Project 6: Dockerized Python Data Pipeline from S3 to RDS with AWS Glue Fallback

Intern: Shyam Champakbhai Chotaliya **Company:** Fortune Cloud Technologies **Date:** [14-07-2025]

Project Summary

This project involved building a resilient, containerized ETL (Extract, Transform, Load) application using Docker and Python. The application reads a CSV file from an Amazon S3 bucket and attempts to load the data into a primary MySQL database hosted on Amazon RDS. If the RDS instance is unreachable for any reason, the application automatically triggers a fallback mechanism, creating a metadata table in the AWS Glue Data Catalog that points to the source data in S3. This ensures data is never lost and its schema is registered for future processing.

Tools and Services Used

- **Containerization:** Docker
- **Programming:** Python 3.9
- **Python Libraries:** Boto3 (AWS SDK), Pandas (Data Manipulation), SQLAlchemy (Database ORM), PyMySQL (MySQL Driver)
- **Data Storage (Source):** Amazon S3
- **Primary Database (Target):** Amazon RDS (MySQL)
- **Fallback Metadata Store:** AWS Glue Data Catalog
- **Security:** AWS IAM (Identity and Access Management) Role

---

Step-by-Step Setup Instructions

1. **AWS Environment Setup:**
    - An **S3 Bucket** was created to store the source `students.csv` file.
    - An **RDS MySQL** instance (Free Tier) was provisioned to act as the primary database. A `studentdb` database and a `students` table were created.
    - An **AWS Glue Data Catalog Database** named `student_data_fallback` was created to serve as the destination for the fallback mechanism.
    - An **IAM Role** was created and attached to the host EC2 instance, granting specific, least-privilege permissions to access S3 and Glue.

2. **Application Development:**
    - A Python script (`main.py`) was written to orchestrate the ETL logic using `boto3`, `pandas`, and `SQLAlchemy`.
    - **Fallback Logic:** The script is wrapped in a `try...except` block. It first attempts to connect to RDS. If any exception occurs (e.g., connection timeout), it catches the error, logs a warning, and proceeds to the fallback function, which uses `boto3` to create a table in the AWS Glue Data Catalog.
3. **Dockerization:**
    - A `Dockerfile` was created to define the container image. It starts with a Python base image, installs all dependencies from `requirements.txt`, copies the application script, and sets the `CMD` to run the script on container startup.
    - The Docker image was built using `docker build -t data-pipeline-app .`
    - The container was run using `docker run data-pipeline-app`. Because it runs on an EC2 instance with an IAM Role, no credentials needed to be passed.

Data Flow

The application's data flow is designed with a primary path and a fallback path to handle potential database failures.

1. Primary Data Flow (Happy Path):

`S3 Bucket (students.csv)` → `Docker Container (Python App)` → `Parse CSV with Pandas` → `Connect to RDS` → `Insert Data into MySQL Table`

- The process begins when the Docker container starts.
- The Python script uses `boto3` to download `students.csv` from S3.
- The CSV data is parsed into a pandas DataFrame.
- The script establishes a connection to the RDS MySQL database using `SQLAlchemy`.
- The data is appended to the `students` table in the `studentdb` database.

2. Fallback Data Flow (RDS Unreachable):

`S3 Bucket (students.csv)` → `Docker Container (Python App)` → `Parse CSV` → `RDS Connection Fails` → **`Create Table in AWS Glue Data Catalog`**

- If the initial attempt to connect to RDS fails (e.g., the database is stopped or there's a network issue), the script's `except` block is triggered.
- The script then uses `boto3` to connect to the AWS Glue service.
- It creates a new table metadata entry in the Glue Data Catalog. This table definition points to the original CSV file's location in S3, making the data queryable via services like Amazon Athena.

Docker Setup

The application was packaged into a portable and self-sufficient container using Docker to ensure consistency across different environments.

- **`Dockerfile`:** A `Dockerfile` was created to define the container image. It performs the following steps:
    1. Starts from an official `python:3.9-slim` base image.
    2. Sets `/app` as the working directory.
    3. Copies the `requirements.txt` file and installs all Python dependencies (`boto3`, `pandas`, `SQLAlchemy`, `PyMySQL`) using `pip`.
    4. Copies the `main.py` application script into the container.
    5. Defines the startup command `CMD ["python", "main.py"]` to execute the script when the container is run.
- **Build & Run:** The image was built using `docker build -t data-pipeline-app .` and run using `docker run data-pipeline-app`.

Testing Strategy

The system's resilience was validated using two distinct scenarios:

1. **"Happy Path" Test (RDS Available):** The Docker container was run while the RDS instance was "Available". The application successfully connected to RDS and inserted the data from the CSV. This was verified by querying the `students` table in the RDS database.
2. **"Failure Path" Test (RDS Unreachable):** The RDS instance was manually **stopped** from the AWS console to simulate a failure. The Docker container was run again. The application logs showed a connection error, followed by the successful execution of the Glue fallback logic. This was verified by checking the AWS Glue Data Catalog for the newly created `students_fallback` table.

IAM Permissions

An IAM Role with a custom policy was used to provide the application with secure access. The policy granted:

- `s3:GetObject` & 'S3:PutObject' permission on the specific source CSV file.
- `glue:CreateTable` and `glue:GetDatabase` permissions scoped to the specific Glue Catalog database.
- **Note:** RDS connection permissions were managed via network security groups (allowing port 3306) and database credentials, not via IAM policy for this standard setup.

Challenges Faced and How They Were Solved

1. **Challenge: RDS Connectivity from within a Docker Container.**
    - **Problem:** The initial attempts to connect to the RDS instance from the running Docker container resulted in "Connection timed out" errors.

- **Solution:** The problem was identified as a network access issue. It was resolved by modifying the **Security Group** attached to the RDS instance to add a new **inbound rule** allowing traffic on port `3306` (MYSQL/Aurora) from source `0.0.0.0/0` (Anywhere). This opened the necessary port for the EC2-hosted container to connect.

2. **Challenge: Providing AWS Credentials Securely to the Container.**

   - **Problem:** The `boto3` library inside the container could not find AWS credentials to access S3 and Glue. Hardcoding credentials in the script or Dockerfile is a major security risk.
   - **Solution:** The best practice of using an **IAM Role** was implemented. An IAM Role with the necessary permissions for S3 and Glue was created and attached to the host EC2 instance. The Docker container automatically inherited these credentials from the EC2 metadata service, allowing for secure and seamless authentication.

3. **Challenge: Incompatible Data Types between Pandas and AWS Glue.**

   - **Problem:** The AWS Glue `CreateTable` API call was failing because the data types inferred by pandas (e.g., `int64`, `object`) are not directly understood by the Glue API.
   - **Solution:** A Python dictionary was created in the `main.py` script to **map pandas data types to their corresponding Glue data types** (e.g., `int64` → `int`, `object` → `string`). The script now iterates through the pandas DataFrame columns and uses this map to build a valid schema before making the API call, resolving the error.

Github Link

https://github.com/Shyamchotaliya/Data-Ingestion-from-S3-to-RDS-with-Fallback-to-AWS-Glue-using-Dockerized-Python-Application (https://github.com/Shyamchotaliya/Data-Ingestion-from-S3-to-RDS-with-Fallback-to-AWS-Glue-using-Dockerized-Python-Application)

List of Submitted Files

- **README.md:** This report.
- **main.py:** The core Python application script.
- **Dockerfile:** The script to build the container image.
- **requirements.txt:** List of Python dependencies.
- **/screenshots/:** A folder containing screenshots of the working application.
- **/sample-iam-policies/:** Folder containing sample IAM JSON policies.

> ***Note:*** *This document covers only Project 6 and marks the completion of all assigned internship projects.*