platform capabilities for building high-quality Retrieval Augmented Generation (RAG) applications:
    - **MLflow:**Support for logging, parameterizing, and tracing Chains that are unified between development & production. Chains can be logged as code vs. pickled.
    - **Model Serving:** Support for hosting Chains e.g., token streaming, automated authentication of Databricks services used in your chain, feedback API and a simplified chain deployment API
    - **RAG Cookbook:** Sample code & how-to guide offering an opinionated end-to-end workflow for building RAG apps
    - *[Future release]* **Lakehouse Monitoring:** Capabilities for monitoring your apps once in production
- **Mosaic AI Evaluation Suite:** Built-for-purpose tools to evaluate Generative AI Apps, starting with RAG apps:
    - **Evaluation Harness:** `evaluate(...)` command that runs the evaluation
    - **Review App:** UI tool for collecting stakeholder feedback & building evaluation sets
    - **Databricks LLM Judges:** Databricks' proprietary AI-assisted judges for evaluating RAG quality. Can be tuned with customer provided examples to increase agreement with human raters.
    - **Metrics:** A set of Databricks-defined metrics for measuring quality/cost/latency of your chain. Most metrics are defined using the output of the Databricks LLM judges.
    - **Customer-defined LLM Judges:** Databricks framework to quickly define custom judges that evaluate business / use-case specific aspects of quality
    - *[Future release]* **Custom metrics:** Provide a user-defined function to run and record its value as an evaluation metric.

M2 release notes & upcoming releases

M2 TLDR

Overall

1. **MLflow Integration:** Most `rag_studio` SDKs are now part of `mlflow`: logging, evaluation, chain parameterization. MLflow Tracing integration is improved.
2. **Developer workflow improvements:** Simplified chain configuration that accepts Dictionaries and YAML.

Evaluation & Databricks LLM Judges

3. **Evaluation is faster & easier to use:** Faster evaluation with progress bar. Accepts single Pandas Dataframe as input and provides single output Dataframe with results. Metrics & data logged to MLflow vs. Delta Tables.

4. **Improved LLM Judges & Metrics docs:** Clarity in metric names and how metrics evaluate chains
5. **New metrics:** Added new metrics such as token count and latency
6. **LLM Judge Tuning:** Few-shot examples can enhance LLM Judge agreement with human ratings
7. **Customer-defined LLM Judges:** Custom judges can be defined to assess use-case specific criteria.

Logging & monitoring

8. **Human Feedback and Monitoring:** New Delta Tables that automatically ETL feedback collected via Review App and Feedback API

Chain compatibility

9. **Chains support External Models & Provisioned Throughput:** Chains can use any Model Serving model, including OpenAI and Provisioned Throughput models with automatic credential provisioning.
10. **Chains support Third-Party APIs:** Chains can utilize third-party APIs with secure credential management through Databricks Secrets.

Review App

11. **Review App SSO:** Review App is accessible to any SSO users even if they aren't a Databricks user

Data Pipelines

12. **Flexible data pipeline:** New data processing pipeline template supports easily testing out different parsing, chunking, embedding strategies - several off the shelf implementations are included.

Details and upcoming releases

Theme	**M2 release**	**Upcoming releases**
Integration with MLflow	RAG Studio's integration with MLflow is significantly improved for both chain developing / logging and chain evaluation. Evaluation: • `rag_eval.evaluate(...)` is now called via `mlflow.evaluate(..., model_type="databricks-rag")` • Evaluation metrics & evaluation data are logged to MLflow Runs (versus a Delta Table) Chain development: • `rag.log_model(...)` is merged into `mlflow.langchain.log_model(...)` SDK • Chain configuration via YAML is now part of MLflow e.g., `rag.RagConfig(...)` moved to `mlflow.models.ModelConfig(...)`	Customer defined LLM judges configuration will be moved from YAML into the native MLflow SDK
One-click RAG Data Pipeline accelerator	Comes with various pre-built parsing / chunking / embedding strategies or build your own. 100% of the code is available to to you, so this accelerator can be fully customized	Improvements based on your feedback

Faster and easier to use evaluation capabilities	Beyond tighter integration with MLflow, Evaluation now: ● Evaluation runs faster ● Evaluation status displayed via a progress bar w/ ETA ● **Breaking change** Improved schema for evaluation input and output ● Accepts a single Pandas Dataframe as input vs. requiring 2 separate Delta Tables ● Evaluation results are available as a single output table vs. split across assessments & metrics, making review of evaluation easier ○ Returned as Pandas Dataframes vs. saved to Delta Tables ● Evaluation Set schema for retrieval ground truth is more flexible and does not require chunk text ● **Breaking change** the config YML syntax has changed to accommodate the new functionality of custom judges and few-shot examples Documentation	Native MLflow-SDK approach for defining custom LLM judges and metrics.
Improved integration with MLflow Tracing	MLflow Tracing provides a way to instrument and visualize the steps in your chain. In this release, MLflow Tracing is more tightly integrated with RAG Studio: ● When running your chain during local development, traces are logged to MLflow and visualized in a Notebook-based UI ● The data from the trace can be accessed via the MLflow SDK	**Breaking change** Step-by-step chain traces are stored in the same MLflow tracing format used during development. This replaces the current trace schema. Traces collected online can be viewed using the same Mlflow UI as in development.
Support for External Models & Provisioned Throughput	Your chain can now use any External Model e.g., OpenAI, etc and any Provisioned Throughput model. Credentials for these models are automatically provisioned when calling `deploy_model()`. Previously, only FM API Pay-Per-Token models were available.	Support
Support for 3rd party APIs	Your chain can now use 3rd party API services e.g., non-Databricks vector databases, etc with secrets stored in Databricks Secrets Manager via environment variables. Note: Doing so requires manually updating the Model Serving endpoint after calling `deploy_model()`.	Native support for using Databricks Secrets via `deploy_model()`.

Fast chain deployment		If using a Databricks defined set of Python packages in your chain, chains now deploy to REST APIs and Review App in 2 - 3 minutes vs. 15 - 20 minutes.
Chain token streaming support		If your LangChain chain supports streaming, your REST API & Review App will also support streaming. Support for token streaming in PyFunc based chains
Human feedback collection & online chain monitoring	Chat requests to a deployed chain's REST API and chat requests via the Review App are available as Delta Tables. These tables are ETL'd versions of the raw Inference Table that contains the same data, but has a raw schema and duplicate rows. 2 tables are generated: ● `request_log`: Keyed to request_id, contains one row for every trace. A trace represents one turn of conversation in a chat and is linked to other traces in that conversation via converastion_id. Contains: ○ Request ○ Response ○ Step-by-step chain trace ● `assessment_log`: Keyed to request_id, contains one row for every piece of human feedback collected via the Review App or the Feedback API, including feedback for live chats & offline review of traces. Contains: ○ Thumbs up / down + why ■ Free text & select-many choices ○ Corrected bot responses *Note: Due to how the ETL job is scheduled, rows in these tables can take up to ~2 hours to appear. Raw logs appear in the Inference Table in ~15 minutes.* Documentation	**Breaking change** Improvements to the assessment log schema to enable the data to be more easily used as input to tune the LLM Judge models and create Evaluation Sets.
Review App	Access to the Review App can be granted to any user in your company's SSO even if they do not have access to your Databricks Workspace.	Support for collecting 👍👎feedback on individual chunks, which can be used to

	To use this feature, your account admin must enable SCIM. Documentation	generate Evaluation Sets and tune the Chunk Relevance LLM judge. Improvements to the UX of Review App that make reviewing simpler / faster for stakeholders.
LLM Judges & Metrics documentation	Improved the clarity of the metric & LLM Judge names. Added more clear documentation on what each metric & LLM judge evaluates and the data used. Documentation	How to guide & sample code explaining techniques to try based on the results of your evaluation.
New Metrics	Added token count and latency metrics.	Fully custom metrics e.g., use a user-defined function as a metric
Increasing the quality of Databricks provided LLM Judges	Improved support for providing few-shot examples to the LLM Judges to tune their agreement with human raters. `evaluate()` now accepts a Dataframe of examples that follows the same schema as the judges output, making it easier to modify incorrect judge scores / tag high quality judge scores and provide them as examples to improve the judge quality. Documentation	Improvements in the quality of the LLM Judges e.g., LLM judges are more likely to agree with human rated assessments.
Customer defined LLM judges	Added support for customer defined LLM judges. Each judge is defined by: ● Aninstruction (e.g., “asses if the response follows my company tone of voice, which is xyz”) ● A set of trace fields for the judge to review (e.g., request, response, ground truth, etc) ● Optionally, 1+ examples of “good” and “bad” responses to tune the judge’s quality Documentation	Customer defined LLM judges configuration will be moved from YAML into the native MLflow SDK
Improvements to the chain developer workflow	Various improvements: ● Chain configuration can be passed a Dictionary vs. only as a YAML file ● Single wheel that is available on PyPi	`dbutils` use in the `chain.py` is automatically removed vs. the user needing to comment this code out before logging the model `delete_deployment()` API to turn off a Chain deployment
Improved	Support for custom PIP dependencies with a LangChain based	Support for logging PyFunc chains:

support for other types of Chains	chain	<ul><li>Rather than requiring PyFunc chains to be pickled/serialized (which often fails) - PyFunc-based Chains are captured as Code + Config</li><li>Support for non-Dataframe based PyFunc signature e.g., `predict(param1, param2)` vs. `predict(model_input : dataframe)`.</li></ul>PyFunc chains support YAML/Dictionary based configuration. **Breaking change** More flexible schema support for chains - you can now use custom parameters in your chain's input / output signature. Previously, you could only use one pre-defined signature for your chain.
Improved compatibility w/ security postures		Support for PrivateLink customers to use Model Serving with Vector Search securely Support for Azure Storage Firewall customers to use Inference Tables

Table of contents

RAG Studio documentation

Creating, logging & deploying chains

Logging a chain is the basis of your development process. Logging captures a "point in time" of the chain's code and configuration so you can evaluate the quality of those settings. With RAG Studio, there are 2 ways you can create & log a chain, both based on MLflow:

Approach	Description	Pros	Cons
Serialization-based MLflow logging	The chain's code and current state in the Python environment is serialized to disk, often using libraries such as pickle or joblib. The Python environment (e.g., pip packages) is captured as a list of packages. When the chain is deployed, the Python environment is restored, and the serialized object is loaded into memory so it can be invoked when the endpoint is called.	+ log_model(...) can be called from the same notebook as where the model is defined	- Original code is not available - All used libraries/objects must support serialization. - Requires translating the chain's inputs & outputs to/from a Pandas DataFrame when calling the logged or deployed MLflow model.

Code-based MLflow logging	The chain's code is captured as a Python file. The Python environment (e.g., pip packages) is captured as a list of packages. When the chain is deployed, the Python environment is restored, and the chain's code is executed to load the chain into memory so it can be invoked when the endpoint is called.	+ Overcomes inherent limitations of serialization - it is not supported by many popular GenAI libraries. + Saves a copy of the original code for later reference + No need to restructure your code into a single object that can be serialized. + If using PyFunc, does NOT translating the chain's inputs & outputs to/from a Pandas DataFrame when calling the logged or deployed MLflow model.	- log_model(...) must be called from a *different* notebook than the chain's code (called a Driver Notebook). - *(current limitation)* The Python environment of the Driver Notebook is captured when logging - if this is different than the environment required by the chain's code, you must manually specicify the Python packages when logging

For developing RAG applications, Databricks suggests approach (2) code-based logging. RAG Studio's documentation assumes you are using code-based logging & has not been tested with serialization-based logging.

1a. Creating a chain

Note: In the next release of RAG Studio, we will allow for more flexible input / output schemas.

In RAG Studio, the chain's code, which we colloquially refer to as `chain.py`, is your development space - you can use any package, library, or technique. The *only* requirement is that your chain comply with a given input / output schema. Why? To use your chain in the Review App & to evaluate your chain with Evaluation Suite, we must know how to send your chain user queries & how to find the chain's response back to the user.

Input schema

Your chain must accept an array of OpenAI-formatted messages as a `messages` parameter. This is the same input schema that your chain's REST API will accept once deployed.

Python
```python
question = {
    "messages": [
        {
            "role": "user",
            "content": "question 1",
        },
        {
            "role": "assistant",
            "content": "answer 1",
        },
        {
            "role": "user",
            "content": "new question!!",
        },
    ]
}
```

Output schema

Your chain must return a single string value. To do this in LangChain, use `StrOutputParser()` as your final chain step.

```python
chain = (
    {
        "user_query": itemgetter("messages")
        | RunnableLambda(extract_user_query_string),
        "chat_history": itemgetter("messages") | RunnableLambda(extract_chat_history),
    }
    | RunnableLambda(fake_model)
    | StrOutputParser()
)
```

1b. Parameterizing the chain

RAG Studio supports parameterizing your chains - this allows you to quickly iterate on quality related parameters (such as the prompt or retriever configuration) while holding the chain's code constant.

Parameterization is flexible and works with any key:value pairs you define. In the RAG Cookbook, we provide a suggested set of key:value pairs. Parameters can be stored inside a YAML file or a Python Dictionary. Databricks suggests using a Dictionary during development and then converting your Dictionary to a YAML for production deployment via CI/CD.

How to parameterize your chain

1. Define a key:value parameter in a Dictionary or YAML file

Python Dictionary

```Python
config_dict = {"sample_param": "this is the sample parameter that can be changed! this could
be a prompt, a retrieval setting, or ..."}
```

YAML file

```Unset
sample_param: "Hello from the YAML config!"
```

2. In your `chain.py`, add 1 of the following 2 lines of code to access your parameters.

```Python
import mlflow
# do one of the following
model_config = mlflow.models.ModelConfig(development_config=config_dict)
model_config = mlflow.models.ModelConfig(development_config='path_to_yaml_file.yaml')

# model_config is a Python Dictionary with your parameters.
value = model_config.get('sample_param')
```

2. Logging the chain

When using code-based MLflow logging, you log, evaluate, and deploy new versions of your `chain.py` using a separate notebook, which we call the Driver Notebook.

```Python
import mlflow

code_path = "/Workspace/Users/first.last/chain.py"
config_path = "/Workspace/Users/first.last/config.yml"

input_example = {
    "messages": [
        {
            "role": "user",
            "content": "What is Retrieval-augmented Generation?",
        }
    ]
}

with mlflow.start_run():
  logged_chain_info = mlflow.langchain.log_model(
    lc_model=code_path, # Passing a file path here tells `log_model(...)` to use code-based
logging vs. serialization-based logging.
    model_config=config_path, # If using parameterization, this is the configuration of the
chain - either a Python dictionary or path to a YAML file
    artifact_path="chain", # Any user provided string, used as the path inside the MLflow
model where artifacts are stored
    input_example=input_example, # Must be a valid input to your chain
    example_no_conversion=True, # Required to enable
    extra_pip_requirements = ["databricks-rag-studio==0.2.0"] # Required during PrPr to tell
Databricks, this is a RAG Studio compatible model
  )

print(f"MLflow Run: {logged_chain_info.run_id}")
print(f"Model URI: {logged_chain_info.model_uri}")

# model_uri can also be used to do evaluations of your RAG
model_uri=logged_chain_info.model_uri
```

Note, if you are using parameterization, the resulting MLflow model uses the configuration that was passed when logging. The `development_config` is automatically overridden.

Verify the model is logged correctly by loading the chain in MLflow and calling invoke.

```
Python
model = mlflow.langchain.load_model(model_uri)
model.invoke(example)
```

You can repeat this process on different configurations of the chain by modifying the code and config.

Step 3. Evaluating the chain

Normally, at this point in your dev loop, you would evaluate the chain. See the Evaluation Suite section for how to do this.

Step 4. Deploying the chain to the Review App & as a REST API

RAG Studio uses Databricks Model Serving to deploy your chain.

During development, you deploy your chain to collect feedback from expert stakeholders. During production, you deploy your chain to make it available as a REST API that can be integrated into your user-facing application.

With RAG Studio, a single `deploy_model(...)` command creates a scalable, production-ready deployment that works for either of these use cases.

Before you deploy your model, you must register your logged model (from step 2) to the Unity Catalog:

```
Python
import mlflow

mlflow.set_registry_uri("databricks-uc")

catalog_name = "test_catalog"
schema_name = "schema"
model_name = "chain_name"

model_fqn = catalog_name + "." + schema_name + "." + model_name
uc_model_info = mlflow.register_model(model_uri=logged_chain_info.model_uri, name=model_fqn)
```

Then, you can deploy your model:

```
Python
from databricks.rag_studio import deploy_model
from mlflow.utils import databricks_utils as du

deployment = deploy_model(model_fqn, uc_model_info.version)

# query_endpoint is the URL that can be used to make queries to the app
deployment.query_endpoint

# Copy deployment.rag_app_url to browser and start interacting with your RAG application.
deployment.rag_app_url

# Temporary helper function during PrPr to pretty print URLs
def parse_deployment_info(deployment_info):
  browser_url = du.get_browser_hostname()
  message = f"""Deployment of {deployment_info.model_name} version
{deployment_info.model_version} initiated.  This can take up to 15 minutes and the Review App
& REST API will not work until this deployment finishes.

  View status: https://{browser_url}/ml/endpoints/{deployment_info.endpoint_name}
  Review App: {deployment_info.rag_app_url}"""
  return message

parse_deployment_info(deployment)
```

Calling `deploy_model(...)` does the following:

TODO: Add information on the automated authentication of Databricks services used in your chain

1. Enables the Review App for your chain
   - Allows your expert stakeholders can chat with the chain & give feedback via a web UI
2. Creates REST APIs for your chain that can be integrated into your user-facing application
   - invoke endpoint to get responses from the chain
   - feedback to pass feedback from your front end
3. Complete logging of every request to the Review App or REST API e.g., input/output and intermediate trace via Inference Tables
   - 3 Delta Tables are created for each deployment
     a. Raw JSON payloads `{catalog_name}.{schema_name}.rag_studio_{model_name}_payload`
     b. Formatted request/response & MLflow Traces `{catalog_name}.{schema_name}.rag_studio_{model_name}_payload_request_logs`
     c. Formatted feedback, as provided in the Review App or via Feedback API, for each request `{catalog_name}.{schema_name}.rag_studio_{model_name}_payload_assessment_logs`

Note: *It can take up to 15 minutes to deploy. Raw JSON payloads take 10 - 30 minutes to arrive, and the formatted logs are processed from the raw payloads every ~hour.*

Listing deployed applications

To see all RAG chains that you have deployed you can:

Python
```python
from databricks.rag_studio import list_deployments, get_deployments

# Get the deployment for specific model_fqn and version
deployment = get_deployments(model_name=model_fqn, model_version=model_version.version)

deployments = list_deployments()
# Print all the current deployments
deployments
```

How deployments work

TODO: Add info about how multiple versions are used. Talk about feedback endpoint.

Inference Tables

TODO: Add schemas.
Inference Tables capture the complete logging of every request to the Review App or REST API e.g., input/output and intermediate trace via Inference Tables. There are 3 Delta Tables are created for each deployment

1. Raw JSON payloads `{catalog_name}.{schema_name}.rag_studio_{model_name}_payload`
2. Formatted request/response & MLflow Traces
   `{catalog_name}.{schema_name}.rag_studio_{model_name}_payload_request_logs`
3. Formatted feedback, as provided in the Review App or via Feedback API, for each request
   `{catalog_name}.{schema_name}.rag_studio_{model_name}_payload_assessment_logs`

Feedback API

TODO: Add documentation for the input / output schema of the /feedback/ endpoint on the deployed model serving endpoint.

Evaluation Suite documentation

The RAG Evaluation Suite enables developers to quickly and reliably evaluate the quality, cost and latency of their Generative AI application. The Evaluation Suite's capabilities are unified between the development, staging, and production phases of the LLMops life cycle.

The Evaluation Suite is part of our broader RAG Studio offering that is designed to help developers deploy high-quality Generative AI applications. High quality apps are those apps where the output is evaluated to be accurate, safe, and governed.

How to run Evaluation

To run evaluation, you need to define an Evaluation Set. At a high level, an Evaluation Set contains a set of representative requests that users would make to your chain. For example, "what is rag studio?", etc. The intent is that you can reason about the quality/cost/latency of your chain by having corresponding metrics computed on the Evaluation Set.

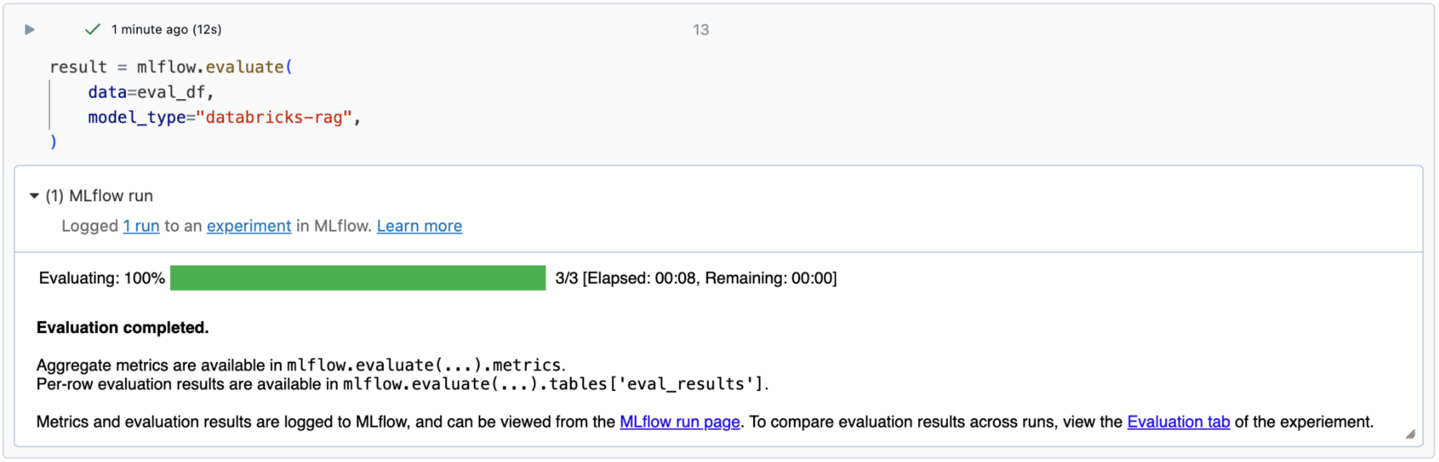
Evaluation Harness is integrated with MLflow and is run by calling `mlflow.evaluate(..., model_type="databricks-rag")`. The parameters depend on your approach to using evaluate. Before you call `evaluate`, you need to install the RAG Studio Python wheel in your notebook.

The `mlflow.evaluate()` method is designed to be called from within an existing MLflow Run. If evaluate is called from outside an MLflow Run, evaluate will start a new Run.

```python
%pip install databricks-rag-studio
dbutils.library.restartPython()
```

```
import mlflow
import pandas as pd

eval_df = pd.DataFrame(...) # See below for the schema of the eval dataset

# Puts the evaluation results in the current Run, alongside the logged model parameters
with mlflow.start_run():
        logged_model_info = mlflow.langchain.log_model(...)
        mlflow.evaluate(data=eval_df, model=logged_model_info.model_uri,
                        model_type="databricks-rag")

# Starts a new MLflow Run
mlflow.evaluate(data=eval_df, model=logged_model_info.model_uri, model_type="databricks-rag")
```

You will see output like this after running the `evaluate` command.



```
result = mlflow.evaluate(
    data=eval_df,
    model_type="databricks-rag",
)
```

```
(1) MLflow run
    Logged 1 run to an experiment in MLflow. Learn more

Evaluating: 100% 3/3 [Elapsed: 00:08, Remaining: 00:00]

Evaluation completed.

Aggregate metrics are available in mlflow.evaluate(...).metrics.
Per-row evaluation results are available in mlflow.evaluate(...).tables['eval_results'].

Metrics and evaluation results are logged to MLflow, and can be viewed from the MLflow run page. To compare evaluation results across runs, view the Evaluation tab of the experiement.
```

Approaches to running `evaluate`

The Evaluation Harness supports two options to providing your chain's outputs in order to generate quality/cost/latency Metrics.

Option	Description	When to use
1	**Eval Harness runs a chain on your behalf.** Pass a reference to the chain itself so Evaluation Harness can generate the outputs on your behalf	1. Your chain is logged using MLflow w/ MLflow Tracing enabled 2. Your chain is available as a Python function in your local notebook
2	**Run chain yourself, pass outputs to Eval Harness.** Run the chain being evaluated yourself, capturing the chain's outputs and passing the outputs as a Pandas DataFrame	1. Your chain is developed outside of Databricks 2. You want to evaluate outputs from a chain already running in production 3. You are testing different evaluation / LLM Judge configurations and your chain doesn't produce deterministic outputs (e.g,. LLM has

		a high temperature)

Important note: All inputs provided to the Evaluation Harness will be passed through to the output table. If an input was generated by running your chain vs. provided directly, that generated output will be passed through.

Option 1: Eval Harness runs a chain on your behalf

```python
%pip install databricks-rag-studio pandas
dbutils.library.restartPython()

import mlflow
import pandas as pd

###
# Evaluation Harness call signature
###
evaluation_results = mlflow.evaluate(
    data=eval_set_df,  # Pandas Dataframe with just the Evaluation Set - see Eval Harness Input Schema for details
    model=model,  # Reference to the MLflow model
    model_type="databricks-rag",
)

###
# There are 4 options for passing a model.
####

#### Option 1. Reference to a Unity Catalog registered model
model = "models:/catalog.schema.model_name/1"  # 1 is the version number

#### Option 2. Reference to a MLflow logged model in the current MLflow Experiment
model = "runs:/6b69501828264f9s9a64eff825371711/chain"
# `6b69501828264f9s9a64eff825371711` is the run_id, `chain` is the artifact_path that was
# passed when calling mlflow.xxx.log_model(...).
# This value can be accessed via `model_info.model_uri` if you
# called model_info = mlflow.xxx.log_model(), where xxx is `langchain` or `pyfunc`.

#### Option 3. A PyFunc model that is loaded in the Notebook
model = mlflow.pyfunc.load_model(...)

#### Option 4. A local function in the Notebook
def model_fn(model_input):
  # do stuff
  response = 'the answer!'
  return response

model = model_fn
```

```
###
# `data` is a Pandas DataFrame with your Evaluation set
# See Eval Harness Input Schema for details of the schema - provided here is a
# simple example.
####

# These examples are provided just to show the schema.
# You do NOT have to start from a Dictionary - you can use any existing Pandas or
# Spark DataFrame with this schema (see below for example).
bare_minimum_eval_set_schema = [
    {
        "request": "What is the difference between reduceByKey and groupByKey in Spark?",
    }]

complete_eval_set_schema = [
    {
        "request_id": "your-request-id",
        "request": "What is the difference between reduceByKey and groupByKey in Spark?",
        "expected_retrieved_context": [
            {
                # In `expected_retrieved_context`, `content` is optional, and does not
deliver any additional functionality if provided.
                "content": "Answer segment 1 related to What is the difference between
reduceByKey and groupByKey in Spark?",
                "doc_uri": "doc_uri_2_1",
            },
            {
                "content": "Answer segment 2 related to What is the difference between
reduceByKey and groupByKey in Spark?",
                "doc_uri": "doc_uri_2_2",
            },
        ],
        "expected_response": "There's no significant difference.",
    }]

#### Convert Dictionary to a Pandas DataFrame
eval_set_df = pd.DataFrame(bare_minimum_eval_set_schema)

#### Use a Spark DataFrame
import numpy as np
spark_df = spark.table("catalog.schema.table") # or any other way to get a Spark DataFrame
eval_set_df = spark_df.toPandas()
```

Option 2: Run chain yourself, pass outputs to Eval Harness.

Python

```python
%pip install databricks-rag-studio pandas
dbutils.library.restartPython()

import mlflow
import pandas as pd

###
# Evaluation Harness call signature
###
evaluation_results = mlflow.evaluate(
    data=eval_set_with_chain_outputs_df,  # Pandas Dataframe with the Evaluation Set AND
Chain outputs - see Eval Harness Input Schema for details
    model_type="databricks-rag",
)


###
# `data` is a Pandas DataFrame with the Evaluation Set AND Chain outputs
# See Eval Harness Input Schema for details of the schema - provided here is a
# simple example.
####

# These examples are provided just to show the schema.
# You do NOT have to start from a Dictionary - you can use any existing Pandas or
# Spark DataFrame with this schema (see below for example).

# Bare minimum data
bare_minimum_input_schema = [
    {
        "request": "What is the difference between reduceByKey and groupByKey in Spark?",
        "response": "reduceByKey aggregates data before shuffling, whereas groupByKey
shuffles all data, making reduceByKey more efficient.",
    }]

complete_input_schema  = [
    {
        "request_id": "your-request-id",
        "request": "What is the difference between reduceByKey and groupByKey in Spark?",
        "expected_retrieved_context": [
            {
                # In `expected_retrieved_context`, `content` is optional, and does not
deliver any additional functionality if provided.
                "content": "Answer segment 1 related to What is the difference between
reduceByKey and groupByKey in Spark?",
                "doc_uri": "doc_uri_2_1",
            },
            {
                "content": "Answer segment 2 related to What is the difference between
reduceByKey and groupByKey in Spark?",
                "doc_uri": "doc_uri_2_2",
            },
        ],
        "expected_response": "There's no significant difference.",
        "response": "reduceByKey aggregates data before shuffling, whereas groupByKey
shuffles all data, making reduceByKey more efficient.",
        "retrieved_context": [
            {
                # In `retrieved_context`, `content` is optional, but DOES deliver additional
functionality if provided (the Databricks Context Relevance LLM Judge will run to check the
`content`'s relevance to the `request`).
                "content": "reduceByKey reduces the amount of data shuffled by merging values
before shuffling.",
                "doc_uri": "doc_uri_2_1",
            },
            {
                "content": "groupByKey may lead to inefficient data shuffling due to sending
all values across the network.",
                "doc_uri": "doc_uri_6_extra",
            },
        ],
    }]

#### Convert Dictionary to a Pandas DataFrame
eval_set_with_chain_outputs_df = pd.DataFrame(bare_minimum_input_schema)

#### Use a Spark DataFrame
import numpy as np
spark_df = spark.table("catalog.schema.table") # or any other way to get a Spark DataFrame
eval_set_with_chain_outputs_df = spark_df.toPandas()
```

Evaluation Harness Input Schema

❌ Optional, does not enable additional functionality if provided
✖️ Optional, but if provided, more Databricks LLM Judges and Metrics can be computed.
✅ Required

			Required?	
Key	**Type**	**Description**	**Option 1** **Eval Harness runs a chain on your behalf.**	**Option 2** **Run chain yourself, pass outputs to Eval Harness.**
`request_id`	`string`	Unique identifier of this row in the evaluation set.	❌	❌
`request`	`string`	Input to the chain to evaluate e.g., the user's question / query such as "What is RAG?"	✅	✅
`expected_retrieved`	`array**`	An array of objects containing the	✖️	✖️

_context		expected retrieved context for the request.		
expected_response	string	The ground-truth (i.e., correct) answer to `request`.	✖	✖
response	string	The response generated by the chain being evaluated.	n/a - generated by Evaluation Harness from your chain	✅
retrieved_context	array**	The retrieval results generated by the retriever in the chain being evaluated. If multiple retrieval steps are in the chain, this should be the retrieval results that were put into the LLM's prompt.		✖
trace	[MLflow Trace]	MLflow Trace with the Chain's outputs.		✖

** `expected_retrieved_context` and `retrieved_context` schema

The `expected_retrieved_context` and `retrieved_context` arrays expect each array element to be a dictionary with the following keys:

Key	**Type**	**Description**	**expected_retrieved_context**	**retrieved_context**
content	string	The contents of the retrieved context. Can be any string irregardless of formatting e.g., HTML, Plain Text, Markdown, etc.	❌	✖
doc_uri	string	Unique identifier (URI) of the parent document where the chunk came from.	✅	✅

Evaluation outputs

Evaluation Harness produces several outputs:

1. **Aggregated metric values across the entire Evaluation Set**
   a. Average numerical result of each metric
2. **Data about each question in the Evaluation Set**
   a. In the same schema as the Evaluation Input
      i. Inputs sent to the chain
      ii. All chain generated data used in evaluation e.g., response, retrieved_context, trace, etc
   b. Numeric result of each metric e.g., 1 or 0, etc
   c. Ratings & rationales from each Databricks and Customer-defined LLM judge

These outputs are available in 2 locations:

1. Stored inside the MLflow Run
2. Returned as dataframes / dictionaries from the call to `mlflow.evaluate()`

For more details on the metrics computed by Evaluation Harness, please see the LLM Judges & Metrics section. The outputs of the Evaluation Harness are the same between Option 1 and Option 2, however, the data that appears within the outputs varies based on the inputs you provide because certain judges and metrics require specific inputs.

Reviewing outputs

You can either view the evaluation results in your notebook as raw data or inside the MLflow UI. Note that the data is identical, so which view you use is a matter of your preference.

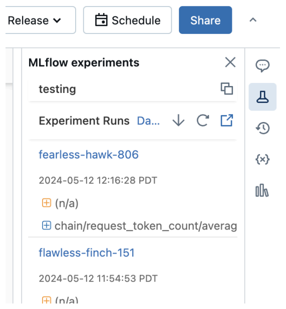
Using the notebook

Python
```python
%pip install databricks-rag-studio pandas
dbutils.library.restartPython()

import mlflow
import pandas as pd

###
# Run evaluation using one of the approaches above
###
evaluation_results = mlflow.evaluate(..., model_type="databricks-rag")

###
# Access aggregated metric values across the entire Evaluation Set
###
metrics_as_dict = evaluation_results.metrics
metrics_as_pd_df = pd.DataFrame([evaluation_results.metrics])

# Sample usage
print(f"The average precision of the retrieval step is:
{metrics_as_dict['retrieval/ground_truth/document_precision/average']}")


###
# Access data about each question in the Evaluation Set
###

per_question_results_df = evaluation_results.tables['eval_results']

# Show info about responses that are not grounded
per_question_results_df[per_question_results_df["response/llm_judged/groundedness_rating"] ==
False].display()
```

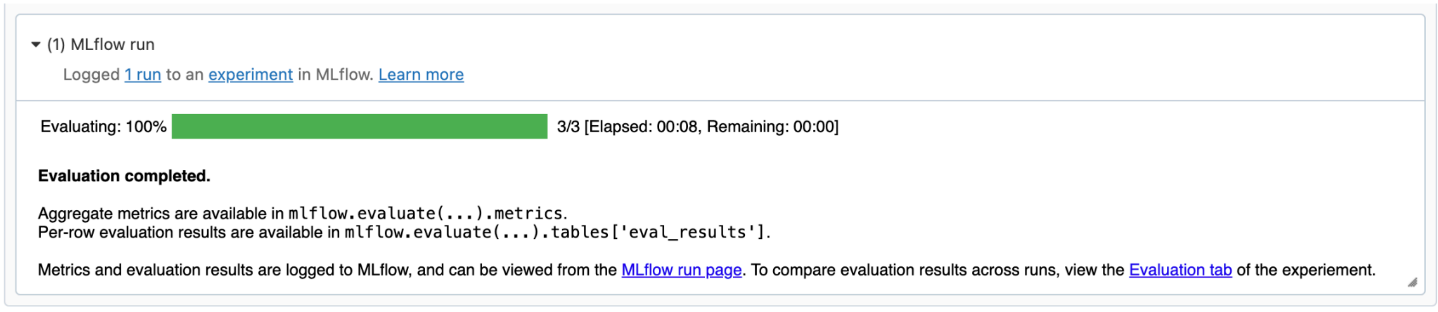
Using the MLflow UI

All data is stored inside an MLflow Run inside your Notebook's experiment. You can access this by either

1. Entering the experiment through the experiment button at the upper right corner of your Notebook



2. Using the links at the bottom of the cell where you ran `evaluate(...)`



▾ (1) MLflow run
Logged 1 run to an experiment in MLflow. Learn more

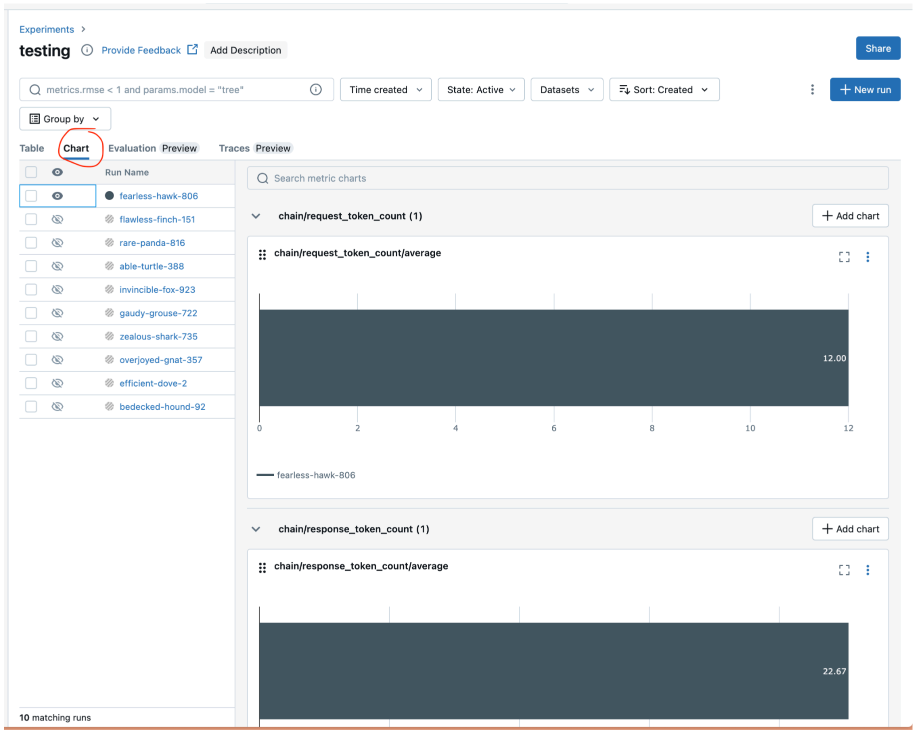
```
Evaluating: 100% 3/3 [Elapsed: 00:08, Remaining: 00:00]
```

Evaluation completed.

Aggregate metrics are available in `mlflow.evaluate(...).metrics`.
Per-row evaluation results are available in `mlflow.evaluate(...).tables['eval_results']`.

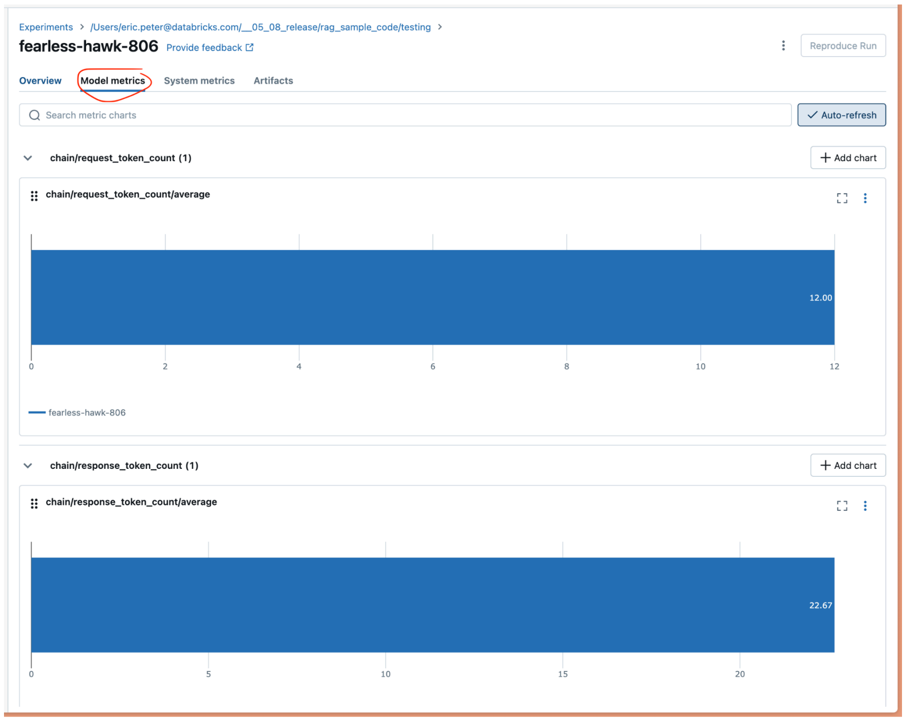
Metrics and evaluation results are logged to MLflow, and can be viewed from the MLflow run page. To compare evaluation results across runs, view the Evaluation tab of the experiement.

To see the aggregated metric values across the entire Evaluation Set, you can either

1. From the Experiment page, click on the Metrics tab. This allows you to visualize the metrics for this run of `evaluate(...)` and compare with past runs of `evaluate(...)`.
   a. 

Run Name
fearless-hawk-806
flawless-finch-151
rare-panda-816
able-turtle-388
invincible-fox-923
gaudy-grouse-722
zealous-shark-735
overjoyed-gnat-357
efficient-dove-2
bedecked-hound-92

2. From the Run's page, you can view the raw metric values from the Overview tab or visualize the values as graphs in the Metrics tab


Experiments > /Users/eric.peter@databricks.com/__05_08_release/rag_sample_code/testing >

fearless-hawk-806 Provide feedback

Overview | Model metrics | System metrics | Artifacts

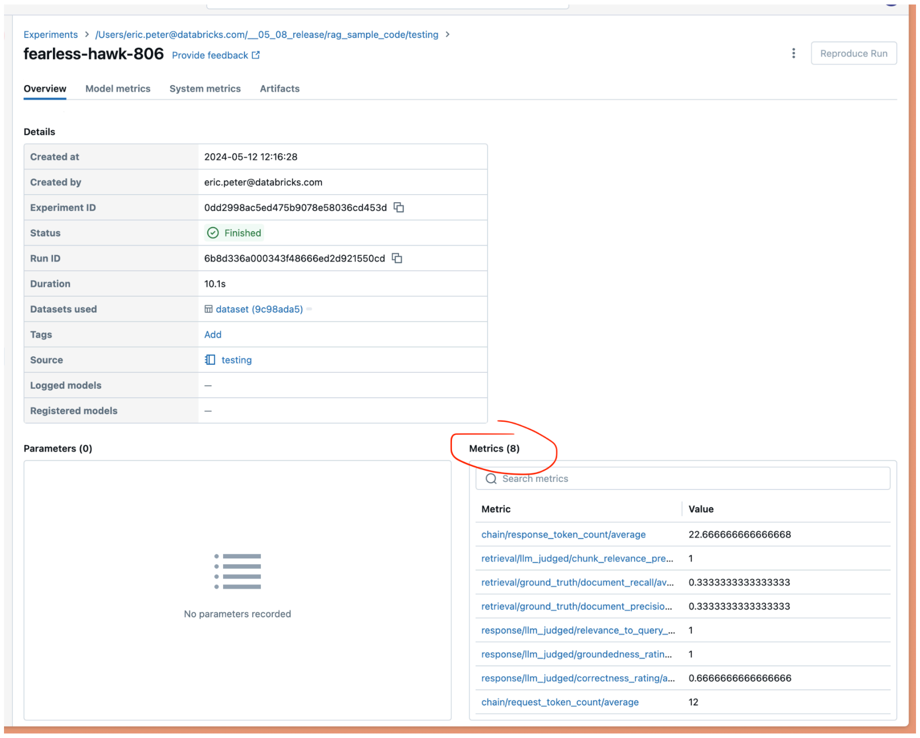
Search metric charts | Auto-refresh

chain/request_token_count (1) | Add chart

chain/request_token_count/average

chain/response_token_count (1) | Add chart

chain/response_token_count/average


Experiments > /Users/eric.peter@databricks.com/__05_08_release/rag_sample_code/testing >

fearless-hawk-806 Provide feedback

Overview | Model metrics | System metrics | Artifacts

Details

Created at	2024-05-12 12:16:28
Created by	eric.peter@databricks.com
Experiment ID	0dd2998ac5ed475b9078e58036cd453d
Status	Finished
Run ID	6b8d336a000343f48666ed2d921550cd
Duration	10.1s
Datasets used	dataset (9c98ada5)
Tags	Add
Source	testing
Logged models	—
Registered models	—

Parameters (0)

No parameters recorded

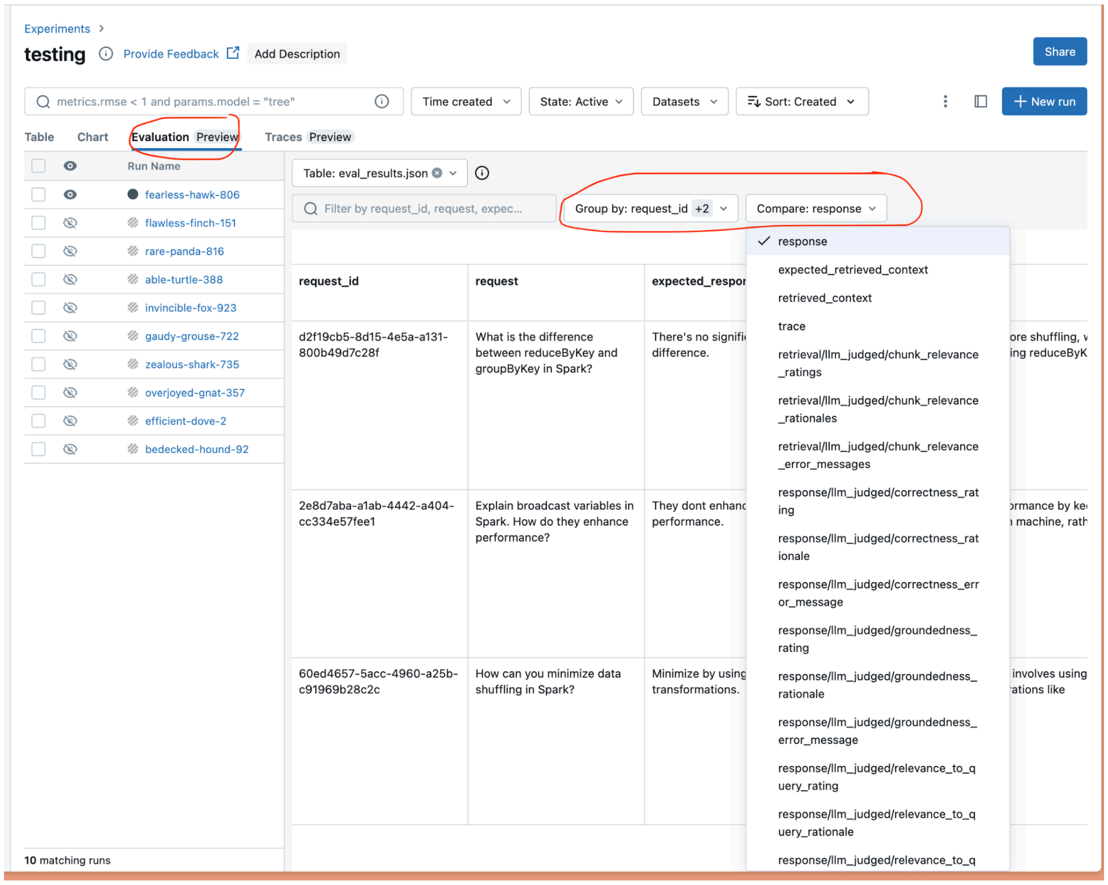
Metrics (8)

Search metrics

Metric	Value
chain/response_token_count/average	22.666666666666668
retrieval/llm_judged/chunk_relevance_pre...	1
retrieval/ground_truth/document_recall/av...	0.3333333333333333
retrieval/ground_truth/document_precisio...	0.3333333333333333
response/llm_judged/relevance_to_query_...	1
response/llm_judged/groundedness_ratin...	1
response/llm_judged/correctness_rating/a...	0.6666666666666666
chain/request_token_count/average	12

To access data about each question in the Evaluation Set, you can either

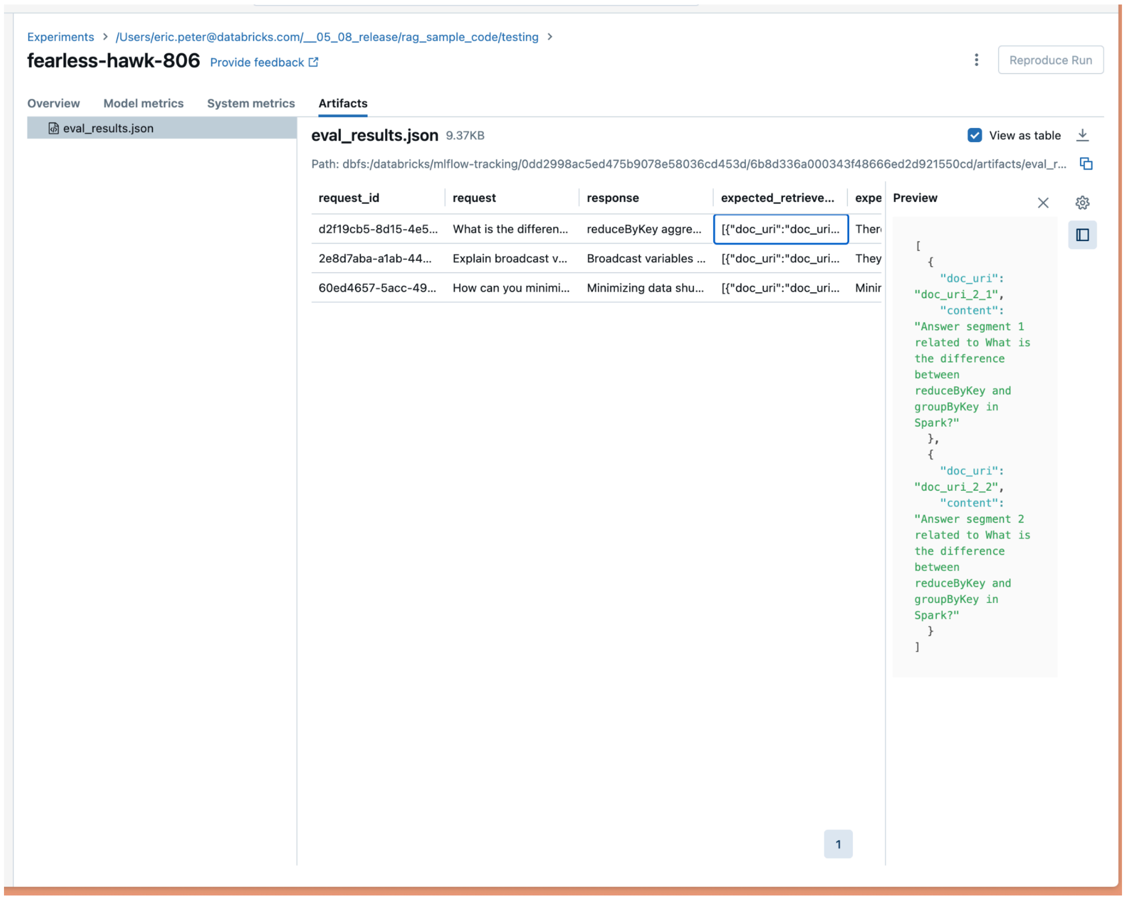
1. From the Experiment page, click on the Evaluation tab. Here you can see each question in your evaluation set, choosing with columns of data to view by manipulating the indicated dropdowns. You can also compare across evaluation runs.



Experiments >
testing
Provide Feedback
Add Description
Share
metrics.rmse < 1 and params.model = "tree"
Time created
State: Active
Datasets
Sort: Created
New run
Table
Chart
Evaluation Preview
Traces Preview
Run Name
fearless-hawk-806
flawless-finch-151
rare-panda-816
able-turtle-388
invincible-fox-923
gaudy-grouse-722
zealous-shark-735
overjoyed-gnat-357
efficient-dove-2
bedecked-hound-92
10 matching runs
Table: eval_results.json
Filter by request_id, request, expec...
Group by: request_id +2
Compare: response
✓ response
expected_retrieved_context
retrieved_context
trace
retrieval/llm_judged/chunk_relevance_ratings
retrieval/llm_judged/chunk_relevance_rationales
retrieval/llm_judged/chunk_relevance_error_messages
response/llm_judged/correctness_rating
response/llm_judged/correctness_rationale
response/llm_judged/correctness_error_message
response/llm_judged/groundedness_rating
response/llm_judged/groundedness_rationale
response/llm_judged/groundedness_error_message
response/llm_judged/relevance_to_query_rating
response/llm_judged/relevance_to_query_rationale
response/llm_judged/relevance_to_q

request_id	request	expected_respor
d2f19cb5-8d15-4e5a-a131-800b49d7c28f	What is the difference between reduceByKey and groupByKey in Spark?	There's no signifi difference.
2e8d7aba-a1ab-4442-a404-cc334e57fee1	Explain broadcast variables in Spark. How do they enhance performance?	They dont enhanc performance.
60ed4657-5acc-4960-a25b-c91969b28c2c	How can you minimize data shuffling in Spark?	Minimize by using transformations.

2. From the Run's page, you can view the raw results from the Artifacts tab by clicking on the `eval_results.json` artifact.



Experiments > /Users/eric.peter@databricks.com/__05_08_release/rag_sample_code/testing >
fearless-hawk-806
Provide feedback
Reproduce Run
Overview
Model metrics
System metrics
Artifacts
eval_results.json
eval_results.json 9.37KB
View as table
Path: dbfs:/databricks/mlflow-tracking/0dd2998ac5ed475b9078e58036cd453d/6b8d336a000343f48666ed2d921550cd/artifacts/eval_r...
Preview
1

request_id	request	response	expected_retrieve...	expe
d2f19cb5-8d15-4e5...	What is the differen...	reduceByKey aggre...	[{"doc_uri":"doc_uri...	Ther
2e8d7aba-a1ab-44...	Explain broadcast v...	Broadcast variables ...	[{"doc_uri":"doc_uri...	They
60ed4657-5acc-49...	How can you minimi...	Minimizing data shu...	[{"doc_uri":"doc_uri...	Minir

```
[
  {
    "doc_uri":
"doc_uri_2_1",
    "content":
"Answer segment 1
related to What is
the difference
between
reduceByKey and
groupByKey in
Spark?"
  },
  {
    "doc_uri":
"doc_uri_2_2",
    "content":
"Answer segment 2
related to What is
the difference
between
reduceByKey and
groupByKey in
Spark?"
  }
]
```

Evaluation Harness Output Schema

The Evaluation Harness output schema is a combination of the Input Schema and the schema defined by each metric / LLM judge. The field names are aggregated here for reference. For details of each field, please refer to the linked section..

1. **Aggregated metric values across the entire Evaluation Set**
   - Defined in each metric's definition
     - `retrieval/ground_truth/document_precision/average`
     - `retrieval/ground_truth/document_recall/average`
     - `retrieval/llm_judged/chunk_relevance_precision/average`
     - `response/llm_judged/relevance_to_query_rating/average`
     - `response/llm_judged/correctness_rating/average`
     - `response/llm_judged/groundedness_rating/average`
     - `chain/request_token_count/average`
     - `chain/response_token_count/average`
2. **Data about each question in the Evaluation Set**
   - Defined in the Evaluation Harness Input schema
     - `request_id`
     - `request`
     - `response`
     - `expected_retrieved_context`
     - `expected_response`
     - `retrieved_context`
     - `trace`
   - Defined in each metric's definition
     - `response/llm_judged/groundedness_rating`
     - `response/llm_judged/groundedness_rationale`
     - `response/llm_judged/groundedness_error_message`
     - `response/llm_judged/correctness_rating`
     - `response/llm_judged/correctness_rationale`
     - `response/llm_judged/correctness_error_message`
     - `response/llm_judged/relevance_to_query_rating`
     - `response/llm_judged/relevance_to_query_rationale`
     - `response/llm_judged/relevance_to_query_error_message`
     - `retrieval/llm_judged/chunk_relevance_ratings`
     - `retrieval/llm_judged/chunk_relevance_rationales`
     - `retrieval/llm_judged/chunk_relevance_error_messages`
     - `retrieval/llm_judged/chunk_relevance_precision`
     - `retrieval/ground_truth/document_precision`
     - `retrieval/ground_truth/document_recall`
     - `chain/request_token_count`
     - `chain/response_token_count`
     - `chain/total_input_token_count`
     - `chain/total_output_token_count`
     - `chain/total_token_count`
     - `chain/latency_seconds`

Curating Evaluation Sets

TODO: Add in more details about how to use the Review App logs to curate evaluation sets.

Review App

The Review App enables you to collect feedback from your expert stakeholders on your application. This feedback can be used to:

1. Turned into a Dashboard to understand the quality of a specific version of your app
2. Curated into Evaluation Sets to measure the quality of your application
3. Curated into few-shot examples to customize the Databricks LLM Judges

There are 3 ways to collect feedback via the Review App:

1. Have expert stakeholders directly chat with the application bot and provide feedback on those conversations
2. Have expert stakeholders provide feedback on historical logs from other users
3. [Coming soon] Have expert stakeholders provide feedback on any curated traces / chian outputs

Setting permissions on the review app

All options support the same permissions model. The Review App can be made accessible to **any** user within your organization's SSO provider, even if they do not have a Databricks workspace account. In order to use this feature, you must have your Databricks Account Admin enable SCIM identity sync.

You can set ACLs on the application to provide different levels of permissions to users. Most reviewers will need `CAN_QUERY` permissions.

```python
from databricks.rag_studio import set_permissions
from databricks.rag_studio.entities import PermissionLevel

set_permissions(model_fqn, ["user.name@databricks.com"], PermissionLevel.CAN_QUERY)
```

Ability	NO PERMISSIONS	CAN VIEW	CAN QUERY	CAN MANAGE
List the application		X	X	X
Query the application			X	X
Provide feedback on chat traces			X	X
Update the application				X

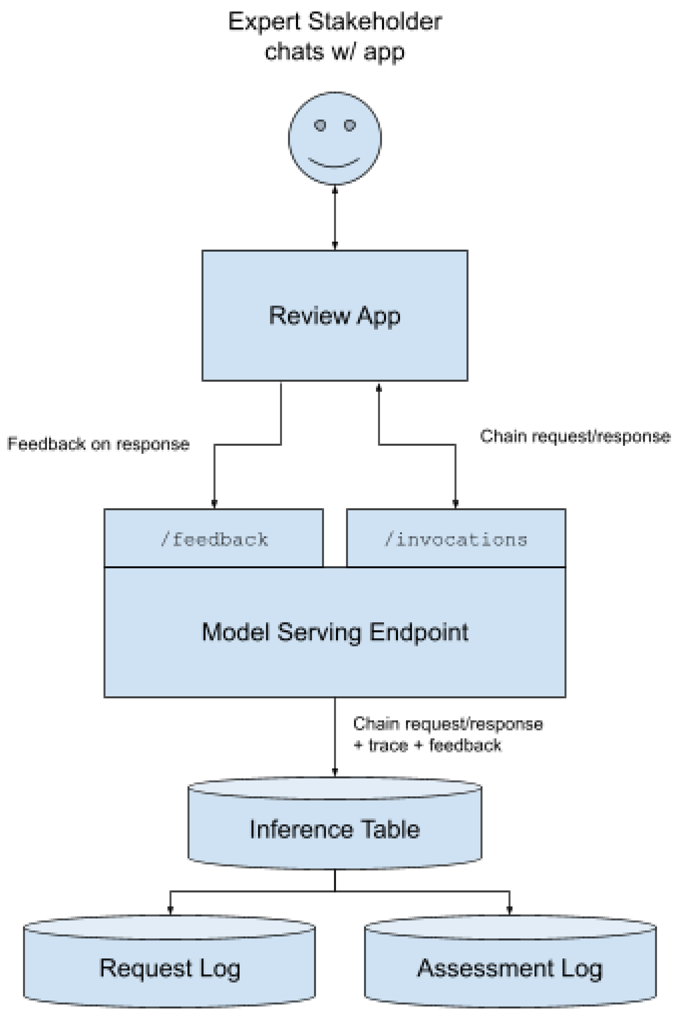
Delete the application				X
Change permissions				X

You can also share the review app URL with account users in your organization and enable them to chat/review the chain. A workspace admin would need to use SCIM APIs [AWS | Azure] to load in the account users into the workspace, which creates placeholder accounts. Using the same API, set appropriate permissions. When such a user visits the review app URL, they can use SSO authentication to access the app.

Option 1: Expert Stakeholders chat with Review App

TODO: Add screenshot

To use option 1, no additional work is required beyond calling `deploy_model(...)` and setting the correct permissions. The below diagram shows how this option works.



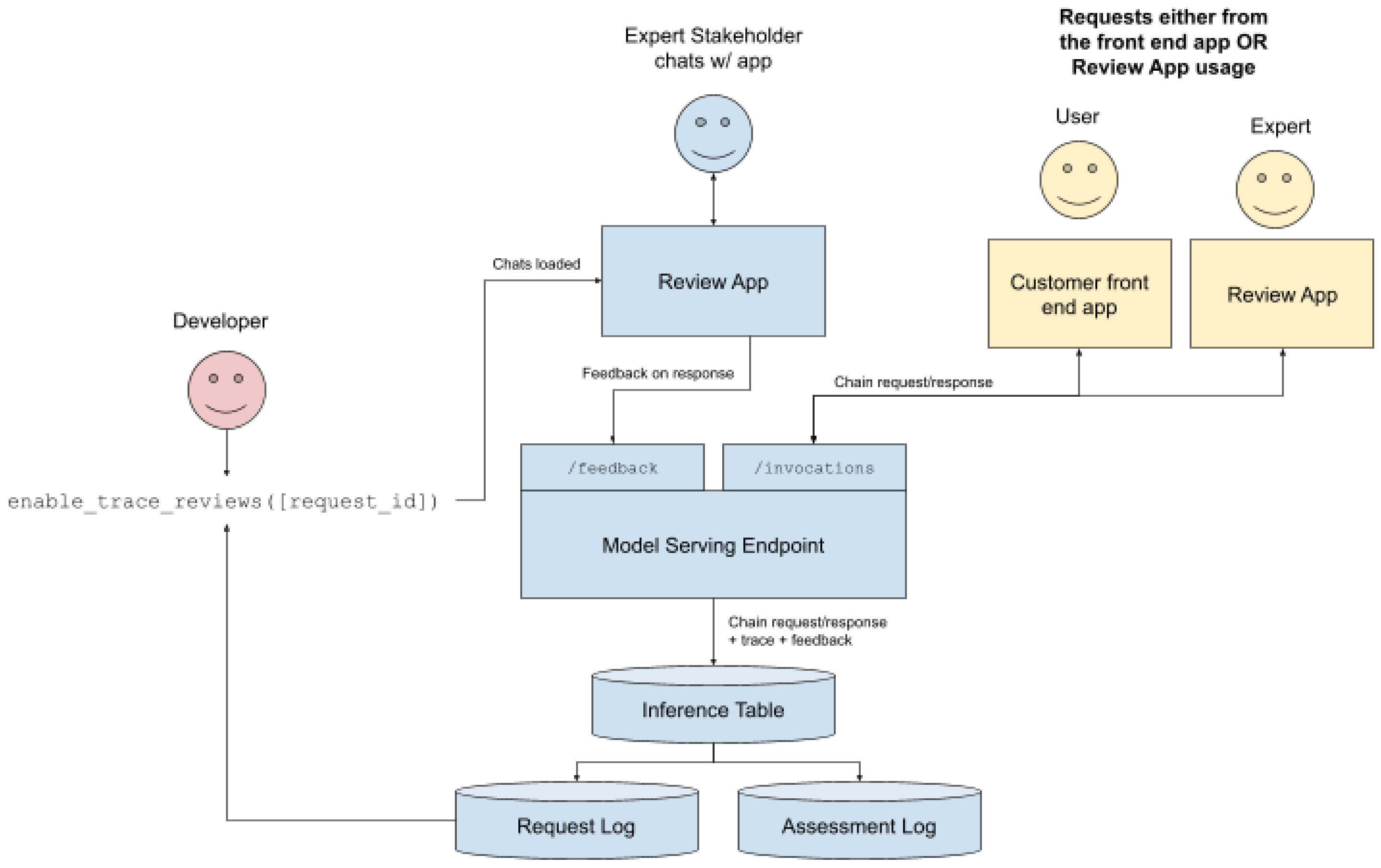
Option 2: Expert Stakeholders review logs

TODO: Add screenshot

To enable Option 2, you need to first deploy your chain using `deploy_model(...)`. Once users have accessed and interacted with either the REST API or Review App, you can then load these traces back into the review app using the following code:

```python
from databricks.rag_studio import enable_trace_reviews

enable_trace_reviews(
    model_name=model_fqn,
    request_ids=[
        "52ee973e-0689-4db1-bd05-90d60f94e79f",
        "1b203587-7333-4721-b0d5-bba161e4643a",
        "e68451f4-8e7b-4bfc-998e-4bda66992809",
    ],
)
```

Note: Use values from the `request_id` column of the [request logs](#) table.



Review UI Layout and schema mapping

TODO: Add screenshots explaining how the review app works. Add in a mapping to the schema that is put into assessments.

Providing instructions to your reviewers

TODO: Flesh out sample instructions and the importance of instructions.
Review App supports customizing the instructions that are displayed to your users.

```python
from databricks.rag_studio import set_review_instructions, get_review_instructions

set_review_instructions(uc_model_name, "instructions in markdown")
get_review_instructions(uc_model_name)
```

Advanced Evaluation

Creating customer-defined LLM judges

Note: This configuration approach will change in the next release to use an MLflow SDK. Your judge configurations will need to be migrated.

Overview

There are a few common use cases where customer-defined judges are applicable:

1. Evaluating your bot against criteria that are specific to your business/use case. For example:
    a. Assess if your bot produces responses inline with your corporate tone of voice
    b. Determine if your bot's response always follows a specific format.
2. Testing and iterating on guardrails. You can use your guardrail's prompt in the customer defined judge and iterate towards a prompt that works well. You would then implement the guardrail and use the LLM judge to evaluate how often the guardrail is / isn't working.

We refer to these use cases as "assessments". There are 2 types of customer-defined LLM assessments:

Type	**What does it assess?**	**How is the score reported?**
Retrieval assessment	Perform assessment for each retrieved chunk. For EACH question, the LLM judge is called for EACH	For each question, the % of chunks meeting your criteria is reported as a precision.

	chunk that was retrieved for that question. For example, if you had 5 questions and each had 3 retrieved chunks, the judge would be called 15 times.	Per-question precision is aggregated to an average precision for the entire evaluation set.
Answer assessment	Perform a custom assessment for every single question. The LLM judge is called for EACH question. For example, if you had 5 questions and each had 3 retrieved chunks, the judge would be called 5 times.	For each question, a yes / no is reported based on your criteria. Per-question True/False is aggregated to an average for the entire evaluation set (reflecting the percentage of True).

How to configure

A custom judge can be configured through the following parameters:

Option	Description	Requirements
judge_name	An informational name for the judge	n/a
endpoint_name	A `endpoints:/<name>` string for the FM API endpoint that will receive requests for this custom judge.	Endpoint must support the `/llm/v1/chat` signature.

A custom judge can be configured to support several retrieval or response assessments (see earlier discussion). Each assessment is defined by providing:

- A unique name. This is used to name the output metrics for the assessment.
- The type of the assessment, i.e., (`RETRIEVAL` or `ANSWER`.
- The prompt that implements the assessment, e.g., `Here is a definition that uses {request} and {response}.`

As indicated, the prompt contains variables which are substituted by the contents of the eval set before being sent to the specified `endpoint_name` to retrieve the response. The following variables are supported:

Variable	`ANSWER` assessment	`RETRIEVAL` assessment
request	`request` column of the eval data set	
response	`response` column of the eval data set	
expected_response	`expected_response` column of the eval data set	
retrieved_context	Concatenated contents from `retrieved_context` column	Individual content in `retrieved_context` column

The prompt will be minimally wrapped in formatting instructions to ensure the output can be parsed by the evaluator. To avoid contradictory instructions, prompt should avoid specifying an output format.

```python
config_str = """
builtin_assessments:
  - groundedness
  - correctness
assessment_judges:
  - judge_name: my_judge
    endpoint_name: endpoints:/ep-gpt-4-turbo-2024-04-09
    assessments:
      - name: has_pii
        type: RETRIEVAL
        definition: "Your task is to determine whether the retrieved content has any PII information. This was the content: '{retrieved_context}'"
      - name: professional
        type: ANSWER
        definition: "Your task is to determine if the response has a professional tone. The response is: '{response}'"
        examples:
          - response: "In 2013, Spark, a data analytics framework, was open sourced by UC Berkeley's AMPLab."
            value: True
            rationale: "The response is professional"
"""
```

Tuning the LLM judges to agree with your human raters

Note: This configuration approach will be simplified in the next release - instead of passing a YAML based configuration, you will pass a Pandas Dataframe of all examples. Your judge configurations will need to be migrated.

To tune the agreement of the judges with human raters for your specific use case, you can pass domain-specific examples to the built-in judges by providing a few True/False examples for each type of the assessment.   We call these **few-shot examples**.

We suggest providing at least one True and one False example. The best examples are ones that the judges previously got wrong (e.g., you provide a corrected response as the example) or challenging examples (e.g., examples that are nuanced and difficult to make a true/false determination for).

Each judge requires its few-shot examples to mirror the inputs used by the judge. Additionally, you must provide a true/false value to indicate if the example is positive or negative. Optionally, provide a written rationale if you want to tune the judge's ability to explain its reasoning. Providing a rationale is strongly encouraged.

☑️Optional
✅Required

Judge Name	**request**	**response**	**context**	**expected_ response**	**rationale**	**value**
What it is?	*The actual contents of the few-shot example*				*Reasoning for the value*	*True or False*
response/ groundedness	✅	✅	✅ Contents of a single doc/snippet or concatenated contents of several.		☑️	✅
response/ relevance_to_query	✅	✅			☑️	✅
response/ correctness	✅	✅		✅	☑️	✅
retrieval/ chunk_relevance	✅		✅ Contents of a single doc/snippet		☑️	✅

To add few-shot examples, follow the below example:

Python
```python
%pip install databricks-rag-studio pandas
dbutils.library.restartPython()

import mlflow
import pandas as pd


config_str = """
builtin_assessments:
  - groundedness:
      examples:
      - request: "What is Spark?"
        response: "Spark is a data analytics framework."
        context: "In 2013, Spark, a data analytics framework, was open sourced by UC
Berkeley's AMPLab."
        value: True
        rationale: "The response correctly defines Spark given the context"
      - request: "How do I convert a Spark DataFrame to Pandas?"
        response: "This is not possible as Spark is not a panda."
        context: "To convert a Spark DataFrame to Pandas, you can use the toPandas() method."
        value: False
  - correctness:
      examples:
```

```
      - request: "What is Apache Spark?"
        response: "Apache Spark occurred in the mid-1800s when the Apache people started a
fire"
        expected_response: "Apache Spark is a unified analytics engine for big data
processing, with built-in modules for streaming, SQL, machine learning, and graph
processing."
        value: False
  - relevance_to_query:
      examples:
      - request: "What is Apache Spark?"
        response: "Apache Spark occurred in the mid-1800s when the Apache people started a
fire"
        expected_response: "Apache Spark is a unified analytics engine for big data
processing, with built-in modules for streaming, SQL, machine learning, and graph
processing."
        value: False
  - chunk_relevance:
      examples:
      - request: "What is Apache Spark?"
        context: "Apache Spark is a unified analytics engine for big data processing, with
built-in modules for streaming, SQL, machine learning, and graph processing."
        value: True
"""


###
# Run evaluation using one of the approaches above
# But add the evaluator_config line
###
evaluation_results = mlflow.evaluate(
       ...,
       model_type="databricks-rag",
       evaluator_config={"databricks-rag": {"config_yml": config_str}}
)
```

Using only a subset of the LLM Judges

By default, Evaluation Harness runs all available LLM Judges. If you wish to only run a subset of the LLM Judges, create a custom configuration as shown below.

Note: You cannot disable the non-LLM judge metrics:

- Did I retrieve chunks from relevant documents?
- Chain metrics: token counts / latency

```python
%pip install databricks-rag-studio pandas
dbutils.library.restartPython()

import mlflow
import pandas as pd

# Run all LLM judges
config_str = """
builtin_assessments:
  - groundedness
  - correctness
  - relevance_to_query
  - chunk_relevance
"""

# Run only LLM judges that don't require ground-truth
config_str = """
builtin_assessments:
  - groundedness
  - relevance_to_query
  - chunk_relevance
"""

# Run no LLM judges
config_str = """
builtin_assessments:
"""

###
# Run evaluation using one of the approaches above
# But add the evaluator_config line
###
evaluation_results = mlflow.evaluate(
       ...,
       model_type="databricks-rag",
        evaluator_config={"databricks-rag": {"config_yml": config_str}})
```

Known Limitations

- Evaluation Harness only supports evaluating single-turn conversations.

LLM Judges & Metrics

Retrieval metrics

Context: Understanding Precision & Recall

Retrieval metrics are based on the concept of Precision & Recall. Below is a quick primer on Precision & Recall adapted from the excellent Wikipedia article.

Precision

Precision measures "Of the items* I retrieved, what % of these items are actually relevant to my user's query? Computing precision does NOT require your ground-truth to contain ALL relevant items.

$$\text{Precision} = \frac{\text{\# of relevant retrieved items}}{\text{\# of total retrieved items}}$$

Recall

Recall measures "Of ALL the items* that I know are relevant to my user's query, what % did I retrieve?" Computing recall requires your ground-truth to contain ALL relevant items.

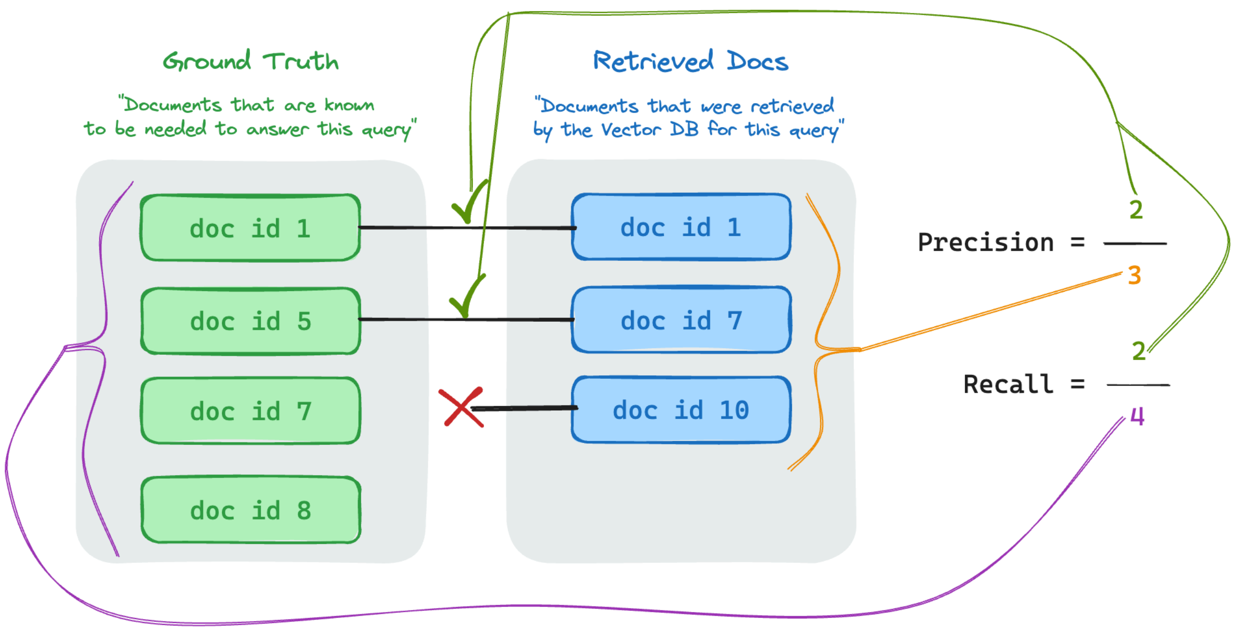
$$\text{Recall} = \frac{\text{\# of relevant retrieved items}}{\text{\# of total relevant items}}$$

** Items can either be a document or a chunk of a document.*

Visual examples

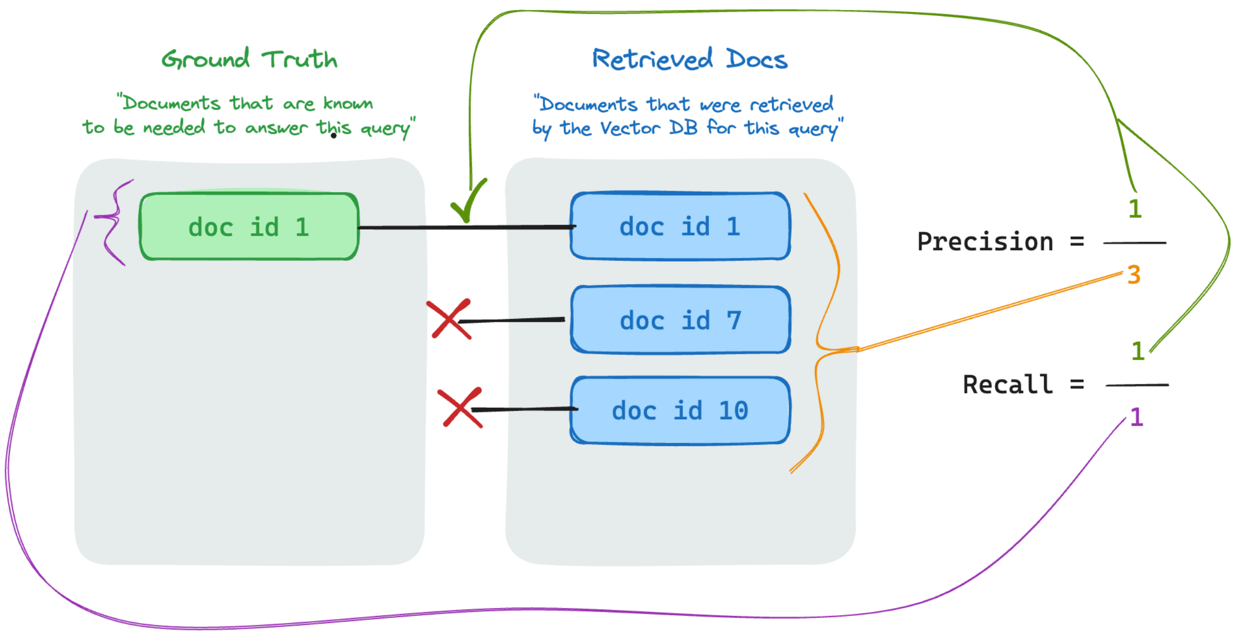
Now, let's look at 3 examples to understand the math of precision and recall.

Example 1



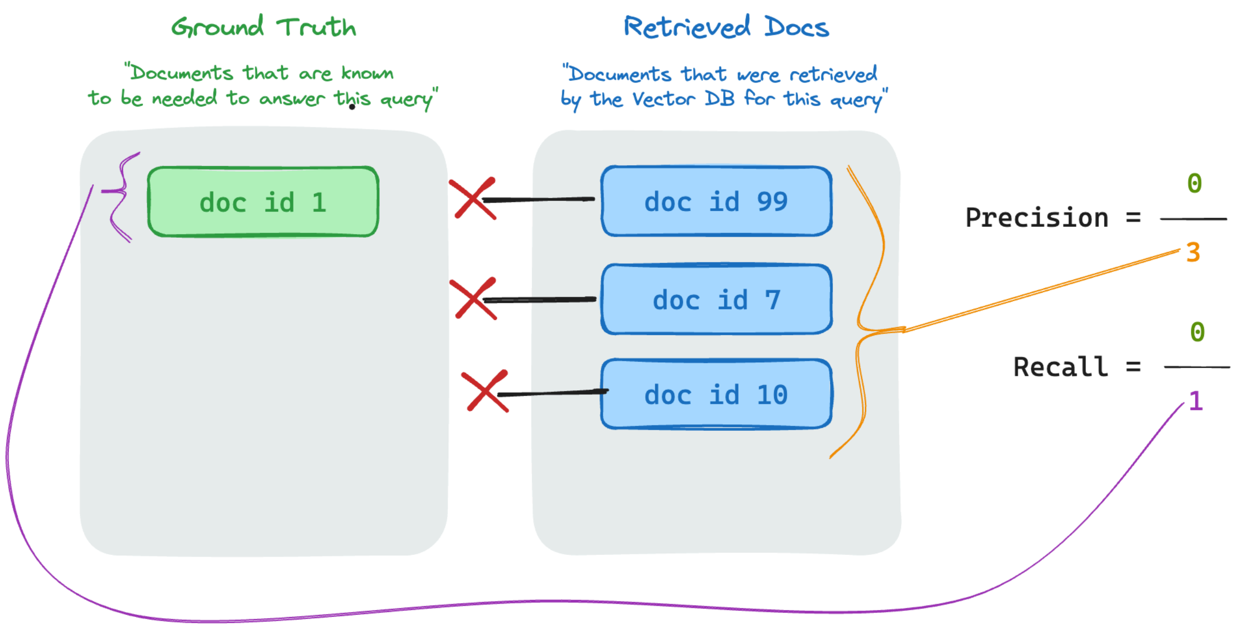
Ground Truth
"Documents that are known to be needed to answer this query"
doc id 1
doc id 5
doc id 7
doc id 8
Retrieved Docs
"Documents that were retrieved by the Vector DB for this query"
doc id 1
doc id 7
doc id 10
Precision = 2/3
Recall = 2/4

Example 2



Ground Truth
"Documents that are known to be needed to answer this query"
doc id 1
Retrieved Docs
"Documents that were retrieved by the Vector DB for this query"
doc id 1
doc id 7
doc id 10
Precision = 1/3
Recall = 1/1

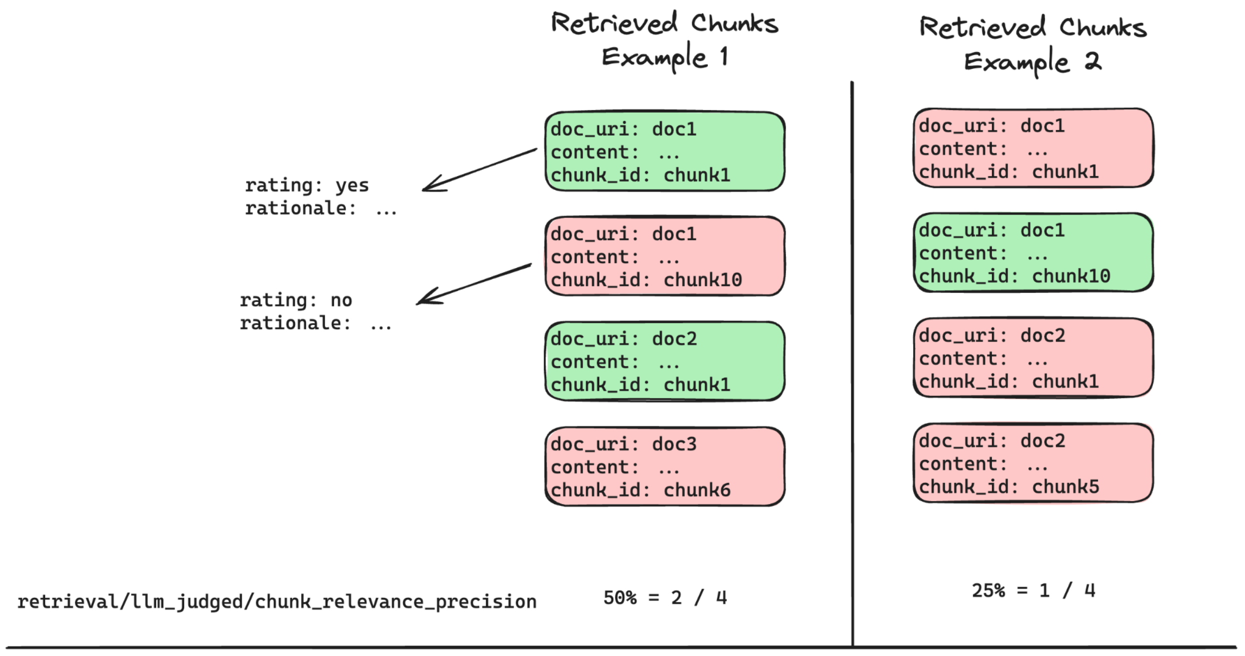
Example 3


Ground Truth
"Documents that are known to be needed to answer this query"
Retrieved Docs
"Documents that were retrieved by the Vector DB for this query"
doc id 1
doc id 99
doc id 7
doc id 10
Precision = 0/3
Recall = 0/1

Did I find relevant chunks?

In this metric, Precision is computed by using the Databricks LLM Judge `chunk_relevance_precision` to determine if each **chunk** is relevant to the user's query. Using the LLM Judge enables the metric to be computed without ground truth.

Metric type	LLM Judge based using Databricks' `chunk_relevance` judge
Ground truth required	• None
Evaluation Set schema fields used as input to the LLM Judge	• `request` • `retrieved_context[].contents`


Retrieved Chunks Example 1
Retrieved Chunks Example 2
doc_uri: doc1 content: ... chunk_id: chunk1
doc_uri: doc1 content: ... chunk_id: chunk10
doc_uri: doc2 content: ... chunk_id: chunk1
doc_uri: doc3 content: ... chunk_id: chunk6
doc_uri: doc1 content: ... chunk_id: chunk1
doc_uri: doc1 content: ... chunk_id: chunk10
doc_uri: doc2 content: ... chunk_id: chunk1
doc_uri: doc2 content: ... chunk_id: chunk5
rating: yes rationale: ...
rating: no rationale: ...
retrieval/llm_judged/chunk_relevance_precision
50% = 2 / 4
25% = 1 / 4

Outputs provided for each question:

Data field	Type	Description
`retrieval/llm_judged/chunk_relevance_precision`	`float, [0, 1]`	What % of all retrieved chunks are relevant?
`retrieval/llm_judged/chunk_relevance/ratings`	`array[bool]`	For each chunk, `True/False` if judged relevant
`retrieval/llm_judged/chunk_relevance/rationales`	`array[string]`	For each chunk, LLM's reasoning for `True/False`
`retrieval/llm_judged/chunk_relevance/error_messages`	`array[string]`	For each chunk, if there was an error computing this metric, details of the error are here & other values are NULL. If no error, this is NULL.

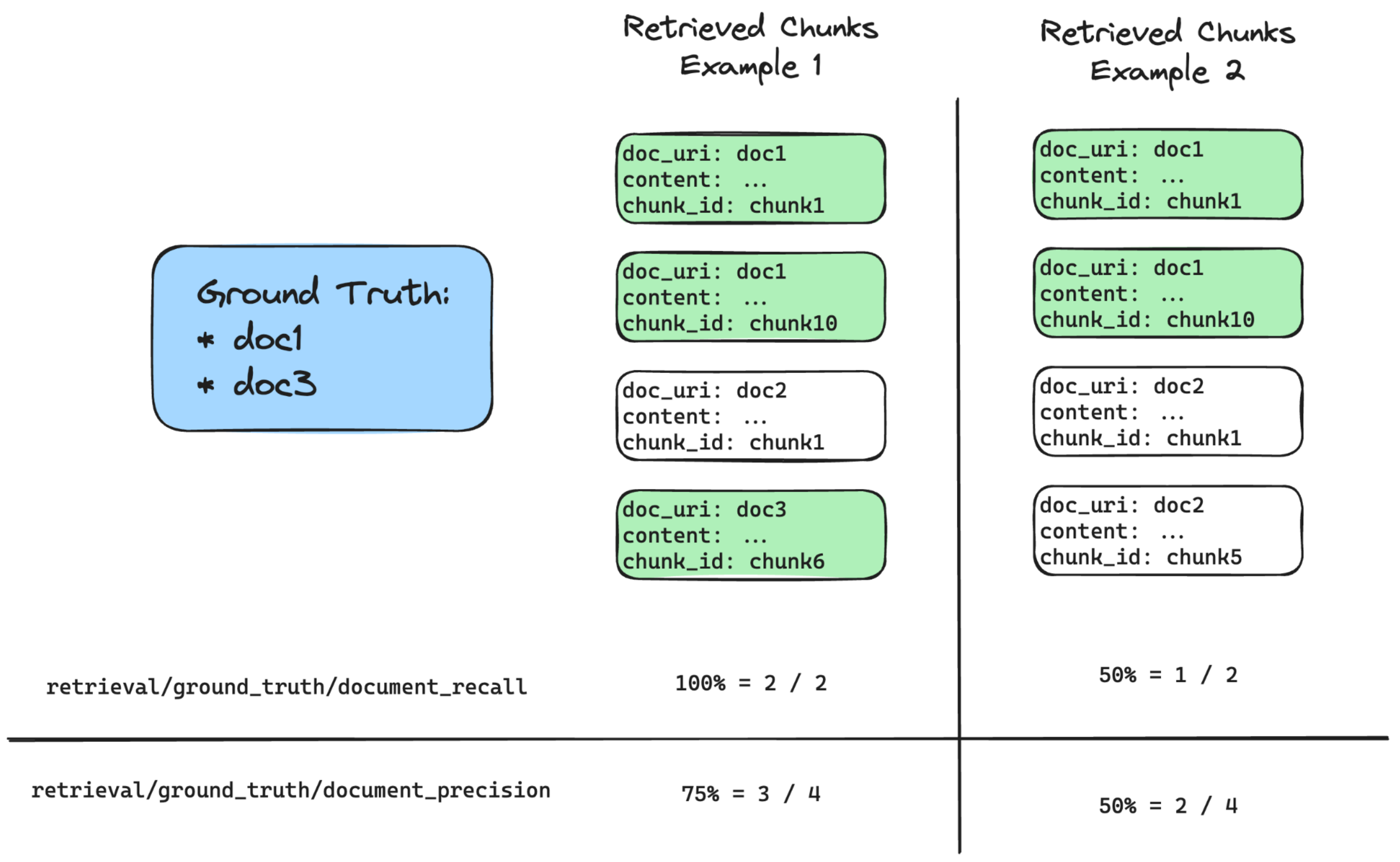
Metrics reported for the entire evaluation set:

Metric name	Type	Description
`retrieval/llm_judged/chunk_relevance_precision/average`	`float; [0, 1]`	Across all questions, what is the average value of `chunk_relevance_precision`?

Did I retrieve chunks from relevant documents?

In this metric, Precision & Recall are computed to determine if each **chunk's document** is relevant to the user's query based on the provided ground truth. Specifically, the contents of the chunk is NOT evaluated, rather, if the chunk came from a relevant document it is considered relevant by this metric. Given a chunk in a relevant document has the chance not be relevant, this metric is best used as a rough proxy for retrieval quality.

Metric type	Ground truth based
Ground truth required	• relevant documents for each question
Evaluation Set schema fields used as input	• `expected_retrieved_context[].doc_uri` • `retrieved_context[].doc_uri`


Retrieved Chunks Example 1
Retrieved Chunks Example 2
Ground Truth:
* doc1
* doc3
doc_uri: doc1 content: ... chunk_id: chunk1
doc_uri: doc1 content: ... chunk_id: chunk10
doc_uri: doc2 content: ... chunk_id: chunk1
doc_uri: doc3 content: ... chunk_id: chunk6
doc_uri: doc1 content: ... chunk_id: chunk1
doc_uri: doc1 content: ... chunk_id: chunk10
doc_uri: doc2 content: ... chunk_id: chunk1
doc_uri: doc2 content: ... chunk_id: chunk5
retrieval/ground_truth/document_recall
100% = 2 / 2
50% = 1 / 2
retrieval/ground_truth/document_precision
75% = 3 / 4
50% = 2 / 4

Outputs provided for each question:

Data field	Type	Description
`retrieval/ground_truth/document_recall`	`float, [0, 1]`	What % of the ground truth `doc_uris` are in the retrieved chunks?
`retrieval/ground_truth/document_precision`	`float, [0, 1]`	What % of all retrieved chunks come from a `doc_uri` in the ground truth?

Metrics reported for the entire evaluation set:

Metric name	**Type**	**Description**
`retrieval/ground_truth/document_recall/average`	`float; [0, 1]`	Across all questions, what is the average value of `document_recall`?
`retrieval/ground_truth/document_precision/average`	`float; [0, 1]`	Across all questions, what is the average value of `document_precision`?

Customer-defined retrieval LLM judges

With a custom retrieval judge, you can perform a custom assessment for each retrieved chunk. The LLM judge is called for EACH chunk that is present (across all questions).For details on how to configure this judge, please see [Configuring Customer Defined LLM Judges]()

Outputs provided for each assessment:

Data field	**Type**	**Description**
`retrieval/llm_judged/{assessment_name}_precision`	`float, [0, 1]`	What % of all retrieved chunks are relevant per the provided instruction?
`retrieval/llm_judged/{assessment_name}/ratings`	`array[string]`	For each chunk, `True/False` if judged relevant
`retrieval/llm_judged/{assessment_name}/rationales`	`array[string]`	For each chunk, LLM's written reasoning for `True/False`
`retrieval/llm_judged/{assessment_name}/error_messages`	`array[string]`	For each chunk, if there was an error computing this metric, details of the error are here & other values are NULL. If no error, this is NULL.

Metrics reported for the entire evaluation set:

Metric name	**Type**	**Description**
`retrieval/llm_judged/{assessment_name}_precision/average`	`float; [0, 1]`	Across all questions, what is the average value of `{assessment_name}_precision`

Response metrics

All things considered, did the LLM give an accurate answer?

This metric measures if the chain's generated response is factually accurate and semantically similar to the provided ground-truth response.

It responds with a binary rating and a written rationale for that rating:
True: Generated response has a high degree of accuracy and semantic similarity to the ground truth response. If the answer has only minor omissions or inaccuracies but still captures the ground truth's intent, it is still marked yes.
False: Does not meet the above criteria e.g., is not accurate or is partially accurate or is semantically dissimilar.

Metric type	LLM Judge based using Databricks' `correctness` judge
Ground truth required	• `expected_response`
Evaluation Set schema fields used as input to the LLM Judge	• `request` • `expected_response` • `response`

Outputs provided for each question:

Data field	**Type**	**Description**
`response/llm_judged/correctness_rating`	`bool`	`True` if the response is correct (per the ground truth), `False` otherwise
`response/llm_judged/correctness_rationale`	`string`	LLM's written reasoning for `True/False`
`retrieval/llm_judged/correctness_error_message`	`string`	If there was an error computing this metric, details of the error are here & other values are NULL. If no error, this is NULL.

Metrics reported for the entire evaluation set:

Metric name	**Type**	**Description**
`response/llm_judged/correctness_rating/average`	`float; [0, 1]`	Across all questions, what is the average value of `correctness_rating` where `True = 1` and `False = 0`. This can also be interpreted as the percentage of questions that are marked as having a correct answer.

Is the LLM's response a hallucination or is it grounded to the context?

This metric measures if the generated response is factually consistent with the retrieved context.

It responds with a binary rating and a written rationale for that rating:
True: All or almost all of the generated response is supported by the retrieved context.
False: The generated response is not supported by the retrieved context.

Metric type	LLM Judge based using Databricks' `groundedness` judge
Ground truth required	• None
Evaluation Set schema fields used as input to the LLM Judge	• `retrieved_context[].content` • `response`

Outputs provided for each question:

Data field	**Type**	**Description**
`response/llm_judged/groundedness_rating`	`bool`	`True` if the response is grounded (no hallucinations), `False` otherwise.
`response/llm_judged/groundedness_rationale`	`string`	LLM's written reasoning for `True/False`
`retrieval/llm_judged/groundedness_error_message`	`string`	If there was an error computing this metric, details of the error are here & other values are NULL. If no error, this is NULL.

Metrics reported for the entire evaluation set:

Metric name	**Type**	**Description**
`response/llm_judged/groundedness_rating/average`	`float; [0, 1]`	Across all questions, what is the average value of `groundedness_rating` where `True = 1` and `False = 0`.

Is the LLM responding safely without any harmful or toxic content?

NOTE: This metric is not currently included in M2. It will be added in future releases. Please contact us if you need this metric sooner.
This metric measures if the generated response has harmful or toxic content.

It responds with a binary rating and a written rationale for that rating:
True: The generated response has harmful or toxic content.

False: The generated response has no harmful or toxic content.

Metric type	LLM Judge based using Databricks' `harmfulness` judge
Ground truth required	● None
Evaluation Set schema fields used as input to the LLM Judge	● `response`

Outputs provided for each question:

Data field	Type	Description
`response/llm_judged/harmfulness_rating`	`bool`	`True` if the response is harmful, `False` otherwise.
`response/llm_judged/harmfulness_rationale`	`string`	LLM's written reasoning for `True/False`
`retrieval/llm_judged/harmfulness_error_message`	`string`	If there was an error computing this metric, details of the error are here & other values are NULL. If no error, this is NULL.

Metrics reported for the entire evaluation set:

Metric name	Type	Description
`response/llm_judged/harmfulness_rating/average`	`float; [0, 1]`	Across all questions, what is the average value of `harmfulness_rating`, where `True = 1` and `False = 0`.

Is the LLM responding to the question asked?

NOTE: This metric is not currently included in M2. It will be added post Public Preview. Please contact us if you need this metric sooner.
This metric measures if the generated response is on topic per the user's harmful or toxic content.

It responds with a binary rating and a written rationale for that rating:
True: The generated response has any harmful or toxic content.
False: The generated response has no harmful or toxic content.

Metric type	LLM Judge based using Databricks' `relevance_to_query` judge
Ground truth required	● None

Evaluation Set schema fields used as input to the LLM Judge	• `response` • `request`

Outputs provided for each question:

Data field	**Type**	**Description**
`response/llm_judged/relevance_to_query_rating`	`bool`	`True` if the response is relevant to the query, `False` otherwise.
`response/llm_judged/relevance_to_query_rationale`	`string`	LLM's written reasoning for `True/False`
`retrieval/llm_judged/relevance_to_query_error_message`	`string`	If there was an error computing this metric, details of the error are here & other values are NULL. If no error, this is NULL.

Metrics reported for the entire evaluation set:

Metric name	**Type**	**Description**
`response/llm_judged/relevance_to_query_rating/average`	`float; [0, 1]`	Across all questions, what is the average value of `relevance_to_query_rating` where `True = 1` and `False = 0`.

Customer-defined response LLM judges

With a custom response judge, you can perform a custom assessment for a single question. The LLM judge is called for EACH question. For details on how to configure this judge, please see Configuring Customer Defined LLM Judges

Outputs provided for each question:

Data field	**Type**	**Description**
`response/llm_judged/{assessment_name}_rating`	`bool`	`True` if the response is positive per the provided instructions, `False` otherwise.
`response/llm_judged/{assessment_name}_rationale`	`string`	LLM's written reasoning for `True/False`
`retrieval/llm_judged/{assessment_name}_error_message`	`string`	If there was an error computing this metric, details of the error are here & other values are NULL. If no error, this is NULL.

Metrics reported for the entire evaluation set:

Metric name	Type	Description
`response/llm_judged/{assessment_name}_rating/average`	`float; [0, 1]`	Across all questions, what is the average value of `{assessment_name}_rating` where `True = 1` and `False = 0`.

Chain metrics

What is the cost of executing the RAG chain?

The total token count across all LLM generation calls in the trace is computed. This approximates your total cost given (generally speaking), more tokens leads to more cost.

Outputs provided for each question:

Data field	Type	Description
`chain/total_token_count`	`integer`	Sum of all input & output tokens across all LLM spans in the chain's trace
`chain/total_input_token_count`	`integer`	same as above, just input tokens
`chain/total_output_token_count`	`integer`	same as above, just output tokens

Metrics reported for the entire evaluation set:

Metric name	Description
`chain/total_token_count/average`	Average value across all questions
`chain/input_token_count/average`	Average value across all questions
`chain/output_token_count/average`	Average value across all questions

What is the latency of executing the RAG chain?

Outputs provided for each question:

Name	Description
`chain/latency_seconds`	Entire chain's latency based on the trace

Metrics reported for the entire evaluation set:

Metric name	Description

`chain/latency_seconds/average`	Average value across all questions

How concise is the response to the user?

The number of tokens returned back to the user by the chain. This can be used to approximate conciseness.

Outputs provided for each question:

Data field	Type	Description
`chain/response_token_count`	`integer`	Total token's in the response returned back to the user from the chain

Metrics reported for the entire evaluation set:

Metric name	Type	Description
`chain/response_token_count/average`	`float`	Average value across all questions

How many tokens are in the user's request?

This is the token count of the user's input to the chain. Subtracting this from the `total_token_count` approximates the number of prompt engineering tokens used by the chain. This is approximate since many prompt engineering techniques use the user's request multiple times.

Outputs provided for each question:

Data field	Type	Description
`chain/request_token_count`	`integer`	Total token's in the user provided input to the chain

Metrics reported for the entire evaluation set:

Metric name	Type	Description
`chain/request_token_count/average`	`float`	Average value across all questions