AWS Big Data Blog

Enrich your AWS Glue Data Catalog with generative AI metadata using Amazon Bedrock

by Manos Samatas and Anastasia Tzeveleka | on 15 NOV 2024 | in Amazon Bedrock, Amazon Machine Learning, Analytics, AWS Big Data, AWS Glue, Generative AI, Intermediate (200), Technical How-to | Permalink | Comments | Share

Metadata can play a very important role in using data assets to make data driven decisions. Generating metadata for your data assets is often a time-consuming and manual task. By harnessing the capabilities of generative AI, you can automate the generation of comprehensive metadata descriptions for your data assets based on their documentation, enhancing discoverability, understanding, and the overall data governance within your AWS Cloud environment. AI recommendations for descriptions are available today in Amazon DataZone. With AI-generated descriptions in Amazon DataZone, data consumers have these recommended descriptions to identify data tables and columns for analysis, which enhances data discoverability and cuts down on back-and-forth communications with data producers. In this blog, we will build on top of our native solutions to help you enrich your custom business metadata, by leveraging external data documentation and foundation models (FMs) on Amazon Bedrock. By leveraging extenal documentation we can improve the accuracy of generated metadata. For this blogpost, we will use the AWS Glue Data Catalog. You can extend the solution to include other data catalogs, such as Amazon DataZone.

AWS Glue is a serverless data integration service that makes it straightforward for analytics users to discover, prepare, move, and integrate data from multiple sources. Amazon Bedrock is a fully managed service that offers a choice of high-performing FMs from leading AI companies like AI21 Labs, Anthropic, Cohere, Meta, Mistral AI, Stability AI, and Amazon through a single API.

Solution overview

In this solution, we automatically generate metadata for table definitions in the Data Catalog by using large language models (LLMs) through Amazon Bedrock. First, we explore the option of in-context learning, where the LLM generates the requested metadata without documentation. Then we improve the metadata generation by adding the data documentation to the LLM prompt using Retrieval Augmented Generation (RAG).

AWS Glue Data Catalog

This post uses the Data Catalog, a centralized metadata repository for your data assets across various data sources. The Data Catalog provides a unified interface to store and query information about data formats, schemas, and sources. It acts as an index to the location, schema, and runtime metrics of your data sources.

The most common method to populate the Data Catalog is to use an AWS Glue crawler, which automatically discovers and catalogs data sources. When you run the crawler, it creates metadata tables that are added to a database you specify or the default database. Each table represents a single data store.

Generative AI models

LLMs are trained on vast volumes of data and use billions of parameters to generate outputs for common tasks like answering questions, translating languages, and completing sentences. To use an LLM for a specific task like metadata generation, you need an approach to guide the model to produce the outputs you expect.

This post shows you how to generate descriptive metadata for your data with two different approaches:

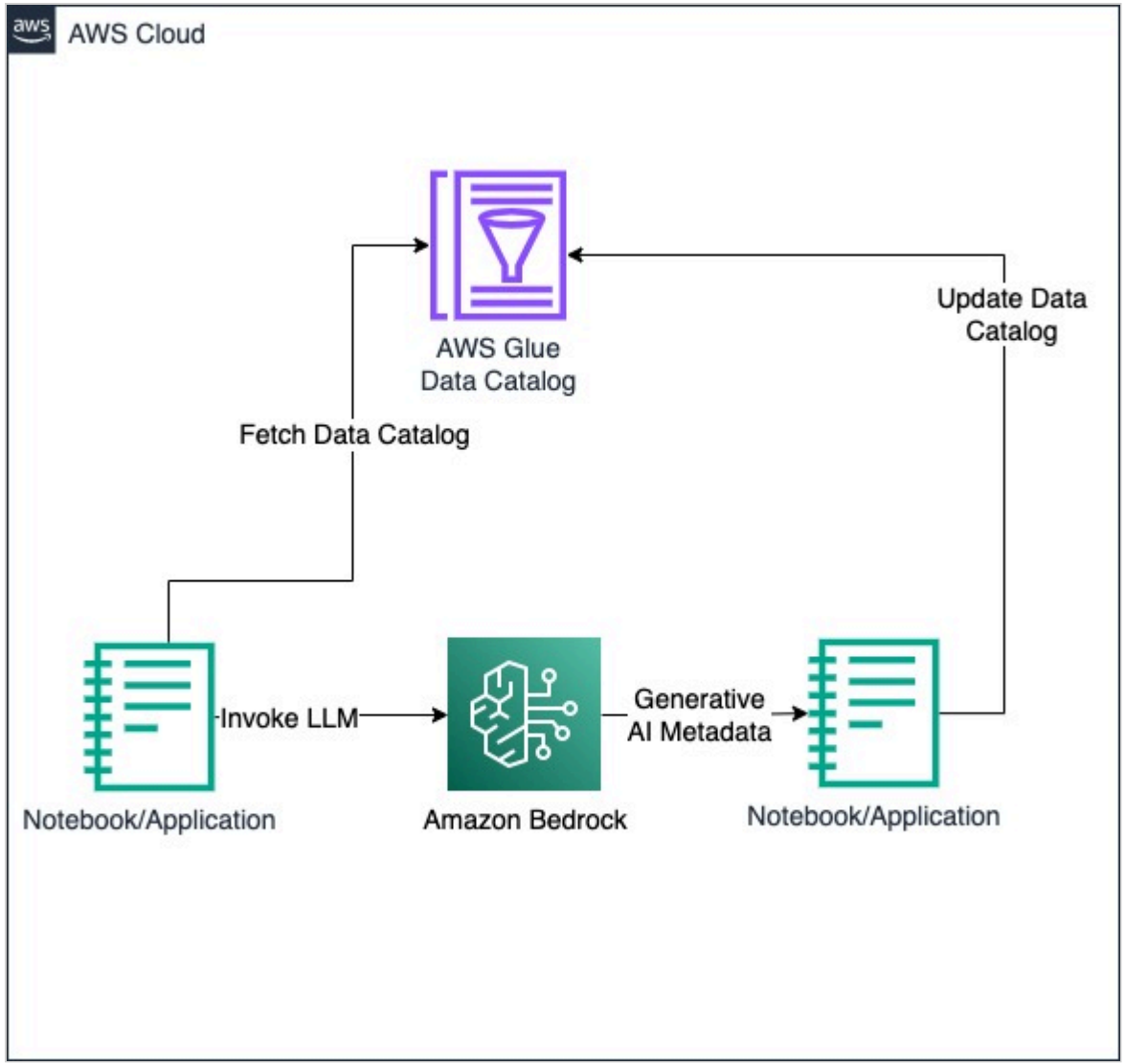
- In-context learning
- Retrieval Augmented Generation (RAG)

The solutions uses two generative AI models available in Amazon Bedrock: for text generation and Amazon Titan Embeddings V2 for text retrieval tasks.

The following sections describe the implementation details of each approach using the Python programming language. You can find the accompanying code in the GitHub repository. You can implement it step by step in Amazon SageMaker Studio and JupyterLab or your own environment. If you're new to SageMaker Studio, check out the Quick setup experience, which allows you to launch it with default settings in minutes. You can also use the code in an AWS Lambda function or your own application.

Approach 1: In-context learning

In this approach, you use an LLM to generate the metadata descriptions. You employ prompt engineering techniques to guide the LLM on the outputs you want it to generate. This approach is ideal for AWS Glue databases with a small number of tables. You can send the table information from the Data Catalog as context in your prompt without exceeding the context window (the number of input tokens that most Amazon Bedrock models accept). The following diagram illustrates this architecture.



Approach 2: RAG architecture

If you have hundreds of tables, adding all of the Data Catalog information as context to the prompt may lead to a prompt that exceeds the LLM's context window. In some cases, you may also have additional content such as business requirements documents or technical documentation you want the FM to reference before

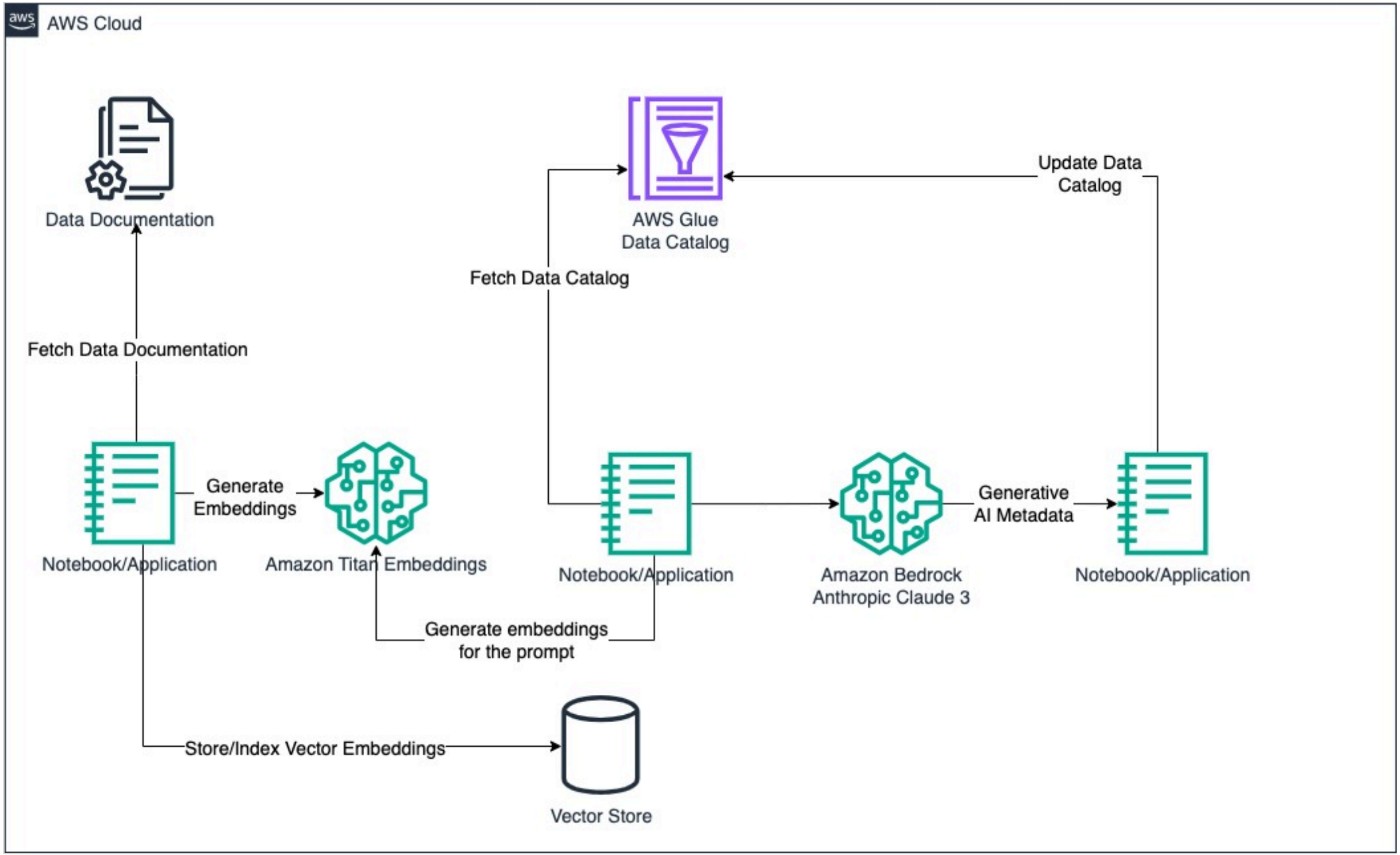
generating the output. Such documents can be several pages that typically exceed the maximum number of input tokens most LLMs will accept. As a result, they can't be included in the prompt as they are.

The solution is to use a RAG approach. With RAG, you can optimize the output of an LLM so it references an authoritative knowledge base outside of its training data sources before generating a response. RAG extends the already powerful capabilities of LLMs to specific domains or an organization's internal knowledge base, without the need to fine-tune the model. It is a cost-effective approach to improving LLM output, so it remains relevant, accurate, and useful in various contexts.

With RAG, the LLM can reference technical documents and other information about your data before generating the metadata. As a result, the generated descriptions are expected to be richer and more accurate.

The example in this post ingests data from a public Amazon Simple Storage Service (Amazon S3): `s3://awsglue-datasets/examples/us-legislators/all`. The dataset contains data in JSON format about US legislators and the seats that they have held in the U.S. House of Representatives and U.S. Senate. The data documentation was retrieved from and the Popolo specification http://www.popoloproject.com/.

The following architecture diagram illustrates the RAG approach.


AWS Cloud
Data Documentation
AWS Glue
Data Catalog
Update Data
Catalog
Fetch Data Catalog
Fetch Data Documentation
Generate
Embeddings
Notebook/Application
Amazon Titan Embeddings
Notebook/Application
Amazon Bedrock
Anthropic Claude 3
Generative
AI Metadata
Notebook/Application
Generate embeddings
for the prompt
Store/Index Vector Embeddings
Vector Store

The steps are as follows:

1. Ingest the information from the data documentation. The documentation can be in a variety of formats. For this post, the documentation is a website.
2. Chunk the contents of the HTML page of the data documentation. Generate and store vector embeddings for the data documentation.
3. Fetch information for the database tables from the Data Catalog.
4. Perform a similarity search in the vector store and retrieve the most relevant information from the vector store.
5. Build the prompt. Provide instructions on how to create metadata and add the retrieved information and the Data Catalog table information as context. Because this is a rather small database, containing six tables, all of the information about the database is included.
6. Send the prompt to the LLM, get the response, and update the Data Catalog.

Prerequisites

To follow the steps in this post and deploy the solution in your own AWS account, refer to the GitHub repository.

You need the following prerequisite resources:

- An AWS account.
- Python and boto3.
- An AWS Identity and Access Management (IAM) role for the AWS Glue crawler that includes the AWSGlueServiceRole policy or equivalent and an inline policy with access to the S3 bucket with the data used in this post. For this post, as part of the environment set up we create a new S3 bucket with the name `aws-gen-ai-glue-metadata-<random_sequence>`. The following is an example inline policy:

```
{
  "Version": "2012-10-17",
   "Statement": [
       {
         "Effect": "Allow",
         "Action": [
             "s3:GetObject",
             "s3:PutObject"
```

```
            ],
            "Resource": [
                "arn:aws:s3:::aws-gen-ai-glue-metadata-*/*"
            ]
        }
    ]
}
```

- An IAM role for your notebook environment. The IAM role should have the appropriate permissions for AWS Glue, Amazon Bedrock, and Amazon S3. The following is an example policy. You can apply additional conditions to restrict it further for your own environment.

```
{
      "Version": "2012-10-17",
      "Statement": [
           {
                "Sid": "GluePermissions",
                "Effect": "Allow",
                "Action": [
                     "glue:GetCrawler",
                     "glue:DeleteDatabase",
                     "glue:GetTables",
                     "glue:DeleteCrawler",
                     "glue:StartCrawler",
                     "glue:CreateDatabase",
                     "glue:UpdateTable",
                     "glue:DeleteTable",
                     "glue:UpdateCrawler",
                     "glue:GetTable",
                     "glue:CreateCrawler"
                ],
```

- Model access for Anthropic's Claude 3 and Amazon Titan Text Embeddings V2 on Amazon Bedrock.

- The notebook `glue-catalog-genai_claude.ipynb`.

Set up the resources and environment

Now that you have completed the prerequisites, you can switch to the notebook environment to run the next steps. First, the notebook will create the required resources:

- S3 bucket
- AWS Glue database
- AWS Glue crawler, which will run and automatically generate the database tables

After you finish the setup steps, you will have an AWS Glue database called `legislators`.

The crawler creates the following metadata tables:

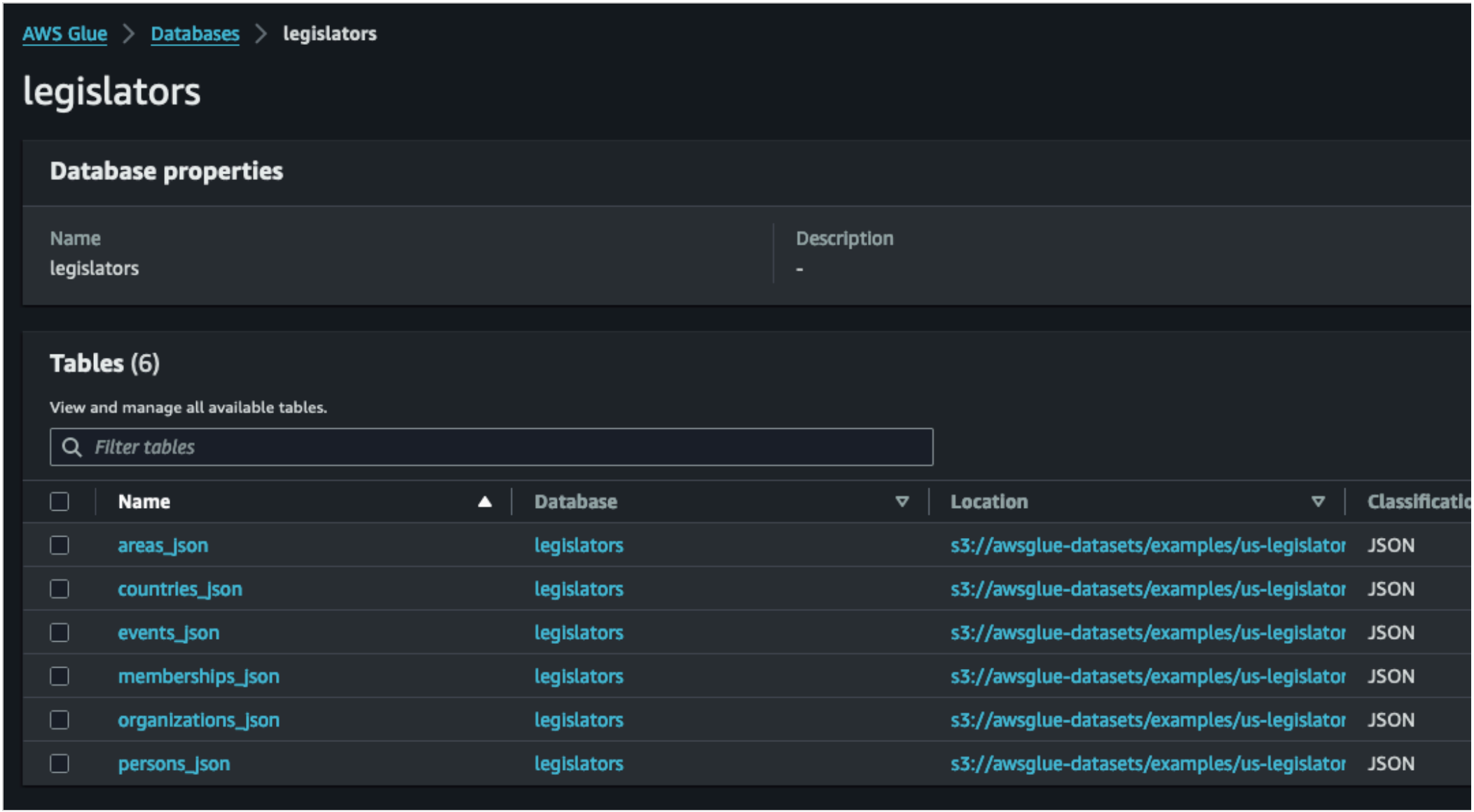
- `persons`
- `memberships`
- `organizations`
- `events`
- `areas`
- `countries`

This is a semi-normalized collection of tables containing legislators and their histories.

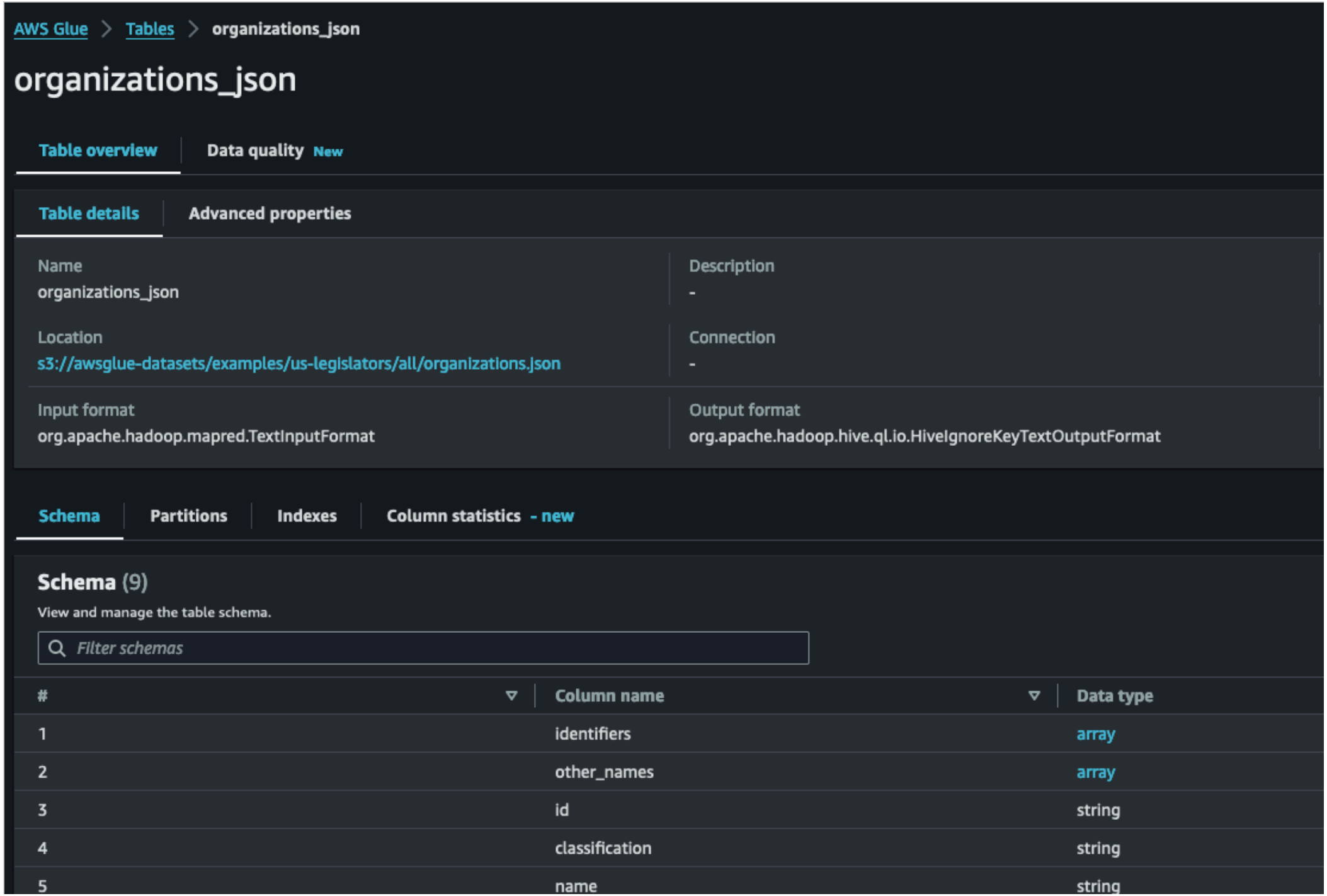
Follow the rest of the steps in the notebook to complete the environment setup. It should only take a few minutes.

Inspect the Data Catalog

Now that you have completed the setup, you can inspect the Data Catalog to familiarize yourself with it and the metadata it captured. On the AWS Glue console, choose **Databases** in the navigation pane, then open the newly created legislators database. It should contain six tables, as shown in the following screenshot:



You can open any table to inspect the details. The table description and comment for each column is empty because they aren't completed automatically by the AWS Glue crawlers.


AWS Glue > Tables > organizations_json
organizations_json
Table overview
Data quality New
Table details
Advanced properties
Name
organizations_json
Description
-
Location
s3://awsglue-datasets/examples/us-legislators/all/organizations.json
Connection
-
Input format
org.apache.hadoop.mapred.TextInputFormat
Output format
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
Schema
Partitions
Indexes
Column statistics - new
Schema (9)
View and manage the table schema.
Filter schemas
Column name Data type
1 identifiers array
2 other_names array
3 id string
4 classification string
5 name string

6	links	array
7	image	string
8	seats	int
9	type	string

You can use the AWS Glue API to programmatically access the technical metadata for each table. The following code snippet uses the AWS Glue API through the AWS SDK for Python (Boto3) to retrieve tables for a chosen database and then prints them on the screen for validation. The following code, found in the notebook of this post, is used to get the data catalog information programmatically.

```
def get_alltables(database):
    tables = []
    get_tables_paginator = glue_client.get_paginator('get_tables')
    for page in get_tables_paginator.paginate(DatabaseName=database):
        tables.extend(page['TableList'])
    return tables

def json_serial(obj):
    if isinstance(obj, (datetime, date)):
        return obj.isoformat()
    raise TypeError ("Type %s not serializable" % type(obj))

database_tables =  get_alltables(database)

for table in database_tables:
    print(f"Table: {table['Name']}")
    print(f"Columns: {[col['Name'] for col in table['StorageDescriptor']['Columns']]}")
```

Now that you're familiar with the AWS Glue database and tables, you can move to the next step to generate table metadata descriptions with generative AI.

Generate table metadata descriptions with Anthropic's Claude 3 using Amazon Bedrock and LangChain

In this step, we generate technical metadata for a selected table that belongs to an AWS Glue database. This post uses the persons table. First, we get all the tables from the Data Catalog and include it as part of the prompt. Even though our code aims to generate metadata for a single table, giving the LLM wider information is useful because you want the LLM to detect foreign keys. In our notebook environment we install LangChain v0.2.1. See the following code:

```
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from botocore.config import Config
from langchain_aws import ChatBedrock

glue_data_catalog = json.dumps(get_alltables(database),default=json_serial)


model_kwargs ={
    "temperature": 0.5, # You can increase or decrease this value depending on the amount of randomness you want injecte
    "top_p": 0.999
}

model = ChatBedrock(
    client = bedrock_client,
    model_id=model_id,
    model_kwargs=model_kwargs
)
```

In the preceding code, you instructed the LLM to provide a JSON response that fits the `TableInput` object expected by the Data Catalog update API action. The following is an example response:

```
{
  "Name": "persons",
```

```
  "Description": "This table contains information about individual persons, including their names, identifiers, contact
  "StorageDescriptor": {
    "Columns": [
      {
        "Name": "family_name",
        "Type": "string",
        "Comment": "The family name or surname of the person."
      },
      {
        "Name": "name",
        "Type": "string",
        "Comment": "The full name of the person."
      },
      {
        "Name": "links",
```

You can also validate the JSON generated to make sure it conforms to the format expected by the AWS Glue API:

```
from jsonschema import validate

schema_table_input = {
    "type": "object",
    "properties" : {
            "Name" : {"type" : "string"},
            "Description" : {"type" : "string"},
            "StorageDescriptor" : {
            "Columns" : {"type" : "array"},
            "Location" : {"type" : "string"} ,
            "InputFormat": {"type" : "string"} ,
            "SerdeInfo": {"type" : "object"}
        }
    }
```

```
}
validate(instance=json.loads(TableInputFromLLM), schema=schema_table_input)
```

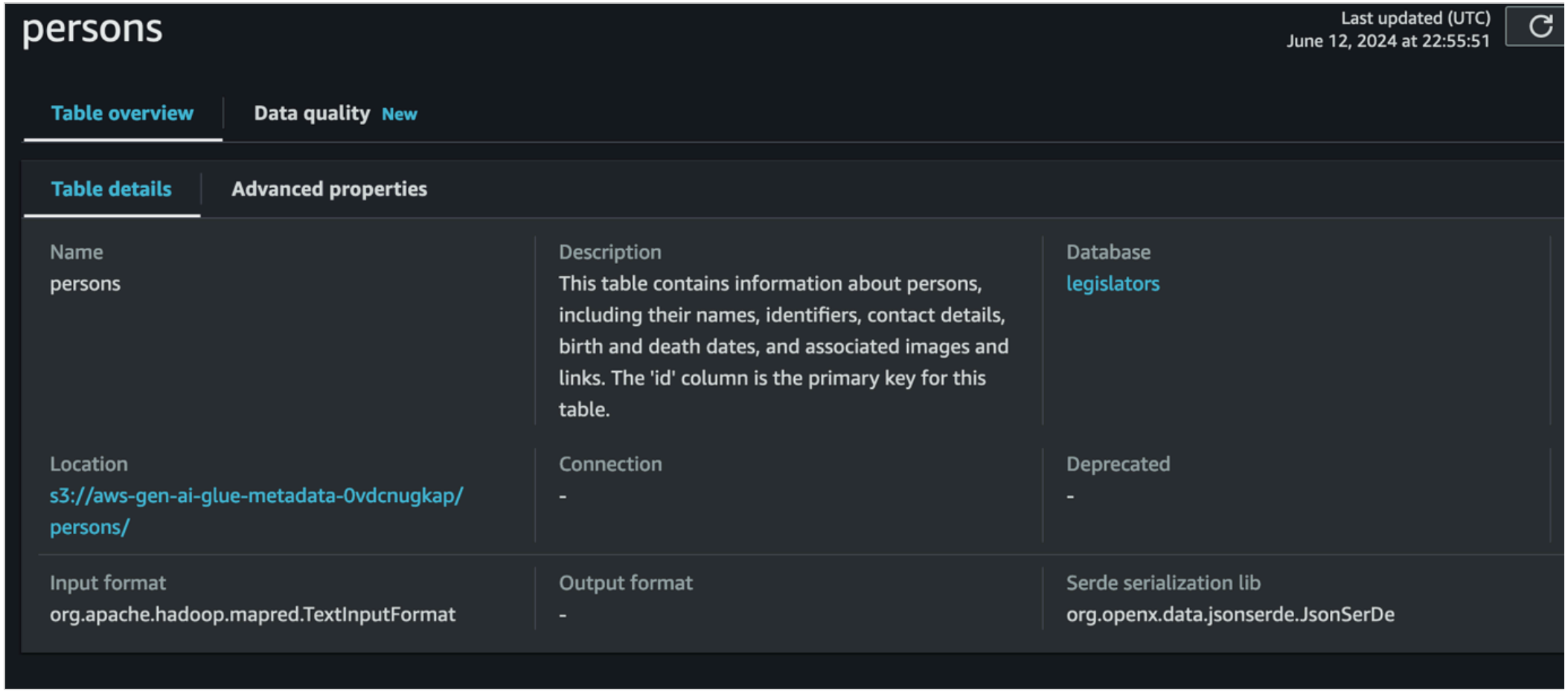
Now that you have generated table and column descriptions, you can update the Data Catalog.

Update the Data Catalog with metadata

In this step, use the AWS Glue API to update the Data Catalog:

```
response = glue_client.update_table(DatabaseName=database, TableInput= json.loads(TableInputFromLLM) )
print(f"Table {table} metadata updated!")
```

The following screenshot shows the persons table metadata with a description.



The following screenshot shows the table metadata with column descriptions.

#	Column name	Data type	Partition key	Comment
1	family_name	string	-	Person's family name
2	name	string	-	Person's full name
3	links	array	-	Links related to the person
4	gender	string	-	Person's gender
5	image	string	-	URL of the person's image
6	identifiers	array	-	Identifiers for the person
7	other_names	array	-	Other names the person may be kno...
8	sort_name	string	-	Name to be used for sorting
9	images	array	-	URLs of additional images of the per...
10	given_name	string	-	Person's given name
11	birth_date	string	-	Person's birth date
12	id	string	-	Unique identifier for the person (Pri...
13	contact_details	array	-	Contact details for the person
14	death_date	string	-	Person's death date (if applicable)

Now that you have enriched the technical metadata stored in Data Catalog, you can improve the descriptions by adding external documentation.

Improve metadata descriptions by adding external documentation with RAG

In this step, we add external documentation to generate more accurate metadata. The documentation for our dataset can be found online as an HTML. We use the LangChain HTML community loader to load the HTML content:

```
from langchain_community.document_loaders import AsyncHtmlLoader

# We will use an HTML Community loader to load the external documentation stored on HTLM
urls = ["http://www.popoloproject.com/specs/person.html", "http://docs.everypolitician.org/data_structure.html",'http://www
```

```
loader = AsyncHtmlLoader(urls)
docs = loader.load()
```

After you download the documents, split the documents into chunks:

```
text_splitter = CharacterTextSplitter(
    separator='\n',
    chunk_size=1000,
    chunk_overlap=200,

)
split_docs = text_splitter.split_documents(docs)

embedding_model = BedrockEmbeddings(
    client=bedrock_client,
    model_id=embeddings_model_id
)
```

Next, vectorize and store the documents locally and perform a similarity search. For production workloads, you can use a managed service for your vector store such as Amazon OpenSearch Service or a fully managed solution for implementing the RAG architecture such as Amazon Bedrock Knowledge Bases.

```
vs = FAISS.from_documents(split_docs, embedding_model)
search_results = vs.similarity_search(
    'What standards are used in the dataset?', k=2
)
print(search_results[0].page_content)
```

Next, include the catalog information along with the documentation to generate more accurate metadata:

```
from operator import itemgetter
from langchain_core.callbacks import BaseCallbackHandler
from typing import Dict, List, Any


class PromptHandler(BaseCallbackHandler):
    def on_llm_start( self, serialized: Dict[str, Any], prompts: List[str], **kwargs: Any) -> Any:
        output = "\n".join(prompts)
        print(output)

system = "You are a helpful assistant. You do not generate any harmful content."
# specify a user message
user_msg_rag = """
Here is the guidance document you should reference when answering the user:

<documentation>{context}</documentation>
I'd like to you create metadata descriptions for the table called {table} in your AWS Glue data catalog. Please follow
```

The following is the response from the LLM:

```
{
  "Name": "persons",
  "Description": "This table contains information about individual persons, including their names, identifiers, contact
  "StorageDescriptor": {
    "Columns": [
      {
        "Name": "family_name",
        "Type": "string",
        "Comment": "The family or last name of the person."
```

```
        },
        {
          "Name": "name",
          "Type": "string",
          "Comment": "The full name of the person."
        },
        {
          "Name": "links"
```

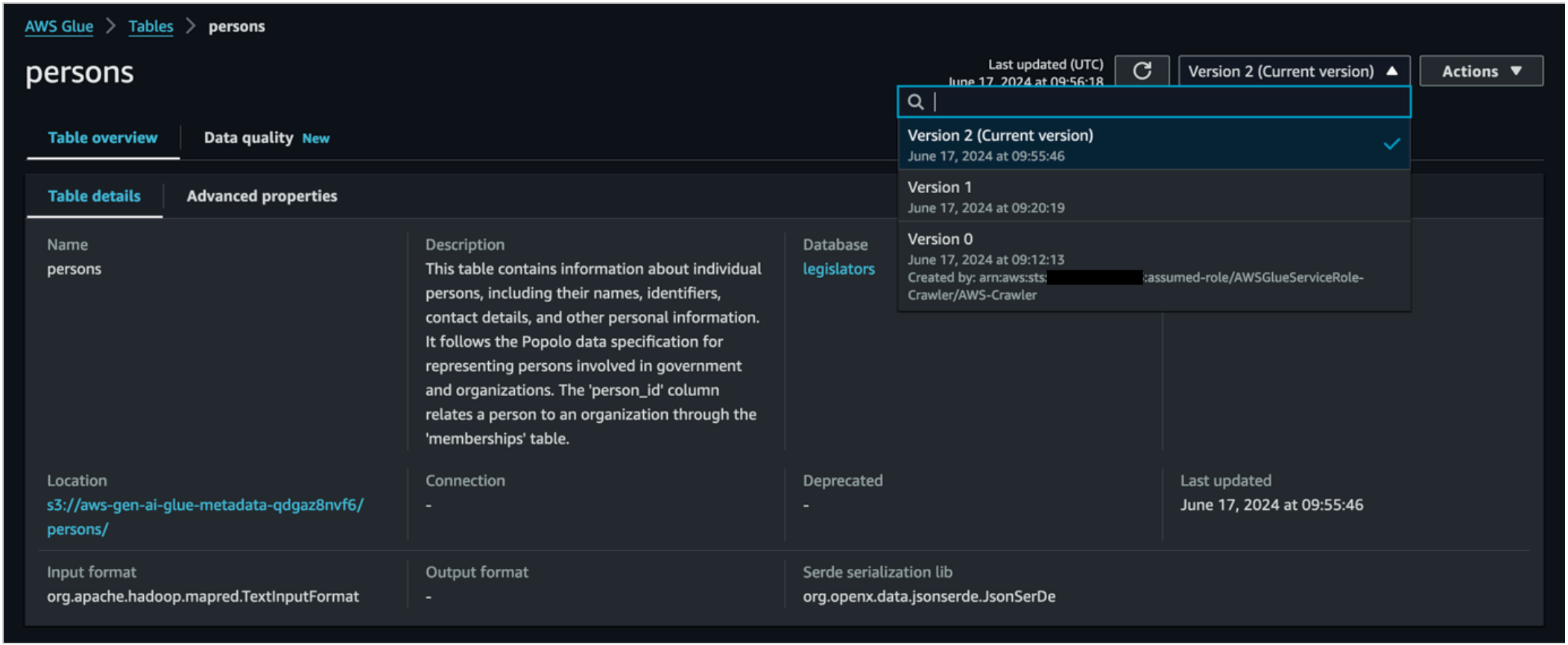
Similar to the first approach, you can validate the output to make sure it conforms to the AWS Glue API.

Update the Data Catalog with new metadata

Now that you have generated the metadata, you can update the Data Catalog:

```
response = glue_client.update_table(DatabaseName=database, TableInput= json.loads(TableInputFromLLM) )
print(f"Table {table} metadata updated!")
```

Let's inspect the technical metadata generated. You should now see a newer version in the Data Catalog for the persons table. You can access schema versions on the AWS Glue console.



Note the `persons` table description this time. It should differ slightly from the descriptions provided earlier:

- **In-context learning table description** – "This table contains information about persons, including their names, identifiers, contact details, birth and death dates, and associated images and links. The 'id' column is the primary key for this table."
- **RAG table description** – "This table contains information about individual persons, including their names, identifiers, contact details, and other personal information. It follows the Popolo data specification for representing persons involved in government and organizations. The 'person_id' column relates a person to an organization through the 'memberships' table."

The LLM demonstrated knowledge around the Popolo specification, which was part of the documentation provided to the LLM.

Clean up

Now that you have completed the steps described in the post, don't forget to clean up the resources with the code provided in the notebook so you don't incur unnecessary costs.

Conclusion

In this post, we explored how you can use generative AI, specifically Amazon Bedrock FMs, to enrich the Data Catalog with dynamic metadata to improve the discoverability and understanding of existing data assets. The two approaches we demonstrated, in-context learning and RAG, showcase the flexibility and versatility of this solution. In-context learning works well for AWS Glue databases with a small number of tables, whereas the RAG approach uses external documentation to generate more accurate and detailed metadata, making it suitable for larger and more complex data landscapes. By implementing this solution, you can unlock new levels of data intelligence, empowering your organization to make more informed decisions, drive data-driven innovation, and unlock the full value of your data. We encourage you to explore the resources and recommendations provided in this post to further enhance your data management practices.

About the Authors

Manos Samatas is a Principal Solutions Architect in Data and AI with Amazon Web Services. He works with government, non-profit, education and healthcare customers in the UK on data and AI projects, helping build solutions using AWS. Manos lives and works in London. In his spare time, he enjoys reading, watching sports, playing video games and socialising with friends.

Anastasia Tzeveleka is a Senior GenAI/ML Specialist Solutions Architect at AWS. As part of her work, she helps customers across EMEA build foundation models and create scalable generative AI and machine learning solutions using AWS services.

Like Share

Comments

Log in to comment