Automating AI-Generated Column Comments in Databricks Using PySpark

Databricks doesn't natively support programmatic generation of AI comments via PySpark or SQL (it's primarily a UI feature in Catalog Explorer). However, you can achieve this by leveraging Databricks' `AI_QUERY` function (which calls an LLM like `databricks-dbrx-instruct`) to generate summaries/descriptions based on column metadata, then apply them using `ALTER TABLE` statements. This workaround is inspired by community and expert implementations.

Below is a complete, adaptable PySpark notebook example to generate and add an AI-generated comment (summary) to **a single specified column** in a Unity Catalog table. It can be easily extended for multiple columns. This assumes:

- You're running in a Databricks notebook with access to Unity Catalog.
- The table exists (e.g., `catalog.schema.table`).
- You have permissions to alter the table.
- An LLM model like `databricks-dbrx-instruct` is available in your workspace.

Step 1: Set Up Parameters (Using Widgets for Reusability)

Run this in a Python cell to parameterize the table and target column:

```
Python

# Use Databricks widgets for easy parameterization (e.g., for workflows)
dbutils.widgets.text("catalog", "main")  # Replace with your catalog
dbutils.widgets.text("schema", "default")  # Replace with your schema
dbutils.widgets.text("table", "your_table")  # Replace with your table
dbutils.widgets.text("target_column", "your_column")  # The column to add AI

catalog = dbutils.widgets.get("catalog")
schema = dbutils.widgets.get("schema")
table = dbutils.widgets.get("table")
target_column = dbutils.widgets.get("target_column")
table_name = f"{catalog}.{schema}.{table}"
```

Step 2: Create SQL Helper Functions (Run in a SQL Cell)

These functions use `AI_QUERY` to generate the AI summary and format it as JSON. Run this in a SQL cell:

```
SQL

-- Helper function to query the LLM
CREATE OR REPLACE TEMPORARY FUNCTION ask_llm(prompt STRING)
RETURNS STRING
LANGUAGE SQL
AS $$
  AI_QUERY(
    "databricks-dbrx-instruct",  -- Or another available model like "meta-lla
    prompt
  )
$$;

-- Function to generate AI description for a specific column
CREATE OR REPLACE TEMPORARY FUNCTION generate_column_description(col_name STR
RETURNS STRING  -- Returns formatted description string directly
LANGUAGE SQL
AS $$
  SELECT
    JSON_EXTRACT_SCALAR(
      ask_llm(
        CONCAT(
          'Generate a concise business summary/description for this table col
          'Column name: ', col_name,
          '. Data type: ', data_type,
          '. Keep it accurate, professional, and under 100 words. ',
          'End with: "AI-generated summary via Databricks on ', CURRENT_DATE(
          'Respond with JSON only: {"description": "<your summary here>"}. No
        )
      ),
      '$.description'
    ) AS generated_desc
$$;
```

Step 3: Generate the AI Comment (Run in a SQL Cell)

Query the column's metadata and generate the AI summary. This creates a temp view with the result:

```
SQL

-- Create temp view with AI-generated description for the target column
CREATE OR REPLACE TEMPORARY VIEW ai_column_summary AS
SELECT
  column_name,
  data_type,
  generate_column_description(column_name, data_type) AS ai_description
FROM ${catalog}.information_schema.columns
WHERE
  table_schema = '${schema}'
  AND table_name = '${table}'
  AND column_name = '${target_column}';
```

Step 4: Apply the Comment Programmatically (Run in a Python Cell)

Fetch the result as a Spark DataFrame and apply it via `ALTER TABLE`. This handles escaping and error logging:

```
Python

# Load the SQL result into a Spark DataFrame
ai_summary_df = spark.sql("SELECT * FROM ai_column_summary")

# Check if a description was generated
if ai_summary_df.count() > 0:
    row = ai_summary_df.collect()[0]
    column_name = row['column_name']
    ai_description = row['ai_description']

    # Escape single quotes for SQL safety
    escaped_comment = ai_description.replace("'", "''")

    # Apply the comment
    try:
        spark.sql(f"""
            ALTER TABLE {table_name}
            ALTER COLUMN {column_name}
            COMMENT '{escaped_comment}'
        """)
        print(f"Successfully added AI-generated comment to column '{column_na
    except Exception as e:
        print(f"Error applying comment to '{column_name}': {str(e)}")
        print("Check table permissions or if the column already has a comment
else:
    print(f"No metadata found for column '{target_column}' in table '{table_n
```

How It Works

1. **Parameterization**: Widgets make it reusable (e.g., trigger via Databricks Workflows).
2. **AI Generation**: `AI_QUERY` prompts an LLM with the column name and data type to create a concise summary. The response is parsed from JSON.
3. **Metadata Query**: Pulls from `information_schema.columns` for accuracy.
4. **Application**: Uses PySpark's `spark.sql` to execute `ALTER TABLE`, which updates the Unity Catalog comment.
5. **Customization**:
    - For **multiple columns**: Replace the `WHERE` clause in Step 3 with `column_name IN ('col1', 'col2')` or remove it entirely, then loop over `ai_summary_df.collect()` in Step 4.
    - **Check Existing Comments**: Before applying, query `information_schema.columns` for `column_comment IS NULL OR column_comment = ''` to filter.
    - **Model Selection**: Swap `"databricks-dbrx-instruct"` for other models available in your workspace (e.g., via `/models` endpoint).
    - **Prompt Tuning**: Edit the prompt in `generate_column_description` for more specific summaries (e.g., domain-specific language).

Example Output

If your column is `customer_age` (type `INT`), the AI might generate:
"This column stores the age of customers in years, suitable for demographic analysis and segmentation. AI-generated summary via Databricks on 2025-12-03."

Run this in sequence in a Databricks notebook. If you encounter LLM access issues, ensure your workspace has AI features enabled. For UI-only generation, use Catalog Explorer > Table > Columns > AI Generate button.