

Onboarding Instructions for RAG Studio

Imprint: Mar 28, 2024

Version: v1.0

This document describes the steps needed to deploy your first custom RAG application using RAG Studio.

However, we suggest getting started by importing the bundle of sample code provided to you into your Workspace and following the [RAG Studio Getting Started](#) notebook to run the getting started tutorials.

You can write your RAG application as a chain in a supported framework (See [current limitations](#)) which can easily be deployed and evaluated on Databricks using RAG Studio. Deploying the chain on our platform provides you an easy way to evaluate the quality of your application and iterate quickly to improve the quality of the application.

Setup

Recommended: Databricks Runtime 15.0, Works on Databricks Runtime 14.3 LTS.

Note: We suggest using the Databricks Runtime NOT the Databricks Machine Learning Runtime.

Python

```
%pip install --quiet "PUT_RAG_EVAL_SUITE_WHEEL_HERE"  
%pip install --quiet "PUT_RAG_STUDIO_WHEEL_HERE"  
%restart_python
```

Note: If using Databricks Runtime 14.3 LTS and below you will need to add the following cell after installing pip package.

Python

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

1. Write your RAG application

When using RAG Studio, you create your chain code using LangChain inside a Notebook. Then, you use a “Driver” notebook to log, evaluate, and deploy new versions of your chain.

We also support providing parameters (key:value pairs) in a yaml configuration file that can be accessed within your code as a JSON object. This allows you to experiment with different parameters (like different vector indexes, LLM endpoints, prompt).

You can see a sample chain code and driver notebooks in the included bundle of code:

- Hello world example: `1_hello_world_chain` and `1_hello_world_chain_driver_notebook`
- Hello world example w/ parameters: `2_hello_world_parameterized_chain` and `2_hello_world_parameterized_chain_driver_notebook`
- RAG Chain: `3_rag_chain` and `3_rag_chain_driver_notebook`

Current Limitations:

- We only support chains using the Langchain framework. Generic python functionality is coming soon.
- Custom python library dependencies and versions are not supported at this time.
- Custom credentials for external services are not supported at this time.
- Your chain should have a [retriever](#) defined, using [Databricks VectorSearch](#). This is what we currently use to determine the application is a valid RAG application and deploy the necessary credentials to your serving endpoint. This requirement is being removed very soon.
- Models other than Databricks hosted Foundational Model APIs are not supported - including Provisioned Throughput and External models (e.g., OpenAI). Removing this restriction is part of the roadmap.
- RAG Studio doesn't support Workspace Repos. You can use a Workspace folder or a Git Folder inside a Workspace Folder.
- Online evaluation does not currently work. We are adding this functionality in.
- The inference table from the deployed chain is in a raw format. To view the traces/human feedback in a more useful schema, you can use the included `6_export_inference_table_to_logs` Notebook - we are adding functionality in to automatically run this notebook.
- Trace logging functionality is limited - we are working to upgrade this functionality and add a visualization UI. If you want early access, please let us know.
- [**ADD any other constraints**]

Python

```
code_path = "/Workspace/Users/akhil.gupta/chain.py"
config_path = "/Workspace/Users/akhil.gupta/config.yml"
```

2. Log the chain

2a) Log to an MLflow Run and verify it.

Python

```
import os
import mlflow
from databricks.rag_studio import log_model

# config_path is optional
logged_chain_info = log_model(code_path=code_path, config_path=config_path)

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
```

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

Python

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

```
model=mlflow.langchain.load_model(model_uri)
model.invoke(question)
```

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

2b) Register to Unity Catalog

Pick the best model to register into Unity Catalog

```
Python
import mlflow

mlflow.set_registry_uri("databricks-uc")

# TO FIX: Specify the catalog, schema and the model name
catalog_name = "test_catalog"
schema_name = "schema"
model_name = "chain_name"

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

3. Deploy the RAG chain

Deploy the chain to:

- 1) Review App so you & your stakeholders can chat with the chain & given feedback via a web UI.
- 2) Chain REST API endpoint to call the chain from your front end
- 3) Feedback REST API endpoint to pass feedback back from your front end.

Note: It can take up to 15 minutes to deploy - we are working to reduce this time to seconds.

Python

```
from databricks.rag_studio import deploy_model

deployment = deploy_model(model_fqn, model_version.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
```

Python

```
# Use the below code block to get the Serving Endpoint to view Deployment
status

from mlflow.utils import databricks_utils as du

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)
```

4. (Optional) Set permissions to the application

You can set ACLs on the application to provide different level of permissions to users

```

Python
from databricks.rag_studio import set_permissions
from databricks.rag_studio.entities import PermissionLevel

set_permissions(model_fqn, ["aravind.segu@databricks.com"],
PermissionLevel.CAN_VIEW)

```

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

5. Listing deployed applications

```

Python
from databricks.rag_studio import list_deployments, get_deployment

# 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

```


Evaluating your RAG application

You can evaluate a RAG application using the Databricks RAG Evaluation Framework. The evaluation framework requires a delta table with a set of reference questions (and optionally ground-truth for each question). We will refer to this as the **eval_set**.

For complete documentation on the RAG evaluation, please see the full Evaluation Suite documentation PDF.

Eval set

The schema of the eval table is as follows:

Column name	Type	Required?	Comment
<code>request_id</code>	STRING		Id of the request (question)
<code>request</code>	STRING		A request (question) to the RAG app, e.g., "What is Spark?"
<code>Expected_response [OPTIONAL]</code>	STRING		(Optional) The expected answer to this question
<code>Expected_retrieval_context [OPTIONAL]</code>	ARRAY< STRUCT< doc_uri: STRING, content: STRING > >		(Optional) The expected retrieval context. The entries are ordered in descending rank. Each entry can record the URI of the retrieved doc and optionally the (sub)content that was retrieved.

Note that the table can contain **partial ground truth** (`expected_response` and `expected_retrieval_context`) for a request or **no ground-truth at all**.

Set up

```
Python
%pip install --quiet "PUT_RAG_EVAL_SUITE_WHEEL_HERE"

dbutils.library.restartPython()
```

Creating an Eval set

```
Python
#####
# If you have a known set of queries, you can build the evaluation dataset
manually
# Alternatively, you can create the evaluation dataset using Spark/SQL - it is
simply an Delta Table with the above schema
#####

eval_dataset = [
    {
        "request_id": "sample_request_1",
        "request": "What is ARES?",
        # Expected retrieval context is optional, if not provided, RAG Studio will
        use LLM judge to assess each retrieved context
        "expected_retrieval_context": [
            {
                "chunk_id": "9517786ecadf3e0c75e3cd4ccefdced5",
                "doc_uri": "dbfs:/Volumes/rag/ericp_m1/matei_pdf/2311.09476.pdf",
                "content": "..."
            },
            {
                "chunk_id": "e8825fe982f7fd190ad828a307d7f280",
                "doc_uri": "dbfs:/Volumes/rag/ericp_m1/matei_pdf/2311.09476.pdf",
                "content": "..."
            }
        ]
    }
]
```



```

    },
    {
      "chunk_id": "e47b43c9c8f8ce11d78342c49ddbea07",
      "doc_uri": "dbfs:/Volumes/rag/ericp_m1/matei_pdf/2311.09476.pdf",
      "content": "..."
    }
  ],
  # Expected response is optional
  "expected_response": "ARES is an Automated RAG Evaluation System for
evaluating retrieval-augmented generation (RAG) systems along the dimensions of
context relevance, answer faithfulness, and answer relevance. It uses synthetic
training data to finetune lightweight LM judges to assess the quality of
individual RAG components and utilizes a small set of human-annotated
datapoints for prediction-powered inference (PPI) to mitigate potential
prediction errors.",
}
]

#####
# Turn the eval dataset into a Delta Table
#####
uc_catalog = "catalog"
uc_schema = "schema"
eval_table_name = "sample_eval_set"
eval_table_fqn = f"{uc_catalog}.{uc_schema}.{eval_table_name}"

df = spark.read.json(spark.sparkContext.parallelize(eval_dataset))
df.write.format("delta").option("mergeSchema",
"true").mode("overwrite").saveAsTable(
  eval_table_fqn
)
print(f"Loaded eval set to: {eval_table_fqn}")

```

Running the Eval

```

Python
from databricks import rag_eval

```

```

model_uri = f"models://{model_fqn}/{model_version.version}"

evaluation = rag_eval.evaluate(eval_set_table_name=eval_table_fqn,
                              model_uri=model_uri)

# Command output
Evaluation completed successfully. You can open the dashboard to visualize the results.
You can inspect the updated metrics in
`rag`.`eval_bugbash_prpr`.`docqa_eval_set_eval_metrics`, and the updated assessments in
`rag`.`eval_bugbash_prpr`.`docqa_eval_set_assessments`.

evaluation
EvaluationOutput(eval_metrics_table_name='`uc_path`.`to`.`eval_table_eval_metrics`',
                 assessments_table_name='`uc_path`.`to`.`eval_table_assessments`',
                 dashboard_id='my_dashboard_id',
                 dashboard_url='url to lakeview dashboard',
                 mlflow_run_id='my_mlflow_run_d',
                 mlflow_run_url='my_mlflow_run_url')

```

Looking at the Evaluation Output

You can past **evaluation.dashboard_url** to look at the results of the evaluation and measure the quality of your data based on **eval_set**