API-driven Cloud Native Solutions (S1-25_CCZG506)

Assignment I

Table of Content

1 Data Pipeline

1.1 Business Understanding

This section should briefly explain the real-world context and why the problem matters. In the highly competitive retail sector, forecasting product demand accurately is critical for ensuring optimal inventory levels, minimising wastage, and maximising customer satisfaction. Retailers often struggle with stockouts or overstocking due to demand fluctuations, seasonality, and consumer trends.

This project focuses on building a demand forecasting solution using historical sales data for 50 items across 10 stores over a 5-year period (from 2013 to 2018). The dataset is sourced from Kaggle's Store Item Demand Forecasting Challenge, which closely mimics real-world retail demand patterns.

The objective is to design and implement a robust data pipeline that:

- Ingests the raw historical sales data
- Cleans, transforms, and prepares it for analysis
- Enables Exploratory Data Analysis (EDA)
- Supports Machine Learning models to forecast future item demand

This pipeline will demonstrate how cloud-native serverless tools like AWS Glue, S3, Lambda, and CloudWatch can be used to build scalable, automated demand forecasting solutions aligned with modern retail business needs.

Overall Architecture

Building on the business problem described above, the overall system architecture has been designed to integrate key AWS services into a unified, automated, and serverless data processing and machine learning workflow. This architecture enables seamless data ingestion, transformation, model training, and continuous monitoring within a cloud-native environment.

The process begins with **Amazon S3**, which serves as the central data lake for storing raw sales data obtained from Kaggle. The data is then processed by **AWS Glue**, where cleaning,

transformation, and preprocessing are performed using PySpark scripts. These steps prepare the data for analysis and model training.

Following preprocessing, the system performs **Exploratory Data Analysis (EDA)** and proceeds to **Machine Learning model training** within AWS Glue. Algorithms such as **Linear Regression** and **Random Forest Regressor** are trained on the prepared dataset to forecast product demand, and the resulting predictions are stored back in S3 for validation and evaluation.

Automation is achieved using **AWS EventBridge**, which schedules the pipeline to execute every two minutes, ensuring real-time updates and retraining as new data becomes available. **Amazon CloudWatch** monitors all stages of the workflow, capturing logs, performance metrics, and execution details for observability and fault tracking.

Finally, an **AWS Lambda function** exposes the system's status and outputs through **API-based access**, allowing users to query application details such as job status, start time, duration, and model performance metrics. This integrated design demonstrates how cloud-native tools can be orchestrated to build a scalable, reliable, and fully automated demand forecasting solution suitable for real-world retail environments.

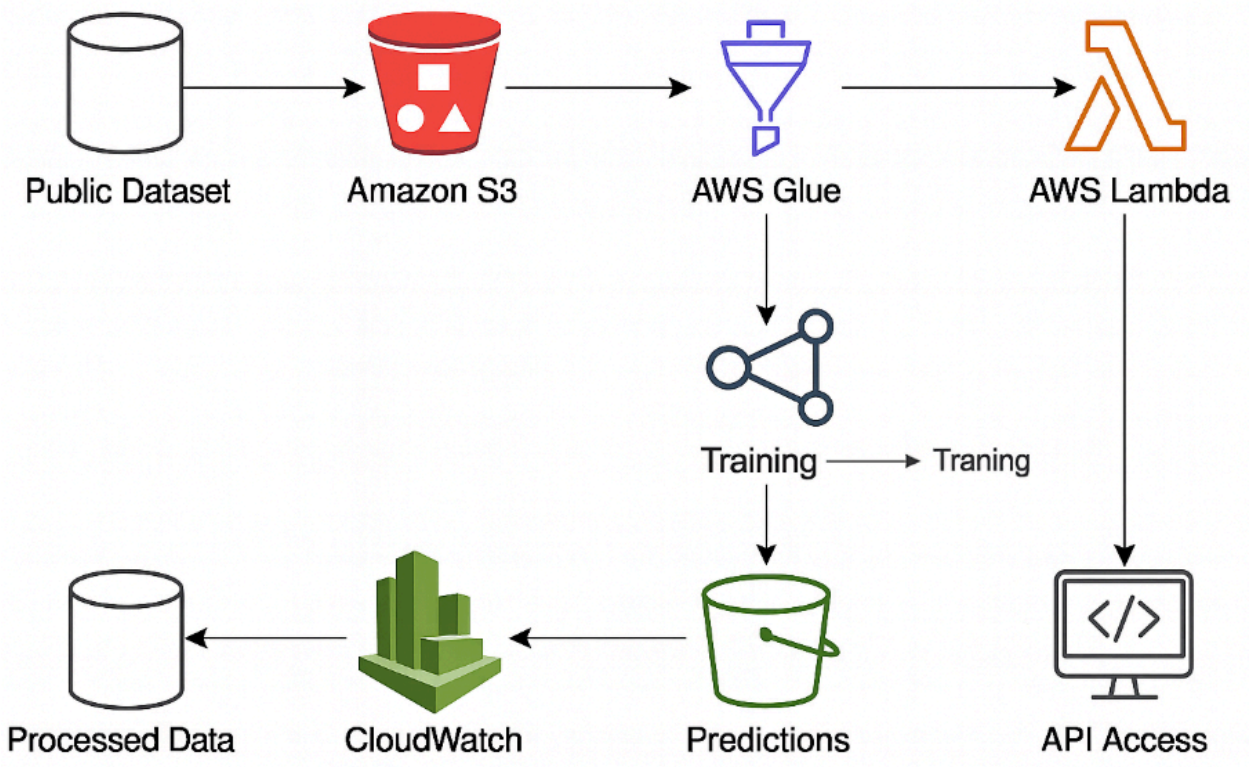


Fig 1: Overall Architecture of the ML Pipeline

Dataset Source

Kaggle - Store Item Demand Forecasting Challenge

1.2 Data Ingestion from Public Dataset (Kaggle)

For this project, we selected the publicly available dataset from the Kaggle Store Item Demand Forecasting Challenge. This dataset provides historical daily sales figures for 50 unique items sold across 10 different stores over a 5-year period (2013–2018), simulating realistic demand fluctuations in the retail industry.

Dataset Source

Platform: Kaggle
Challenge: Store Item Demand Forecasting
Objective: Predict 3 months of item-level daily sales at different store locations.
Size: 18.7 MB (3 CSV files)

Download Procedure

Once you land on the Kaggle competition page:

1. Click the Data tab in the top navigation bar (see screenshot below).
2. On the right-hand side, locate the Data Explorer section.
3. Click **Download All** to get all relevant CSV files:
    - train.csv (training data)
    - test.csv (test data)
4. If prompted, accept the competition rules and terms to enable the download.

KAGGLE · PLAYGROUND CODE COMPETITION · 7 YEARS AGO

Late Submission

Store Item Demand Forecasting Challenge

Predict 3 months of item sales at different stores

Overview Data Code Models Discussion Leaderboard Rules Team Submissions

Dataset Description

The objective of this competition is to predict 3 months of item-level sales data at different store locations.

File descriptions

- **train.csv** - Training data
- **test.csv** - Test data (Note: the Public/Private split is time based)
- **sample_submission.csv** - a sample submission file in the correct format

Data fields

- **date** - Date of the sale data. There are no holiday effects or store closures.
- **store** - Store ID
- **item** - Item ID
- **sales** - Number of items sold at a particular store on a particular date.

Files
3 files

Size
18.7 MB

Type
csv

License
Subject to Competition Rules

Fig 2: Data tab location on Kaggle challenge page

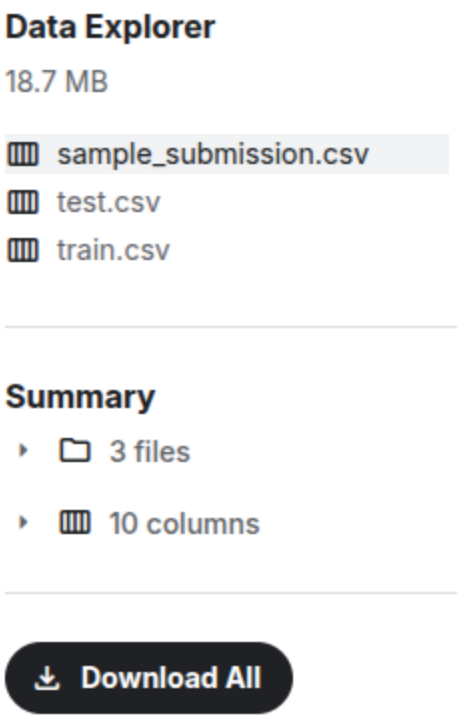


Fig 2: Data Explorer section showing files available for download

Upload to S3

Next step in the process is to upload the dataset in S3, our chosen data lake. To be able to upload the dataset in S3, let's first create the S3 bucket and the directory structure.

Create S3 Bucket for Ingestion

Let's create a dedicated S3 bucket to store raw and processed data. Create the following bucket in S3:

Bucket Name: scdf-project-data

And then create the following folder structure inside the bucket:

```
s3://scdf-project-data/
    └── raw/
    └── processed/
    └── training/
        ├── train.csv/
```

```
└── test.csv/
```

This structure reflects a standard data lake zone architecture:

- raw/ → for unprocessed source files
- processed/ → for cleaned Parquet outputs
- training/ → for split CSVs (train.csv, test.csv) used in ML workflows

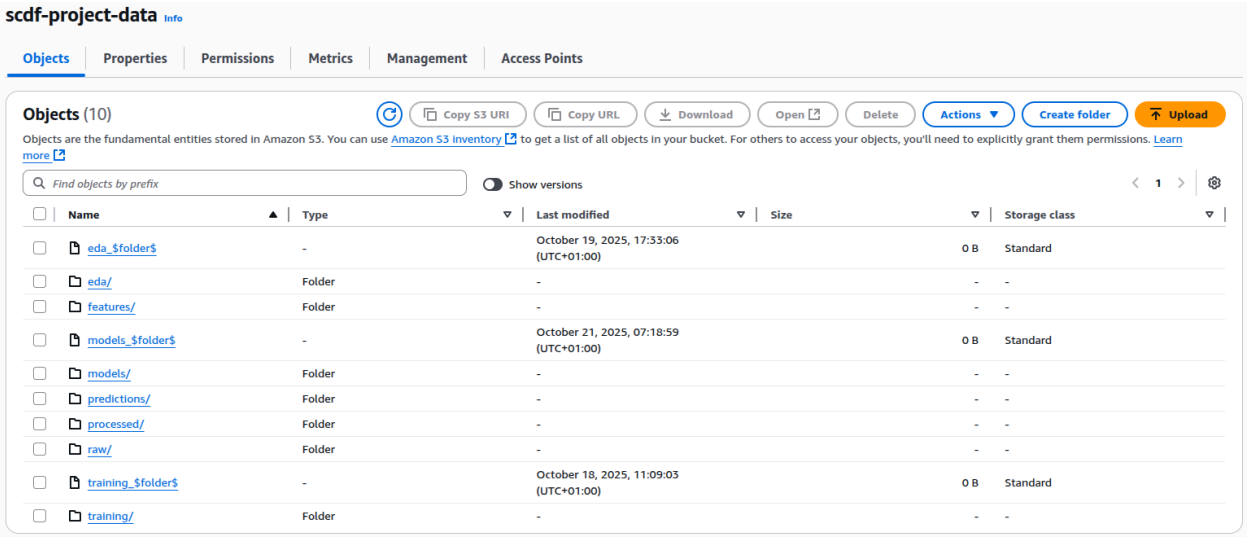


Fig 3: S3 bucket and folder structure

Upload to S3 (raw/)

You need to upload only the two relevant files — train.csv and test.csv — to the raw/ folder using the AWS Console. This setup allows downstream services like AWS Glue to access the data directly for transformation and analysis.

1.3 Data Preprocessing

Now that the data has been ingested and made available in the `raw/` folder in S3, we need to develop a serverless data preprocessing pipeline using **AWS Glue**, written in **PySpark**.

Key Objectives

The preprocessing pipeline aims to:

- Remove nulls or corrupted rows (if any)
- Ensure correct data types

- Normalise numerical values for ML training
- Split data into train and test subsets

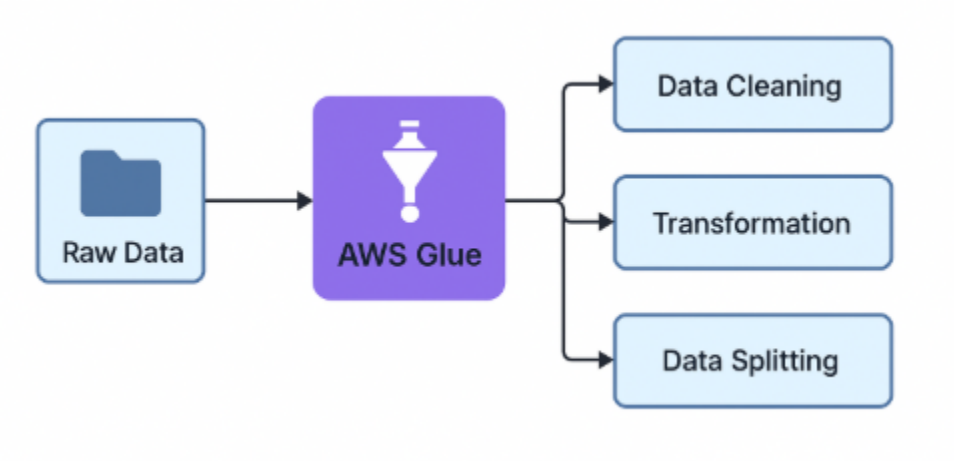


Fig 4: Conceptual model of Data Preprocessing

Overview of AWS Glue and PySpark

AWS Glue is a fully managed serverless data integration service provided by Amazon Web Services. It is designed for building, running, and orchestrating extract-transform-load (ETL) pipelines at scale. Glue simplifies the data engineering workflow by automatically provisioning resources, managing dependencies, and scaling execution environments without the need to manage infrastructure. It supports both visual (no-code) and code-based job authoring, and it integrates seamlessly with other AWS services like S3, Athena, Lambda, and CloudWatch. Glue also includes a metadata catalog to track and query datasets across the data lake.

PySpark, the Python API for Apache Spark, provides a distributed computing engine for processing large-scale data efficiently. It allows for fast, parallelised ETL operations across a cluster of virtual nodes. With PySpark, developers can express complex transformations and analytics using familiar Python syntax, while benefiting from Spark's underlying performance and scalability. It supports SQL queries, machine learning pipelines, and streaming data — all of which are useful in building modern, cloud-native data workflows.

Why We Chose AWS Glue with PySpark

This project requires an automated, cloud-native pipeline to handle ingestion, cleaning, transformation, and train-test splitting on a moderately sized dataset (91,000+ rows). **AWS**

Glue was chosen because it provides a serverless, scalable ETL engine that requires no infrastructure setup, making it ideal for periodic batch jobs. It also offers native support for reading and writing from S3, integrating with IAM, and logging via CloudWatch — all essential for a secure and observable data pipeline.

PySpark was selected as the execution engine within Glue due to its performance and flexibility. It allows us to perform type casting, missing value handling, Min-Max scaling, and dataset partitioning using concise, readable Python code. The ability to leverage Spark's `.randomSplit()` and MLlib transformers (like `MinMaxScaler`) made it ideal for preprocessing tasks in a demand forecasting context, with the added benefit of being easily scalable for larger datasets in future deployments.

Data Preprocessing with AWS Glue

Once the dataset has been ingested into S3, the next logical step is to prepare the data for analysis and machine learning. This phase is called data preprocessing — and it typically includes:

- Cleaning the data (e.g. handling missing values)
- Ensuring correct data types
- Scaling or normalising numerical features
- Structuring the dataset for training and testing (splitting)

Our goal here is to build a repeatable, serverless, and cloud-native preprocessing pipeline that reads from the S3 raw/ folder, performs these transformations, and outputs cleaned, structured data to S3 processed/ and training/.

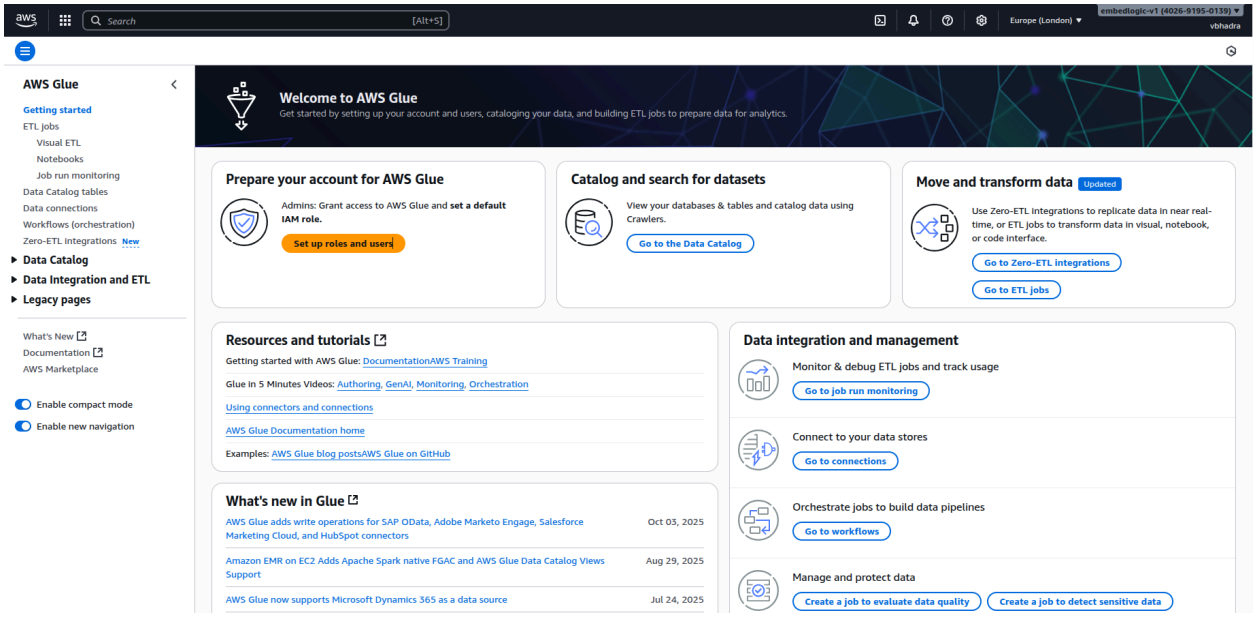
For our purposes we have chosen AWS Glue. We chose **AWS Glue** because of the following reasons:

- It is **serverless** — no infrastructure to manage
- It supports **PySpark**, allowing efficient, distributed processing using familiar Python syntax
- It natively integrates with **S3**, **CloudWatch**, and **IAM**
- It's ideal for ETL jobs that run periodically or in response to ingestion events

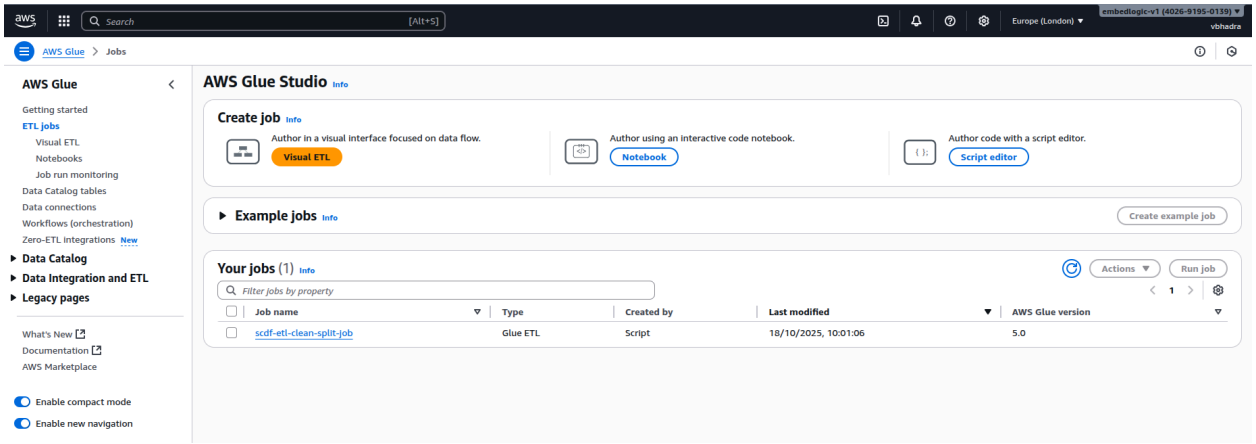
This makes Glue perfect for automating preprocessing pipelines in a scalable and cost-effective way.

Creating Glue Job in AWS

The next step is to create a Glue ETL job in AWS. For that go to Glue in AWS console.



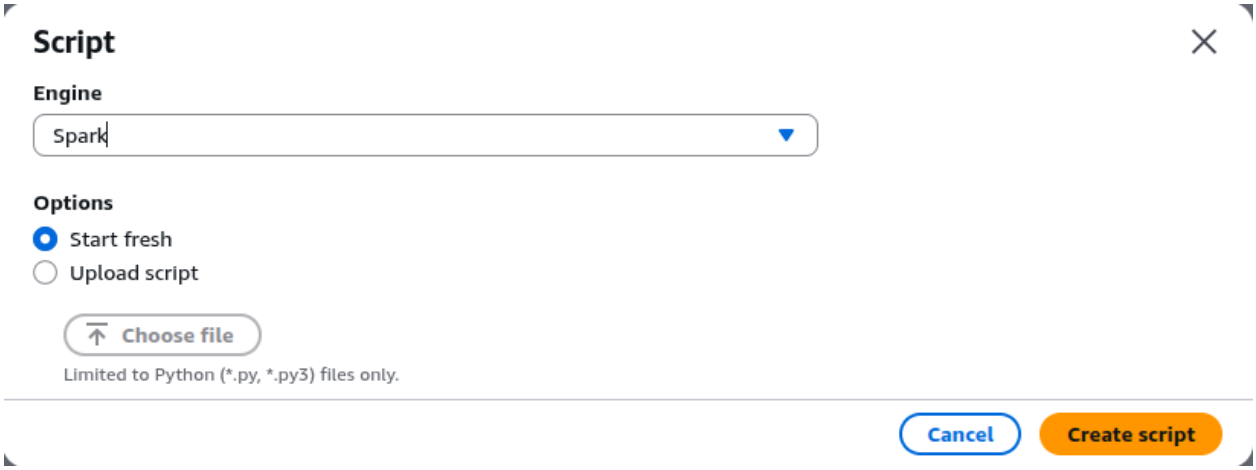
One the right hand pane, navigate to the ETL Jobs as shown in the figure below:



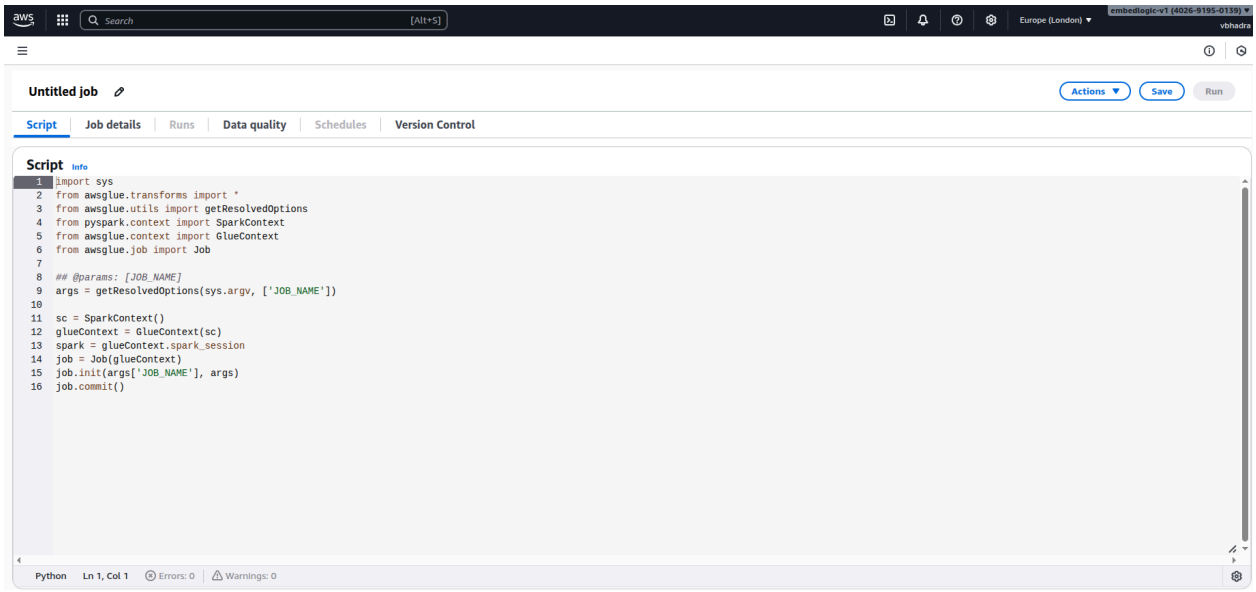
There are three ways you can create an Glue ETL job:

1. Visual ETL
2. Notebook
3. Script Editor

We will use **Script Editor**. Click on the Script Editor icon on the right. It will ask to choose Engine to use, select Spark and in Options Start Fresh. Both are thankfully default options.



Then click the "**Create script**" button. This will open the script editor as shown in the figure below:



Name the script something which you can relate to this task in hand. In my case I named it: **scdf-etl-clean-split-job**.

Python ETL Script for Glue Job

To implement this job, we wrote a PySpark-based ETL script that performs the following:

- Loads the train.csv dataset from the S3 raw/ folder
- Drops any rows with missing or null values

- Casts all relevant columns to appropriate data types (e.g. sales to int, date to date)
- Writes cleaned data to the processed/ folder in Parquet format
- Splits the data randomly into 80% training and 20% testing
- Writes both train and test splits as CSVs into the training/ folder

GitHub Source

The full script is hosted on GitHub and can be downloaded here:

📎 Download `scdf-etl-clean-split-job.py`

You can copy and paste the code into the Glue Studio Script Editor, or upload it as a `.py` file into an S3 location and reference it from the job.

Developing the Python ETL Script

The Python script below implements a complete **Extract–Transform–Load (ETL)** pipeline using **AWS Glue** and **PySpark**. It reads raw sales data from Amazon S3, cleans it, converts it to a more efficient storage format, splits it into training and testing subsets, and writes the results back to S3 — all while maintaining Glue's job-tracking capabilities.

Job Initialisation

The script we are developing begins by importing the necessary libraries and setting up the AWS Glue job environment. This includes `SparkContext`, `GlueContext`, and `Job`, which collectively provide the execution context for distributed ETL operations.

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job
from pyspark.sql.functions import col
from datetime import datetime
```

The job parameters (like the job name) are retrieved from Glue's command-line arguments, and the Spark and Glue contexts are initialised.

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

This setup ensures that the script runs within AWS Glue's distributed Spark infrastructure and is properly registered for monitoring and logging.

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
```

The getResolvedOptions() function reads from sys.argv. When AWS Glue runs the job, it automatically appends `--JOB_NAME scdf-etl-clean-split-job` to `sys.argv`.

Reading the Raw Data from S3

Once the environment is ready, the script reads the input dataset from the **raw zone** of the S3 bucket.
The path points to the uploaded Kaggle sales dataset (`base_sales.csv`).

```
input_path = "s3://scdf-project-data/raw/base_sales.csv"
df = spark.read.option("header", "true").csv(input_path)
```

This step ingests the CSV file into a Spark DataFrame, allowing distributed operations across the cluster. The `header=true` option ensures that column names from the file's first row are retained for ease of reference.

Data Cleaning and Transformation

Before analysis, basic cleaning is performed to remove missing or invalid entries.
Here, `dropna()` is used as a minimal preprocessing step.

```
df_cleaned = df.dropna()
```

This ensures the dataset remains consistent and reliable for downstream tasks. Additional cleaning steps like type casting or outlier removal could be added later as part of the DataOps phase.

Writing the Cleaned Data to the Processed Zone

Cleaned data is then written to the **processed/** folder in **Parquet format**, which offers column-based compression and faster analytics performance.

```
processed_path = "s3://scdf-project-data/processed/"
df_cleaned.write.mode("overwrite").parquet(processed_path)
```

The **processed zone** forms a clean, structured data layer — a key component of a production-ready data lake.

Splitting the Data for Model Training and Testing

Once the dataset is cleaned, the next step is to divide it into two parts — one for training the machine learning model, and another for testing how well the model performs on unseen data.

We use an 80:20 ratio, meaning 80% of the data goes into training and 20% is reserved for testing. Splitting the data is a standard practice in machine learning to evaluate model performance objectively. The model is trained on one portion (training set) and then tested on data it hasn't seen before (test set). This helps detect issues like **overfitting** — where a model performs well on known data but poorly on new inputs — and ensures the solution generalises well to real-world scenarios. Using a fixed split also ensures consistency across experiments and makes comparisons fair and repeatable.

```
train_df, test_df = df_cleaned.randomSplit([0.8, 0.2], seed=42)
```

Here, the `randomSplit()` function does exactly what it says — it randomly divides the dataset. The `seed=42` ensures that this split is reproducible every time the job runs. This is important because consistency across pipeline runs helps avoid surprises and ensures fairness when evaluating model performance.

Writing Training and Testing Data to S3

The resulting datasets are stored separately under the **training/** directory in S3. Each subset is saved as CSV for easy integration with ML frameworks such as SageMaker or scikit-learn.

```
output_prefix = "s3://scdf-project-data/training/"
train_df.write.mode("overwrite").option("header", "true").csv(output_prefix
```

```
+ "train.csv")
test_df.write.mode("overwrite").option("header", "true").csv(output_prefix
+ "test.csv")
```

This ensures modularity — analytical and ML pipelines can directly consume data from clearly defined S3 paths.

Finalising and Committing the Job

Finally, the Glue job is **committed** to mark successful completion.
This step is important for job tracking and triggering dependent workflows within AWS Glue.

```
job.commit()
```

IAM Role and Policy Setup for AWS Glue ETL

To enable our Glue job to securely interact with AWS services like Amazon S3 and AWS Glue itself, we first created a dedicated IAM role. This role is assumed by AWS Glue at runtime and must include all necessary permissions for reading/writing data, accessing scripts, and running jobs successfully.

Create a Custom IAM Role for AWS Glue

Before running any Glue job, you need to assign it a role with the correct trust relationship and access policies. Here's how we created our role:

Create the IAM Role

1. Navigate to the IAM Console:
   https://console.aws.amazon.com/iam
2. Click “Roles” → “Create role”
3. Choose Trusted Entity:
    - Select “AWS service”
    - Use case: Choose “Glue”
4. Click “Next” to skip permissions for now (we’ll attach custom ones shortly)

5. Name your role something descriptive: scdf-ingest-simulator-role
6. Click **"Create role".** This role will allow the Glue service to assume it during job execution.

Attach Custom IAM Policies

Once the role is created, we attached two inline policies:

Policy 1: AllowS3IngestOps

Purpose

This policy allows the Glue job to:

- Read raw input files from raw/
- Write cleaned and split data to processed/ and training/
- List the S3 bucket
- Handle special $folder$ marker objects that Glue sometimes writes automatically

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:DeleteObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::scdf-project-data",
                "arn:aws:s3:::scdf-project-data/raw/*",
                "arn:aws:s3:::scdf-project-data/processed/*",
                "arn:aws:s3:::scdf-project-data/training/*",
                "arn:aws:s3:::scdf-project-data/processed_$folder$",
                "arn:aws:s3:::scdf-project-data/training_$folder$"
            ]
        }
```

```
    ]
}
```

Policy 2: AllowGlueAssetsAccess

Purpose

This policy allows AWS Glue to access its **own internal assets bucket**, such as:

- Script files
- Job metadata
- Dependencies stored in AWS-managed S3 locations

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::aws-glue-assets-402691950139-eu-west-2",
                "arn:aws:s3:::aws-glue-assets-402691950139-eu-west-2/*"
            ]
        }
    ]
}
```

Policy 3: AllowCloudWatchLogsForGlue

If you would like to generate CloudWatch logs after the Glue job executes you need to attach the following **CloudWatch log permissions** to your IAM role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
```

```
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": [
        "arn:aws:logs:eu-west-2:402691950139:log-group:/aws-glue/jobs/*"
      ]
    }
  ]
}
```

How to Attach the Policies

For each policy:

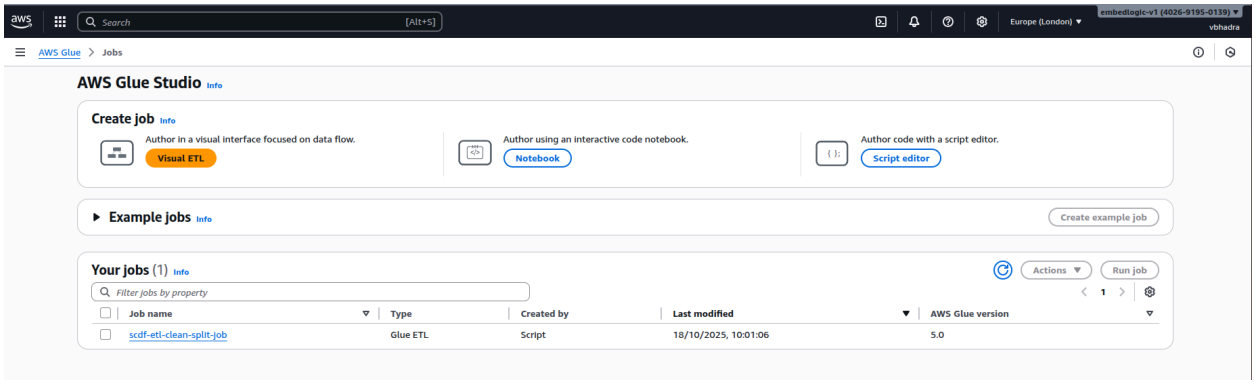
1. Go to IAM → Roles
2. Click your role: **scdf-ingest-simulator-role**
3. Scroll to Permissions → Add inline policy
4. Click “JSON” tab, paste the policy content
5. Click “Review policy”, give it a name (e.g., AllowS3IngestOps)
6. Click Create policy

Repeat for the second and third policy.

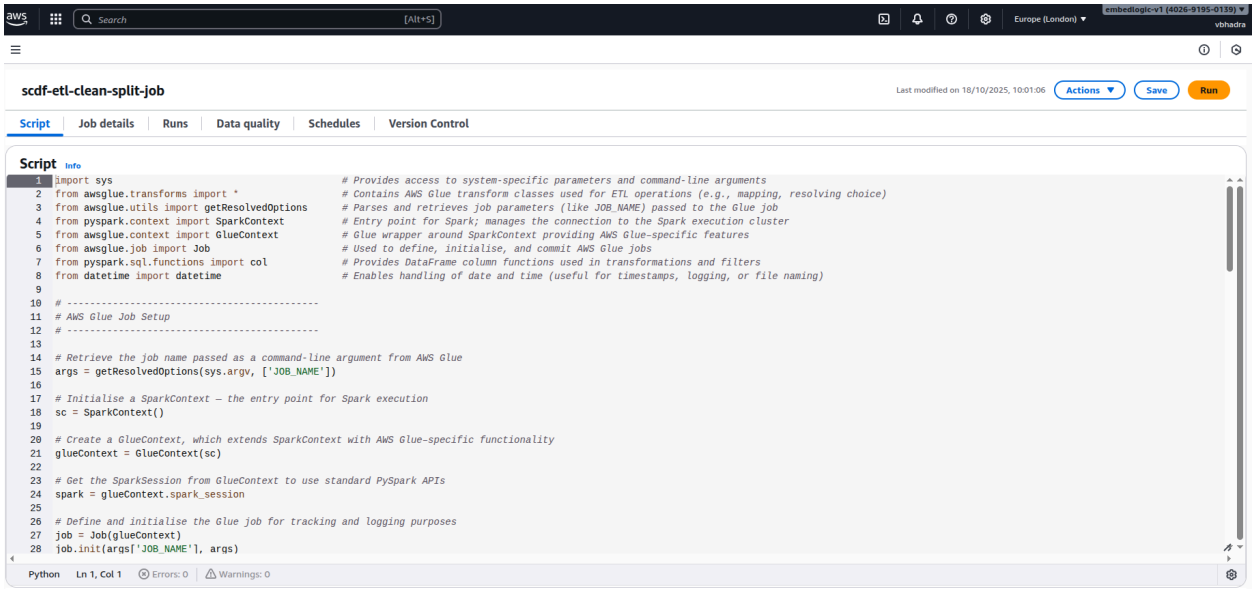
Running the ETL Job in AWS Glue

This section will walk you through the following steps:

1. Open Glue Studio and navigate to Jobs



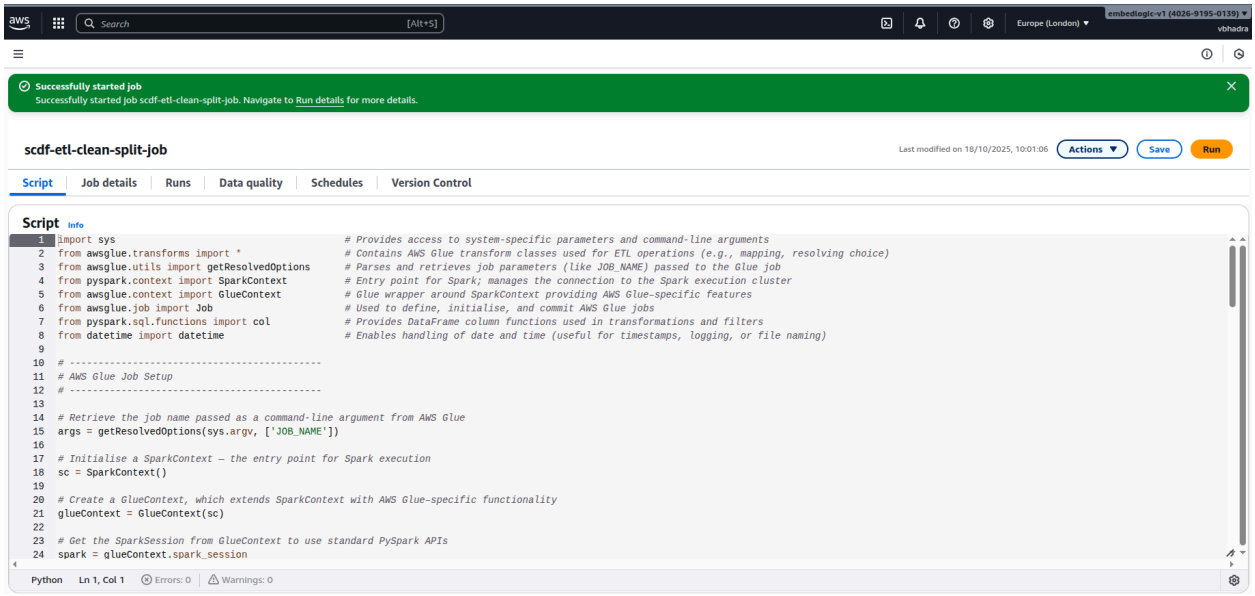
2. Selecting scdf-etl-clean-split-job



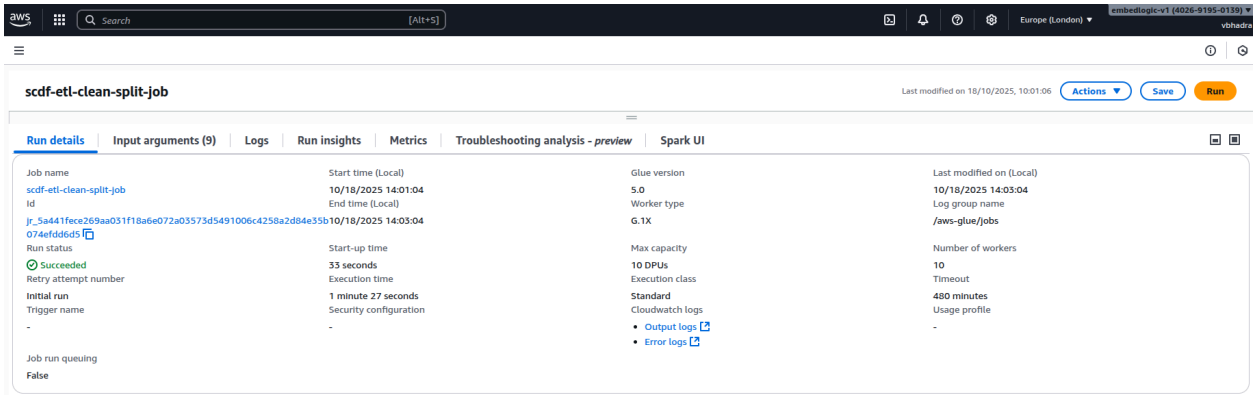
3. Clicking Run. You should see a message like the below on the screen:

Successfully started job

Successfully started job scdf-etl-clean-split-job. Navigate to Run details for more details.

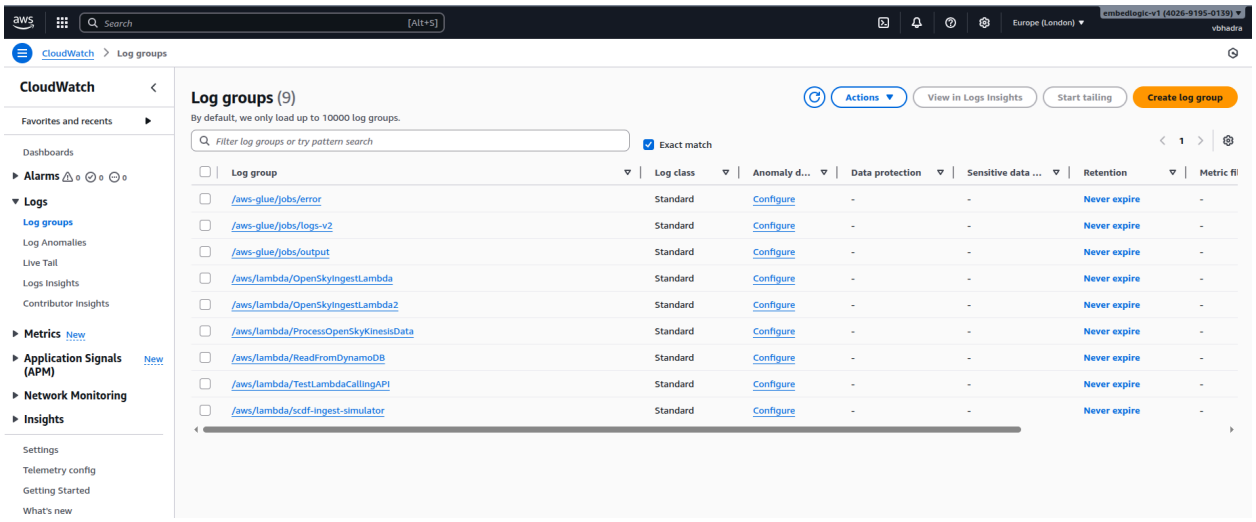


4. Monitoring job status (succeeded/failed). Go to Runs Tab for the status of the job started:



5. Viewing logs in CloudWatch. Go to the CloudWatch service in AWS and navigate to Log Groups and under log groups you should be able to see an entry created for the Glue job, named something like the below:

   /aws-glue/jobs/error



Check the logs and see if there is something interesting.

6. Verifying output in:
    - s3://scdf-project-data/processed/
    - s3://scdf-project-data/training/train.csv/
    - s3://scdf-project-data/training/test.csv/

Enhancing the Preprocessing Step

In the first version of our ETL script, the preprocessing logic was intentionally kept lightweight to get the pipeline up and running. The approach did not explicitly handle missing values or data types, simply loading the raw data as-is. This simplistic strategy presents several key shortcomings:

- **It can cause unnecessary data issues** — without explicit type casting, columns remain as strings, which can cause errors or unexpected behavior in downstream transformations or machine learning models.
- **It lacks observability** — without logging schema details, missing value counts, or statistical summaries, there's no visibility into the data's state before transformation, making debugging and validation difficult.
- **It risks silent data quality problems** — missing values were implicitly accepted without checks or imputation, which can degrade model performance or cause runtime failures.

To make preprocessing more robust, traceable, and machine-learning-ready, we enhanced the script in several key ways.

Column Type Conversion and Logging

We began by explicitly casting our columns to their correct data types. For example, the original dataset stores all values as strings. So we added:

```
df = (
    df.withColumn("store", col("store").cast("int"))
      .withColumn("item", col("item").cast("int"))
      .withColumn("sales", col("sales").cast("float"))
      .withColumn("date", col("date").cast("date"))
)
```

Then, to confirm that each field is now correctly typed, we added a simple loop to log column names and their types:

```
print("---- Column Data Types ----")
for name, dtype in df.dtypes:
    print(f"{name}: {dtype}")
```

This small addition provides immediate visibility into how the data is structured before transformations—a crucial step in any pipeline.

Missing Value Imputation

Rather than dropping all rows with nulls, we impute missing values in the sales column using the column's mean:

```
mean_sales =
df.select(mean("sales").alias("mean_sales")).collect()[0]["mean_sales"]
df_cleaned = df.fillna({"sales": mean_sales})
```

This preserves more of the original dataset while still addressing incomplete records. In business datasets—especially in retail—it's common to encounter small gaps, so filling rather than discarding aligns better with real-world use.

We also log the number of missing values per column before imputation:

```
missing_info = {colname: df.filter(col(colname).isNull()).count() for
```

```
colname in df.columns}
print("---- Missing Value Check ----")
print(missing_info)
```

Summary Statistics

To make the pipeline more transparent and exploratory, we added basic descriptive statistics using PySpark's built-in .describe() method:

```
df.describe(["sales", "store", "item"]).show()
```

This offers a quick view of mean, min, max, standard deviation, and counts—helping detect outliers, unusual scale differences, or missing distributions early on.

Sales Column Normalisation

Machine learning models often perform better when numeric features are scaled within a standard range, so we introduced **Min-Max normalization** on the sales column. This required assembling the column into a vector (a Spark ML requirement), applying MinMaxScaler, then cleaning up the result:

```
assembler = VectorAssembler(inputCols=["sales"], outputCol="sales_vector")
scaler = MinMaxScaler(inputCol="sales_vector", outputCol="sales_scaled")
pipeline = Pipeline(stages=[assembler, scaler])
scaler_model = pipeline.fit(df_cleaned)
df_scaled = scaler_model.transform(df_cleaned)

# Convert vector to scalar float
vector_to_float = udf(lambda vec: float(vec[0]), FloatType())
df_final = (
    df_scaled.withColumn("sales_scaled_value",
vector_to_float(col("sales_scaled")))
             .drop("sales_vector", "sales_scaled")
)
```

After this step, the sales_scaled_value column holds values between 0 and 1, ensuring this feature won't dominate model training due to scale.

Improved Traceability via CloudWatch Logs

Throughout these enhancements, we added print() statements after each transformation to log the DataFrame's internal state and track the success of key stages. These messages appear in CloudWatch Logs under your Glue job's output stream. This brings a DevOps perspective to the pipeline—making every transformation observable, debuggable, and reproducible in production.

Together, these changes transform our basic preprocessing script into a robust, cloud-native ETL pipeline step that's ready for real-world machine learning workflows.

1.4 Exploratory Data Analysis (EDA)

After cleaning and normalising our dataset, the next logical step in the pipeline is to perform Exploratory Data Analysis (EDA). This stage is essential to understand the structure, relationships, and variability within the data — which directly informs feature selection, model choice, and performance expectations.

EDA allows us to extract early insights, identify potential anomalies, and uncover correlations between variables before committing to model training. In this assignment, we used AWS Glue and PySpark to perform EDA in a distributed, scalable, and cloud-native way.

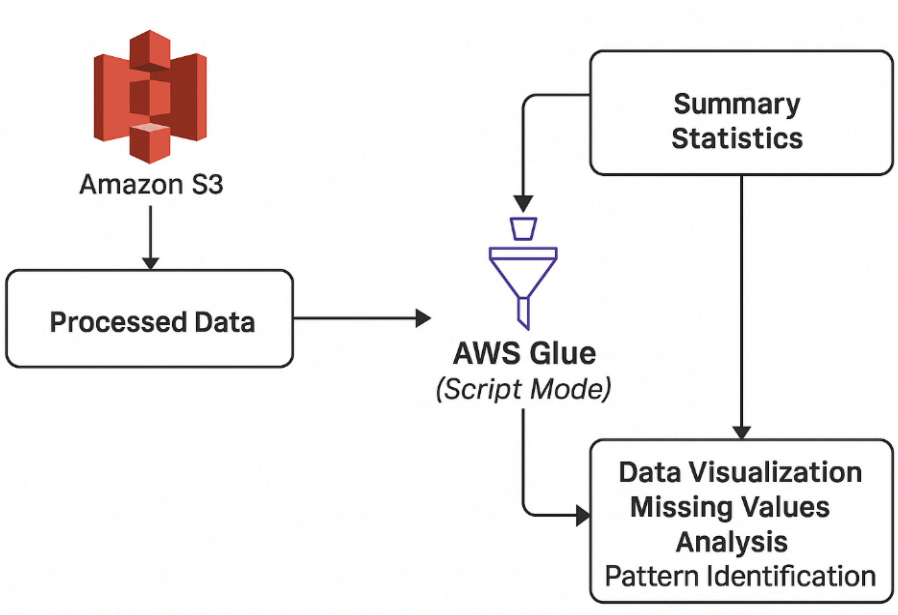


Figure 4: Conceptual model EDA

Establishing the EDA Environment

To maintain consistency across stages, the exploratory analysis will be conducted using **AWS Glue Studio Notebook mode**. This environment will provide the scalability of a Spark cluster with the convenience of a managed Jupyter-like interface, allowing queries and transformations to run directly on the processed dataset stored in **Amazon S3**.

A new **Glue job** will be created, referencing the same **IAM role** used in the preprocessing phase, ensuring access to both **S3** and **CloudWatch**. The preprocessed dataset — stored in **Parquet format** under the *processed* directory — will then be loaded as a **Spark DataFrame** for analysis.

```
from awsglue.context import GlueContext
from pyspark.context import SparkContext
from pyspark.sql.functions import col, year, month, dayofweek, avg, sum as
_sum, to_date

sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session

# Load processed data
df = spark.read.parquet("s3://scdf-project-data/processed/")
df.printSchema()
df.show(5)
```

This confirms that the preprocessing stage successfully outputs a schema-consistent, machine-learning-ready dataset.

Code Walkthrough

The goal of this code block is to initialise a distributed Spark environment within AWS Glue and load the preprocessed dataset for exploratory analysis. Each line contributes to setting up a scalable, cloud-native data exploration environment.

Initialising the Spark and Glue Contexts

```
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
```

Here, the execution environment is being instantiated:

- **SparkContext()** creates a new Spark session across the managed Glue cluster. This session controls all resource allocation and parallel task execution.
- **GlueContext(sc)** wraps the Spark context and enables Glue's additional features, including Data Catalog integration, logging, and job orchestration.
- **glueContext.spark_session** provides the standard SparkSession interface, allowing you to run familiar Spark DataFrame operations such as .read(), .select(), .groupBy(), and .describe().

This setup effectively transforms Glue into a fully operational Spark analytics engine, ready to process large-scale datasets directly from S3.

Loading the Processed Dataset

```
df = spark.read.parquet("s3://scdf-project-data/processed/")
```

This line retrieves the cleaned and normalised dataset produced in the previous ETL phase.

Key points:

- **.read.parquet()** instructs Spark to load Parquet files — a columnar storage format optimised for analytical workloads and query performance.
- The data resides in the processed/ directory of the same S3 bucket used previously, ensuring seamless pipeline continuity.
- Because the preprocessing script explicitly casted column types (store, item, sales, and date), Spark automatically recognises their correct data types at this stage.

This operation verifies data persistence and schema consistency between pipeline stages.

Verifying Schema and Sample Records

```
df.printSchema()
df.show(5)
```

These two commands are critical for validation:

- **printSchema()** displays the structure of the DataFrame, listing each column name along with its inferred type (e.g. store: int, item: int, sales: float, date: date). This ensures that all type casting performed during preprocessing was successful.

- show(5) prints the first five rows of data, allowing a quick visual inspection to confirm the dataset has been correctly loaded and that no unexpected transformations occurred during the handoff from ETL to EDA.

Together, these commands act as an integrity checkpoint — confirming that your ETL output is schema-consistent, readable, and ready for machine learning–oriented exploration.

Interpretation and Outcome

This initial setup completes the environment verification step of the EDA stage. By successfully reading from the processed Parquet dataset and inspecting its schema, we validate that:

- The preprocessing stage correctly produced a machine learning ready dataset.
- The Glue cluster can access the required S3 resources using the same IAM role.
- The Spark environment is active and configured for further analytical operations, such as computing summary statistics, time-series aggregations, and correlation analysis.

Understanding Dataset Structure

Once the processed dataset was successfully loaded into the AWS Glue environment, the first analytical step in Exploratory Data Analysis (EDA) was to examine its structure and statistical characteristics. This step provides a foundational understanding of how the dataset is organised — including data types, column relationships, and basic numerical summaries — ensuring that it aligns with the expectations defined during preprocessing.

Inspecting the Schema and Data Types

To confirm that all preprocessing transformations were correctly applied, the following commands were executed:

```
df.printSchema()
df.show(5)
```

The schema output retrieved from the CloudWatch logs verified that each column was properly typed and ready for analysis:

```
root
 |-- date: date (nullable = true)
 |-- store: integer (nullable = true)
 |-- item: integer (nullable = true)
 |-- sales: float (nullable = true)
 |-- sales_scaled_value: float (nullable = true)
```

This structure confirms that the preprocessing stage successfully applied the intended type casting and scaling transformations:

- date — identifies the transaction date in standard date format.
- store and item — integer identifiers for the retail outlet and product respectively.
- sales — the original daily sales value (floating-point).
- sales_scaled_value — the normalised version of the sales figure (scaled between 0 and 1 using Min-Max scaling).

The inclusion of both sales and sales_scaled_value columns ensures that downstream analyses can use either the raw or scaled metric, depending on the modelling or visualisation requirements.

Sample Records

A preview of the dataset was generated using:

```
df.show(5)
```

The first five records, retrieved directly from the CloudWatch logs, confirm that the dataset was successfully read from the processed Parquet files stored in Amazon S3:

```
+----------+-----+----+-----+------------------+
|      date|store|item|sales|sales_scaled_value|
+----------+-----+----+-----+------------------+
|2013-01-01|    1|   1| 13.0|       0.056277055|
|2013-01-02|    1|   1| 11.0|        0.04761905|
|2013-01-03|    1|   1| 14.0|       0.060606062|
|2013-01-04|    1|   1| 13.0|       0.056277055|
|2013-01-05|    1|   1| 10.0|       0.043290045|
+----------+-----+----+-----+------------------+
only showing top 5 rows
```

This verified that:

- All columns loaded correctly with the intended data types.
- The dataset contained valid numeric and date values without null or malformed entries.

- The preprocessed data was successfully preserved in Parquet format and is now machine learning ready.

Exploratory Data Insights

With the dataset successfully validated and loaded, the next step in the Exploratory Data Analysis (EDA) process was to derive statistical summaries and aggregated views that describe sales behaviour across time, stores, and items. Using AWS Glue and PySpark, this analysis was executed in a distributed, scalable manner directly on the processed data stored in Amazon S3.

Descriptive Statistics for Sales

The first analytical step involved computing overall descriptive statistics for the **sales** column to understand its central tendency and spread:

```
df.describe(["sales"]).show()
```

The output retrieved from CloudWatch was as follows:

```
+-------+------------------+
|summary|             sales|
+-------+------------------+
|  count|            913000|
|   mean|52.250286966046005|
| stddev|28.801143603517264|
|    min|               0.0|
|    max|             231.0|
+-------+------------------+
```

This summary provides a quick overview of the dataset's numeric distribution:

- The dataset contains 913,000 records, confirming completeness and high data volume.
- The mean sales value is approximately 52.25, with a standard deviation of about 28.80, indicating moderate variability in store-item performance.
- The minimum and maximum sales values (0.0 to 231.0) reflect a realistic range of retail transactions across different store-item combinations.

This descriptive layer establishes the baseline for detecting anomalies, assessing variability, and informing scaling decisions in future modelling phases.

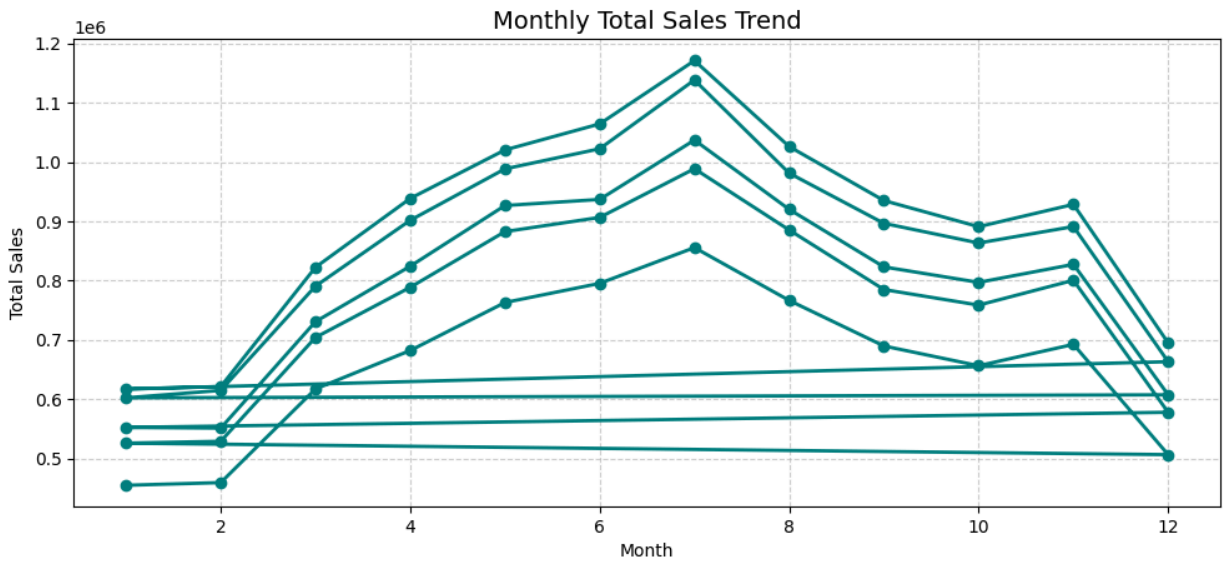
Temporal Sales Trends

To examine monthly sales patterns over time, the year() and month() functions were applied to the date column, followed by a group-wise aggregation of total monthly sales:

```
from pyspark.sql.functions import year, month, sum as _sum
df = df.withColumn("year", year(col("date"))).withColumn("month",
month(col("date")))
monthly_sales = df.groupBy("year",
"month").agg(_sum("sales").alias("total_sales")).orderBy("year", "month")
monthly_sales.show(10)
```

The following output segment illustrates total monthly sales for 2013:

```
+----+-----+-----------+
|year|month|total_sales|
+----+-----+-----------+
|2013|    1|   454904.0|
|2013|    2|   459417.0|
|2013|    3|   617382.0|
|2013|    4|   682274.0|
|2013|    5|   763242.0|
|2013|    6|   795597.0|
|2013|    7|   855922.0|
|2013|    8|   766761.0|
|2013|    9|   689907.0|
|2013|   10|   656587.0|
+----+-----+-----------+
only showing top 10 rows
```



From this, we can infer:

- Steady growth in total sales from January through July.
- A seasonal plateau during mid-year, suggesting cyclical consumer behaviour.
- The presence of temporal variability, which supports the eventual inclusion of time-based features (month, quarter, season) during feature engineering.

This monthly aggregation validates that the pipeline can efficiently summarise time-series data at scale.

Store-Level Performance Analysis

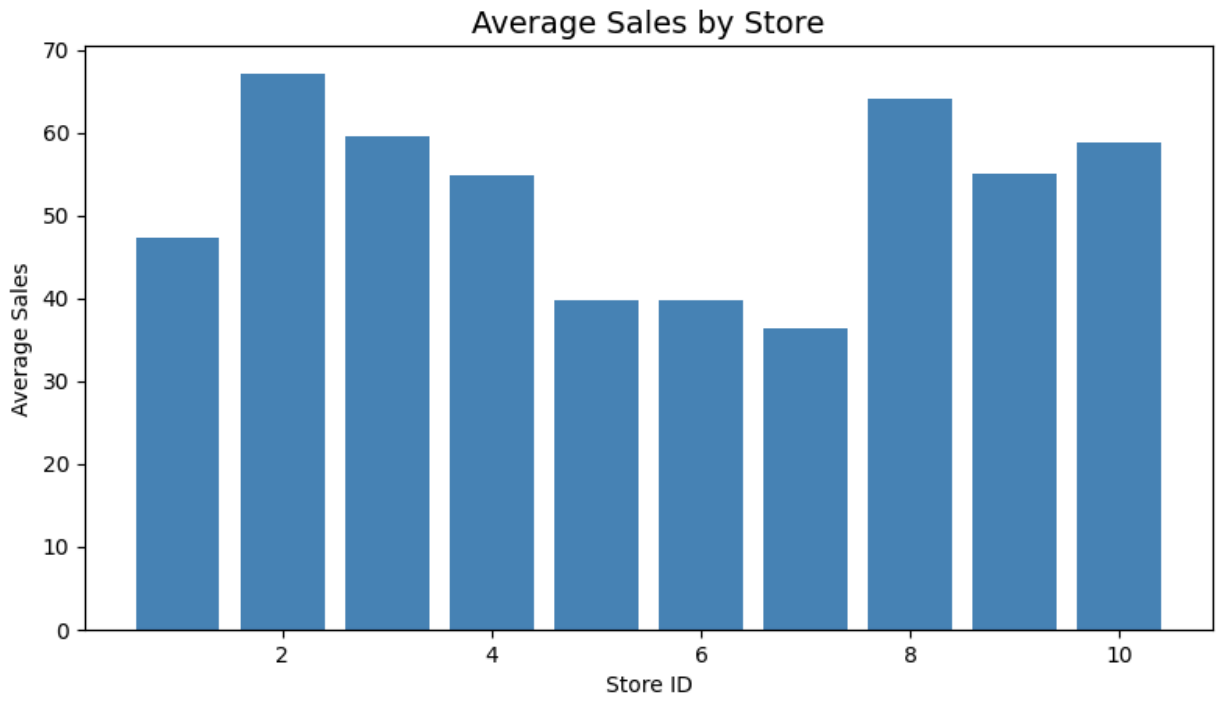
Next, store-level performance was assessed by calculating the average sales per store:

```
store_avg =
df.groupBy("store").agg(avg("sales").alias("avg_sales")).orderBy(col("avg_s
ales").desc())
store_avg.show(10)
```

The output revealed clear differences in performance across retail locations:

```
+-----+------------------+
|store|         avg_sales|
+-----+------------------+
```

```
|    2| 67.03316538882804|
|    8| 64.14204819277109|
|    3|59.530602409638554|
|   10| 58.70928806133625|
|    9|55.049025191675796|
|    4| 54.90294633077766|
|    1|47.268378970427165|
|    5| 39.77016429353779|
|    6|39.733515881708655|
|    7|36.363734939759034|
+-----+------------------+
```



This ranking highlights that **Store 2** consistently achieved the highest average sales, while **Stores 6 and 7** recorded lower averages. Such store-level variation can be used to build location-specific demand forecasting models or to identify underperforming outlets requiring operational adjustments.

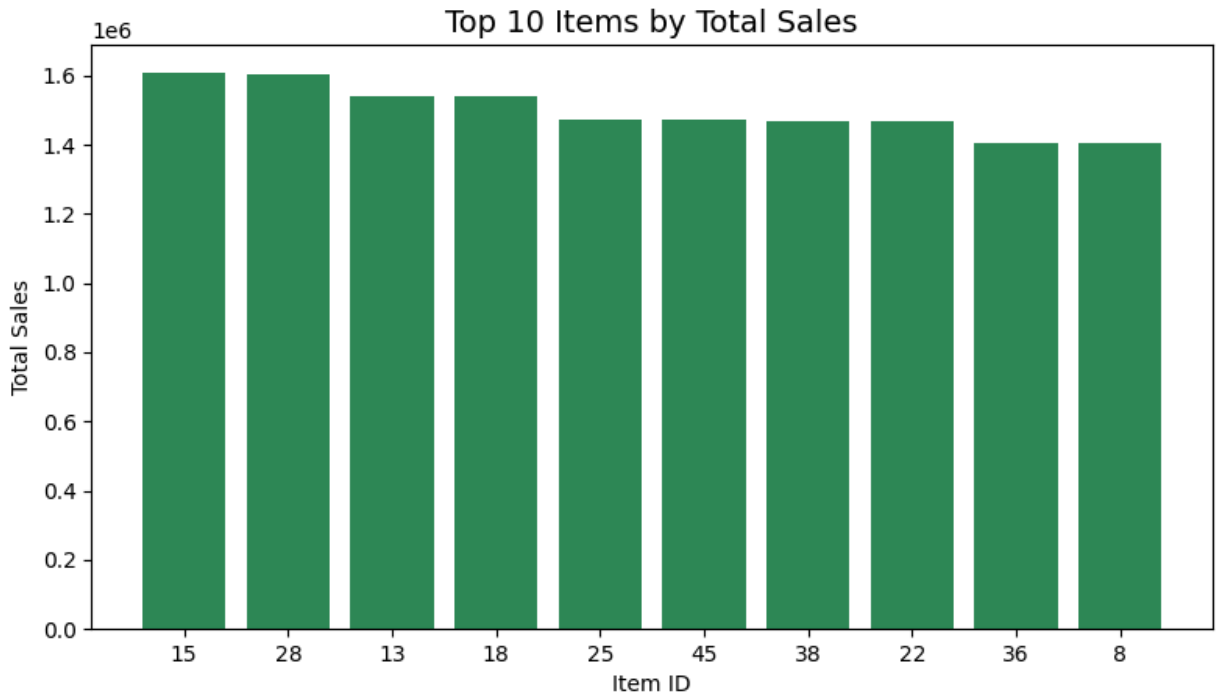
Item-Level Performance Analysis

Similarly, item-level aggregation was performed to identify the top-selling products across all stores:

```
item_sales =
df.groupBy("item").agg(_sum("sales").alias("total_sales")).orderBy(col("tot
al_sales").desc())
item_sales.show(10)
```

The output was as follows:

```
+----+-----------+
|item|total_sales|
+----+-----------+
|  15|  1607442.0|
|  28|  1604713.0|
|  13|  1539621.0|
|  18|  1538876.0|
|  25|  1473334.0|
|  45|  1471467.0|
|  38|  1470330.0|
|  22|  1469971.0|
|  36|  1406548.0|
|   8|  1405108.0|
+----+-----------+
only showing top 10 rows
```



The results show that items **15**, **28**, and **13** were the highest contributors to total sales during the analysed period. This insight will be particularly valuable during feature engineering, where item-level popularity or historical demand strength can be used to enrich predictive features for machine learning.

Missing Value Check

A missing value check was also performed to confirm the completeness of the dataset:

```
missing_info = {c: df.filter(col(c).isNull()).count() for c in df.columns}
print(missing_info)
```

The output retrieved from CloudWatch logs was:

```
Missing Values Summary:
{'date': 0, 'store': 0, 'item': 0, 'sales': 0, 'sales_scaled_value': 0,
'year': 0, 'month': 0, 'dayofweek': 0}
```

This confirms that all eight columns are **100 % complete**, with no missing or null records. The earlier data-cleaning and imputation steps in the preprocessing stage successfully ensured dataset integrity and consistency. Having a null-free dataset is essential for distributed computation within Spark, as it prevents skewed aggregations and invalid type operations during model training.

Correlation Analysis and Output Persistence

After verifying dataset completeness, a quantitative correlation analysis was performed to understand how key variables interact and influence sales outcomes. This step helps determine which attributes carry predictive potential and which may be redundant or weakly associated with the target variable.

Numerical Correlation Analysis

To compute Pearson correlation coefficients between major numeric columns, the following PySpark commands were executed:

```
# Correlation between key numerical variables
corr_store_item = df.stat.corr("store", "item")
corr_store_sales = df.stat.corr("store", "sales")
```

```
corr_item_sales = df.stat.corr("item", "sales")

print("Correlation between store and item:", corr_store_item)
print("Correlation between store and sales:", corr_store_sales)
print("Correlation between item and sales:", corr_item_sales)
```

The CloudWatch log output was as follows:

```
Correlation between store and item: 7.063209925969646e-16
Correlation between store and sales: -0.008170361306182861
Correlation between item and sales: -0.05599807493660445
```

These values provide meaningful insights:

- Store vs Item (≈ 0) — negligible correlation, confirming that store identifiers and item identifiers are independent categorical features.
- Store vs Sales (≈ -0.008) — near-zero correlation, implying that sales variation is not strongly tied to store ID alone. It likely depends more on other temporal or product-specific factors.
- Item vs Sales (≈ -0.056) — weak negative correlation, suggesting minor variation in item-level demand but no strong linear dependency.

Such results reinforce the idea that sales behaviour is multi-factorial — influenced by time, item, and location combinations rather than any single attribute in isolation. This finding directly motivates the feature-engineering phase, where interaction features and lagged sales trends will be incorporated to capture these complex relationships.

Inference

By completing correlation computation and persisting the analytical outputs to Amazon S3, the EDA phase achieves full reproducibility and observability.
The dataset is now:

- Fully validated with zero missing values.
- Statistically summarised across temporal, store, and item dimensions.
- Correlated and contextualised, providing actionable insight into feature relationships.

- Persisted in cloud storage, making it readily accessible for model training.

The next step in this pipeline will be Feature Engineering and Model Training Preparation - where temporal, categorical, and interaction-based features will be generated to feed a predictive demand forecasting model.

Feature Engineering and Model Training Preparation

After completing the Exploratory Data Analysis (EDA) stage, the next logical step in our pipeline is Feature Engineering — a critical phase that bridges data exploration and model training.

In this stage, we transform the cleaned and analysed dataset into a machine-learning–ready form by creating new, meaningful features that capture seasonality, store-item interactions, and temporal patterns.

Objective

The objective of this phase is to:

- Generate additional temporal and statistical features that can help the model recognise trends and seasonality.
- Encode categorical features (store, item) and maintain numeric consistency for ML algorithms.
- Ensure the enriched dataset is complete, schema-consistent, and stored in a structured S3 location ready for model ingestion.

Establishing the Feature Engineering Environment

To maintain continuity and reproducibility across all pipeline stages, we perform this phase using AWS Glue in script mode — the same configuration used for preprocessing and exploratory data analysis. Running all jobs within a consistent Glue environment ensures identical cluster setup, IAM role usage, and S3 access permissions throughout the workflow.

A new Glue job named scdf-feature-engineering-job is created under the existing IAM role. To allow the job to persist the engineered feature outputs, the IAM policy is extended with the following additional resource path:
`"arn:aws:s3:::scdf-project-data/features/*"`

This policy update ensures that the Glue job can securely store transformed feature datasets in the features/ directory, enabling a seamless transition into the next phase — model training and evaluation.

Code Walkthrough and Output Verification

The feature engineering stage extends the preprocessing and EDA pipeline, enriching the dataset with temporal and lag-based predictors critical for time-series forecasting. The implementation, contained in glue_script_feature_engineering.py, follows a well-defined five-step flow consistent with earlier Glue jobs. The script is available via the public github repository created for this project.

AWS Glue Job Setup

The script begins with the standard setup:

```
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

This initialises the Spark environment and links it to AWS Glue's job lifecycle management.

Reading the Processed Dataset

```
input_path = "s3://scdf-project-data/processed/"
df = spark.read.parquet(input_path)
print("Processed dataset loaded successfully.")
```

The CloudWatch logs confirm successful data load:

2025-10-20T06:26:09.718Z Processed dataset loaded successfully.

This validates seamless continuity between the preprocessing and feature engineering stages.

Creating Temporal Features

Using PySpark's date functions:

```
df = df.withColumn("year", year(col("date"))) \
       .withColumn("month", month(col("date"))) \
       .withColumn("day_of_week", dayofweek(col("date")))
print("Temporal features (year, month, day_of_week) created.")
```

Corresponding CloudWatch entry:
2025-10-20T06:26:09.853Z Temporal features (year, month, day_of_week) created.

These features introduce seasonality awareness into the dataset, which later models will exploit for demand forecasting.

Lag and Rolling Average Features

The script constructs short-term trend indicators using Spark's window functions:

```
window_spec = Window.partitionBy("store", "item").orderBy("date")
df = df.withColumn("lag_1", lag("sales", 1).over(window_spec))
df = df.withColumn("lag_7", lag("sales", 7).over(window_spec))
df = df.withColumn("rolling_avg_7",
avg("sales").over(window_spec.rowsBetween(-6, 0)))
print("Lag and rolling average features created.")
```

CloudWatch Log confirmation:

2025-10-20T06:26:10.077Z Lag and rolling average features created.

- lag_1: previous day's sales
- lag_7: previous week's sales
- rolling_avg_7: seven-day moving average of sales

Together, these enhance the dataset's ability to represent momentum and temporal dependencies.

Handling Missing Values and Persisting Outputs

Missing lag values are imputed, and the final data is written back:

```
df = df.na.fill(0, subset=["lag_1", "lag_7", "rolling_avg_7"])
```

```
print("Missing values in lag features imputed with zeros.")
output_path = "s3://scdf-project-data/features/"
df.write.mode("overwrite").parquet(output_path)
print("Feature-engineered dataset written to:", output_path)
```

We have confirmed this from CloudWatch logs:

```
2025-10-20T06:26:10.151Z  Missing values in lag features imputed with
zeros.
2025-10-20T06:26:25.522Z  Feature-engineered dataset written to:
s3://scdf-project-data/features/
2025-10-20T06:26:25.528Z  Feature engineering job completed successfully
at: 2025-10-20 06:26:25.527828
```

These entries collectively verify full job success, schema stability, and data persistence.

1.5 DataOps Automation and Scheduling

Objective

After successfully implementing the ingestion, preprocessing, feature engineering, and exploratory data analysis (EDA) stages, the next objective was to **automate their execution** and establish **continuous monitoring**. Automation ensures that data transformation and analytical jobs are executed in a consistent and repeatable manner, while monitoring provides visibility into system health and performance through real-time metrics. Together, these two components complete the **DataOps layer** of the solution — enabling **reliability**, **transparency**, and **operational consistency** across the data pipeline.

AWS Glue Workflow

The first stage of automation involves creating an **AWS Glue Workflow** to orchestrate the end-to-end data pipeline. This workflow will execute all the Glue jobs — covering **ETL (cleaning and splitting)**, **EDA**, and **feature engineering** — in a well-defined **sequential order**.

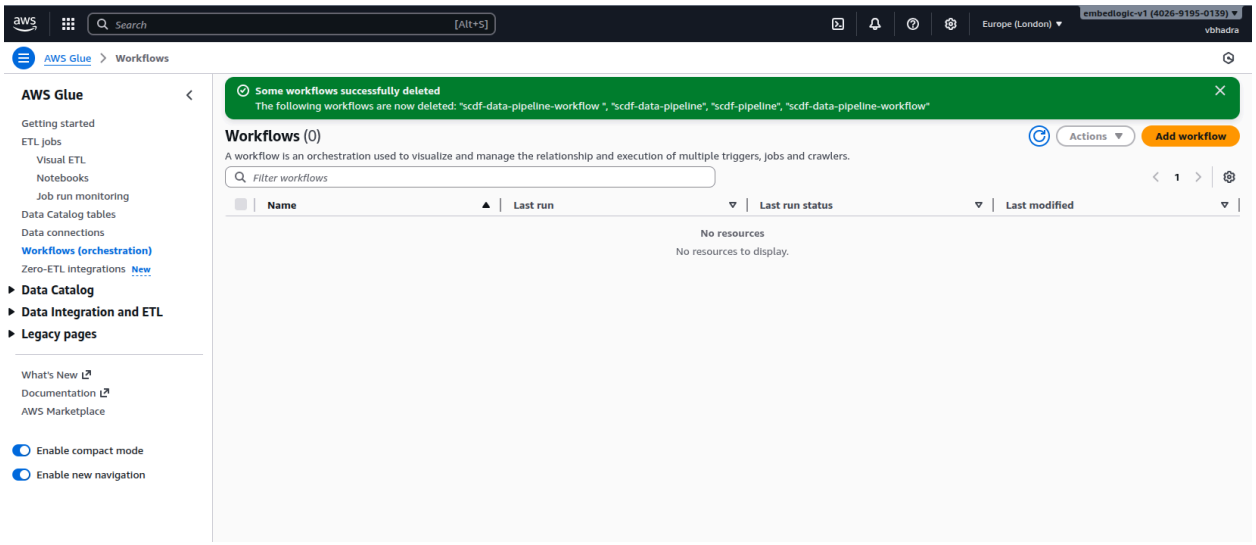
Once created, the workflow will be **manually triggered** to verify that each job runs successfully and that all outputs are written to their respective S3 destinations. Job execution status and logs will be validated through **Amazon CloudWatch** to ensure proper sequencing and error-free completion.

In the second stage, an **Amazon EventBridge rule** will be configured to **automate the workflow execution** at regular intervals (e.g., daily or hourly), ensuring continuous and unattended data processing.

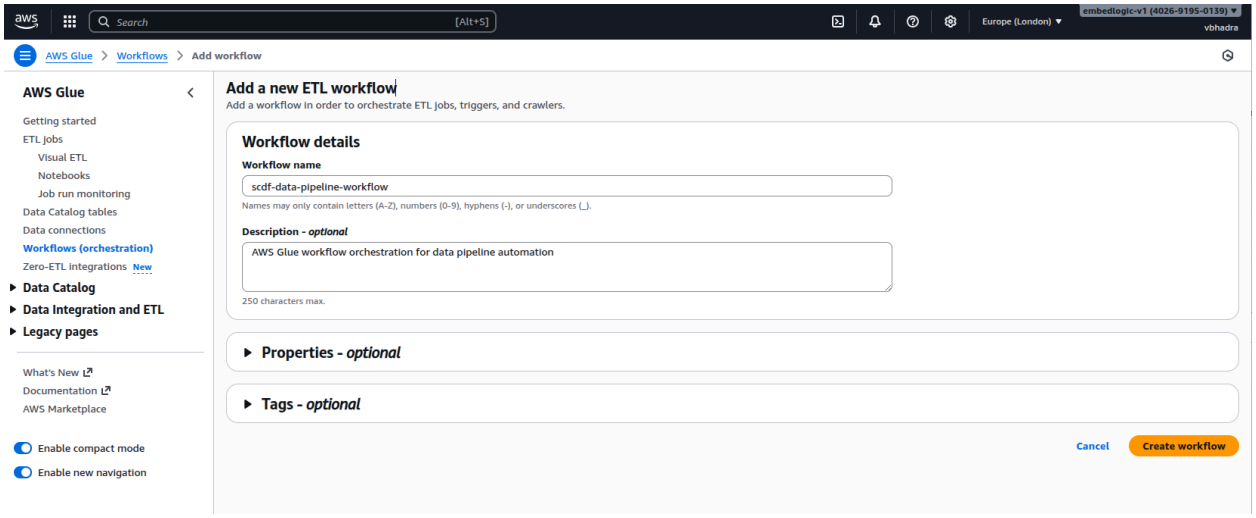
At this stage, the **machine learning Glue job** is intentionally excluded. It will be integrated later during the **end-to-end ML pipeline automation phase**, where model training and evaluation will be combined with the upstream data preparation workflow to form a complete, production-ready pipeline.

Creating Glue Workflow

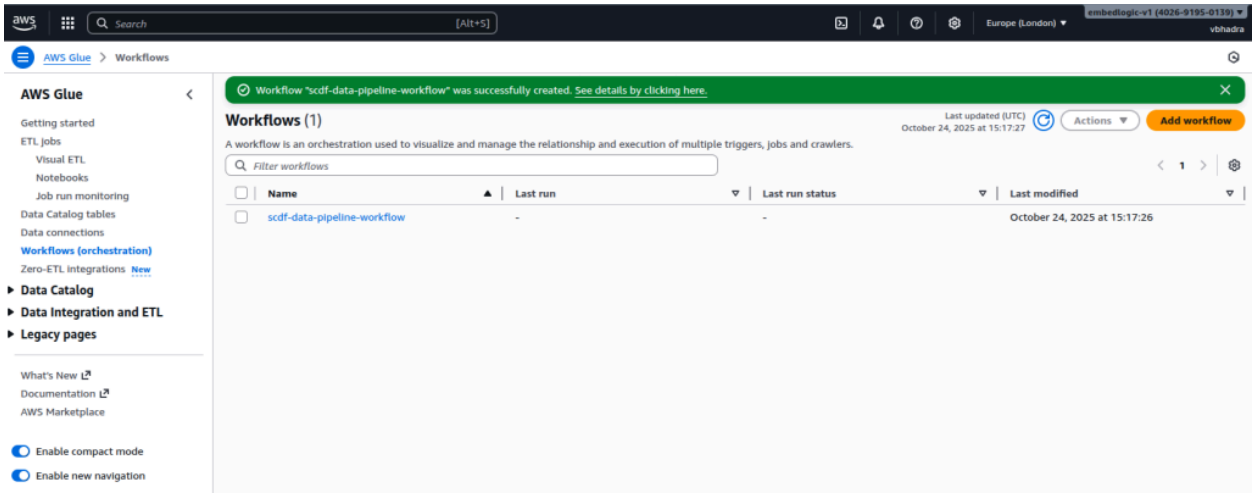
To create an AWS Workflow go to the AWS Glue Service and click on Workflow (orchestration).



Click on the Add Workflow button located at the top right corner of the page. Fill in the following details for the workflow:



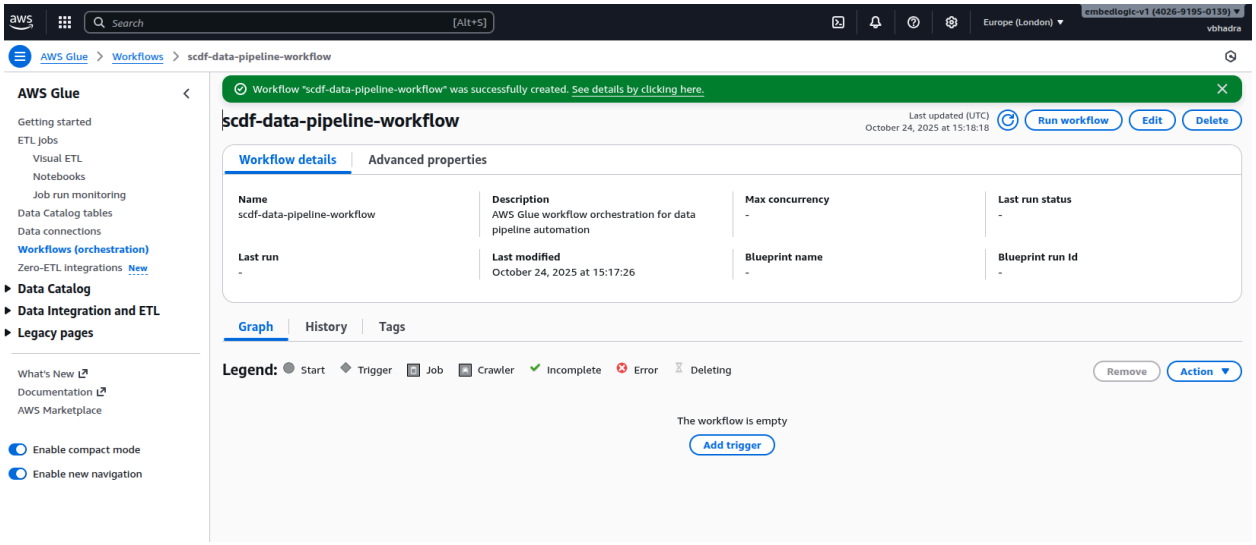
Click on the Create Workflow button at the bottom right corner. A new workflow named scdf-data-pipeline-workflow will be created for you:



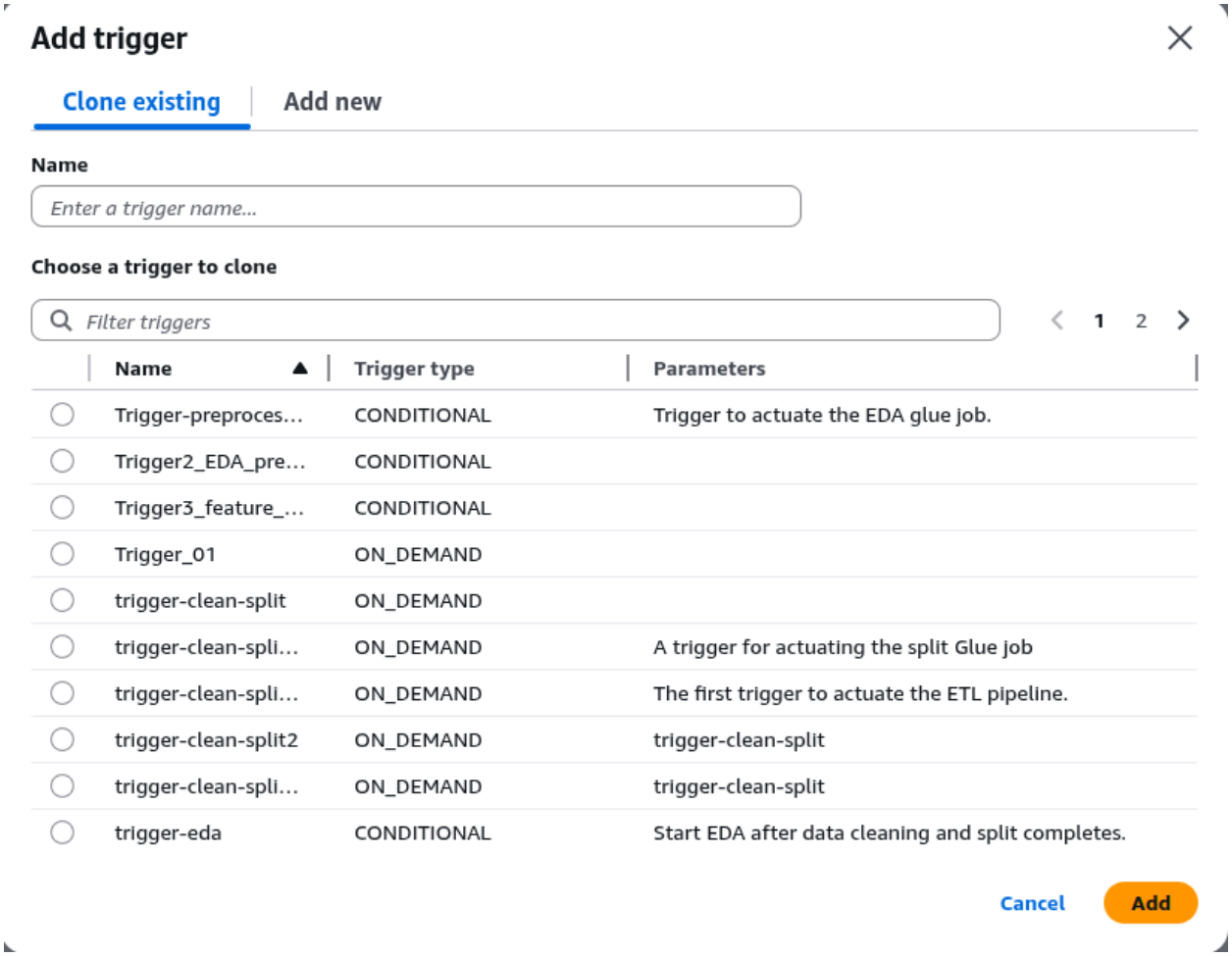
Once created, the workflow appears in our orchestration list. However, a workflow is only as effective as its triggers — and that's what we configured next.

Setting Up Triggers for Sequential Execution

Click on the workflow to go to its detail page:

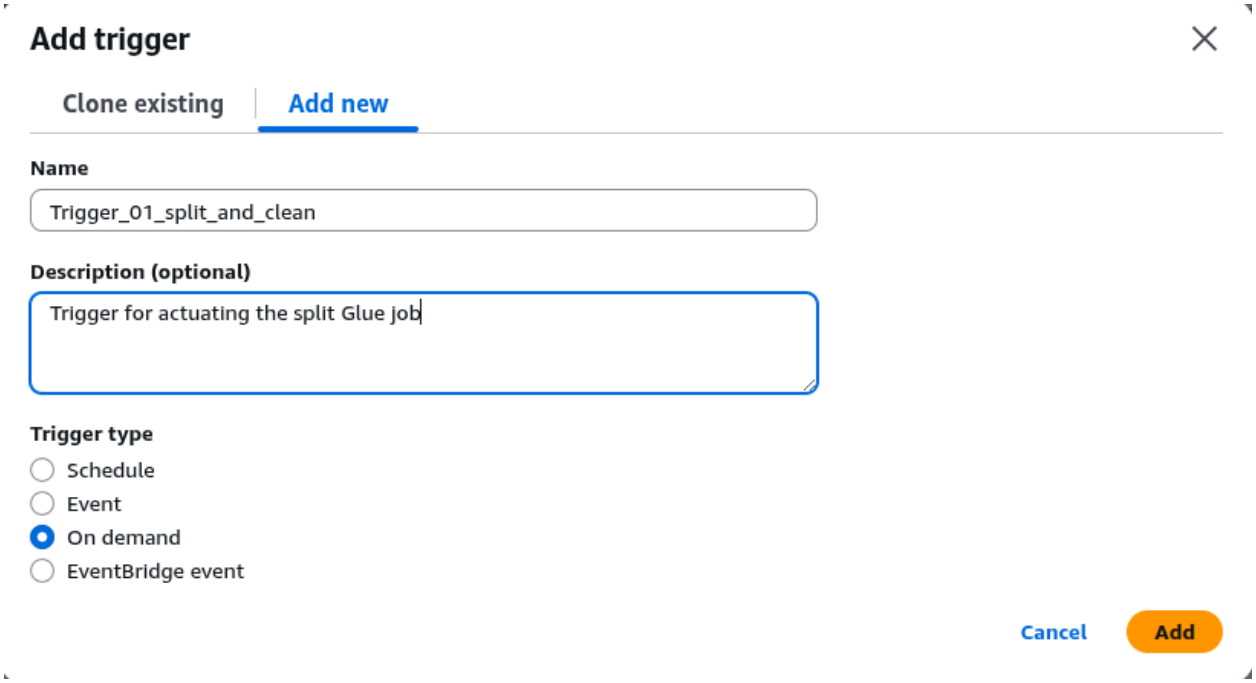


Once created, the workflow appears in our orchestration list. However, a workflow is only as effective as its triggers — and that's what we set up next.age:

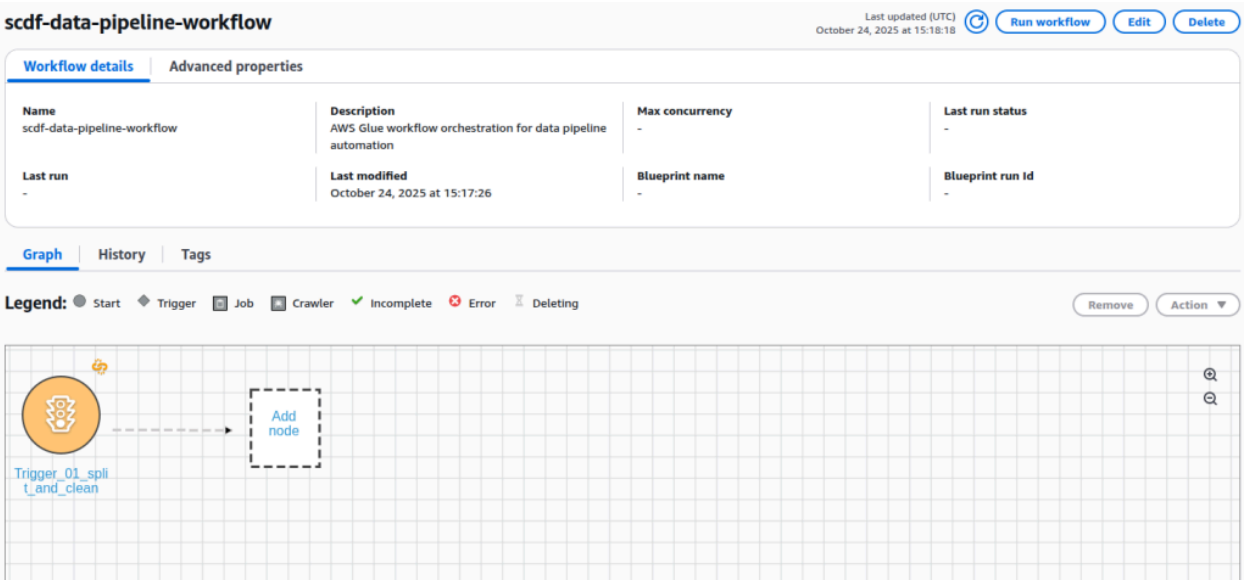


A pop-up screen will appear, giving you the option to choose an existing trigger or create a new one. For our purposes, we'll create a new trigger. Click on Add New and fill in the following details for the trigger:

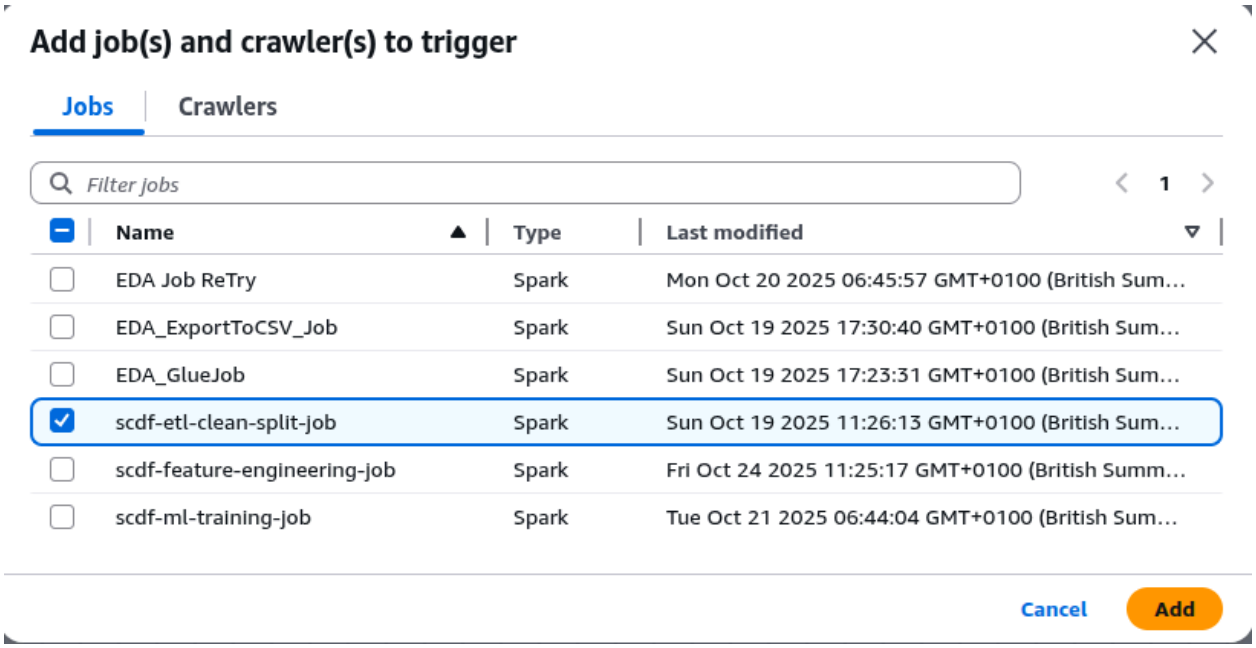
```
Trigger Name: Trigger_01_split_and_clean
Description: Trigger for actuating the split Glue job
Trigger Type: On demand
```



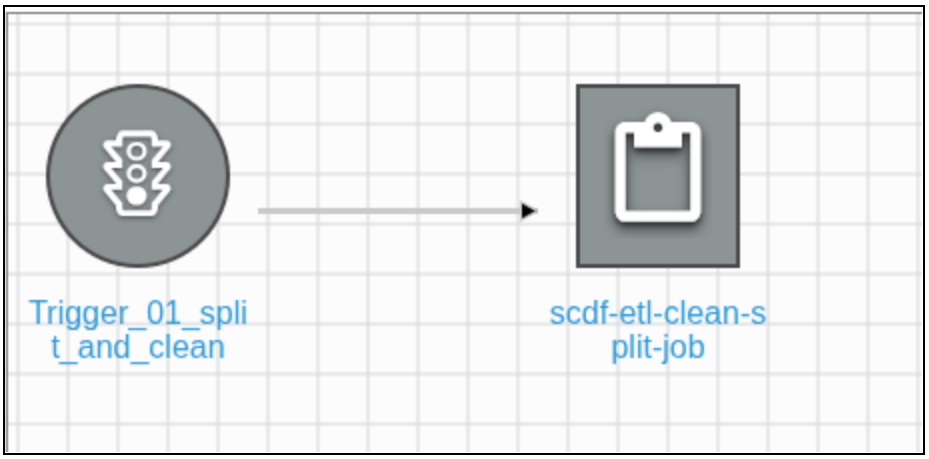
Click on the Add button at the bottom, and you should see that the trigger has been created.



Next, we need to attach a Glue job to the trigger. This first trigger will initiate the first Glue job in our pipeline, scdf-etl-clean-split-job. Click on Add Node and select scdf-etl-clean-split-job from the list.

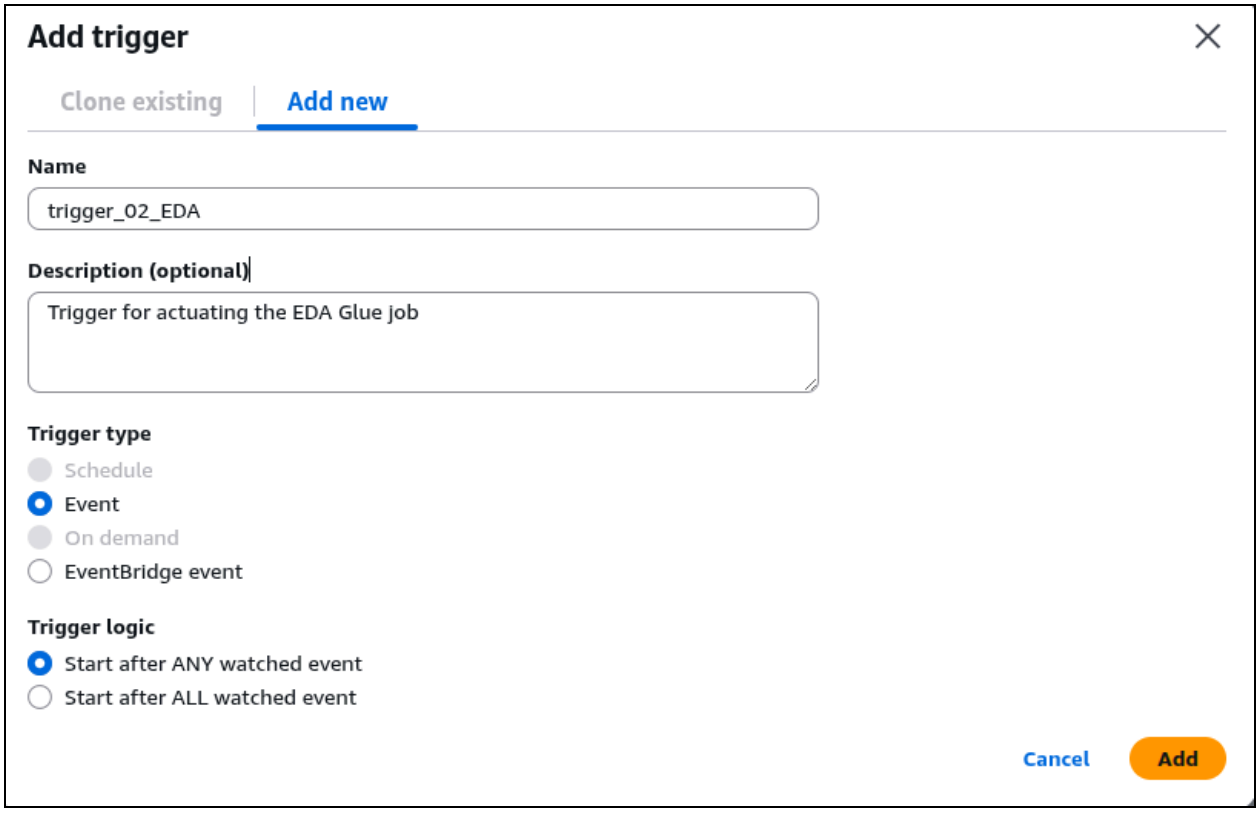


Click on Add to add the job.



Click on scdf-etl-clean-split-job, and it will expand, prompting you to attach the next trigger in the pipeline.

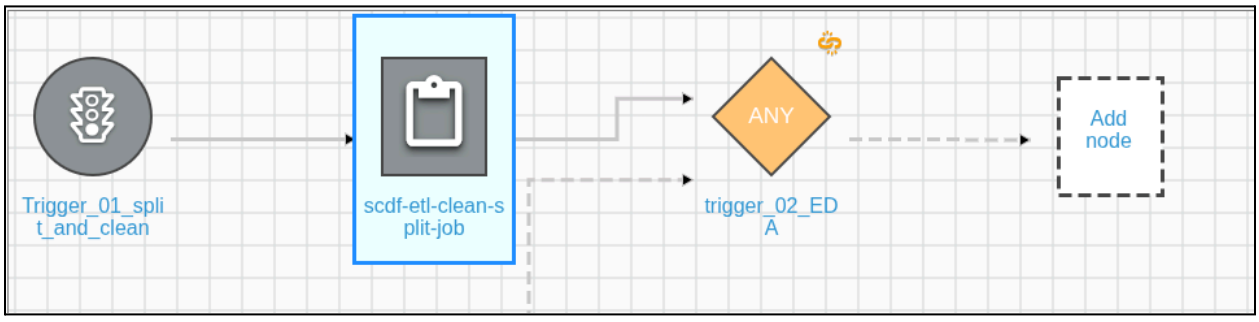
Next, we need to create another trigger for the EDA Glue job. Click on Add Trigger, as shown in the diagram.



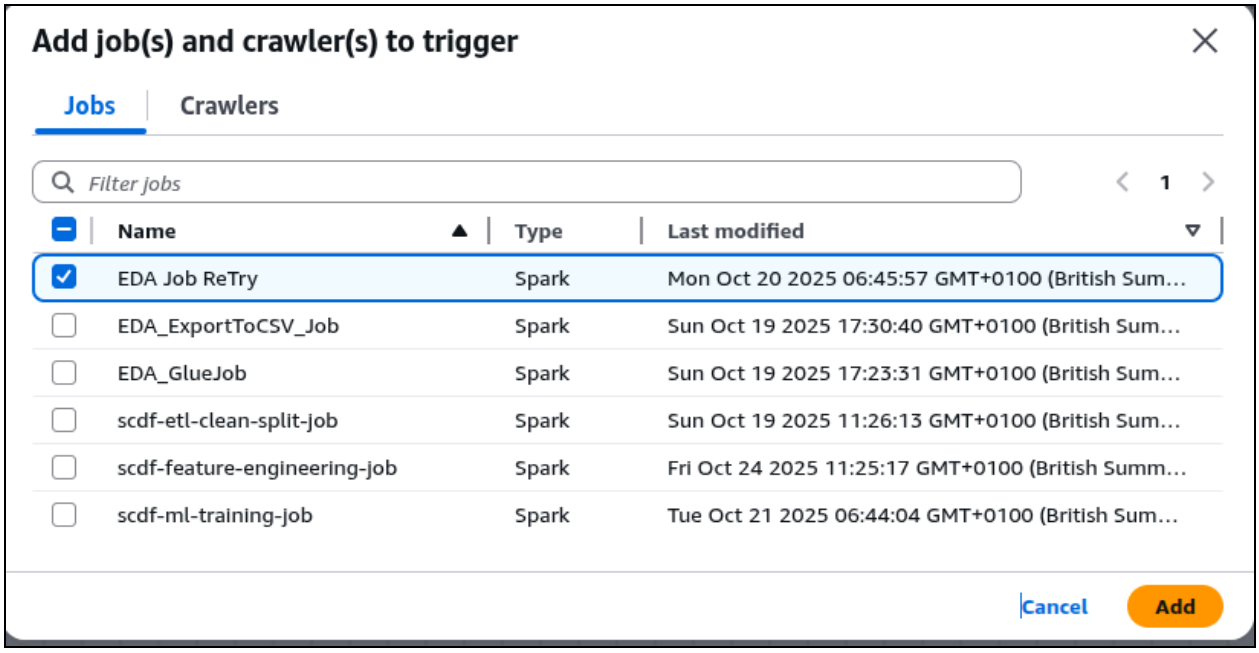
Fill in the following details about the trigger:

```
Trigger Name: trigger_02_EDA
Description: Trigger for actuating the EDA Glue job
Trigger Type: Event
Trigger Logic: Start after ANY watched event
```

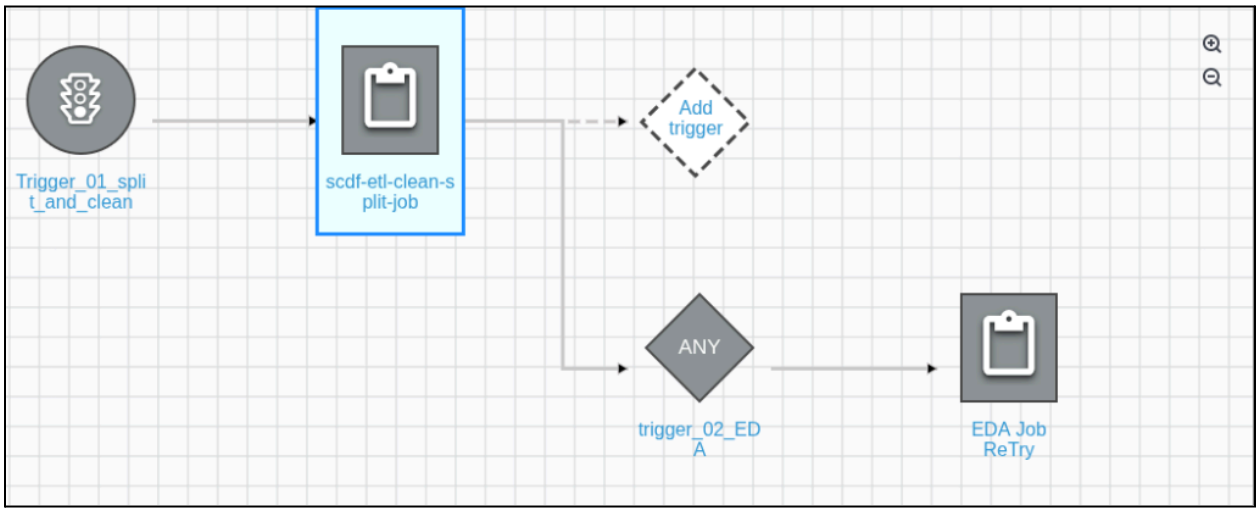
Click on Add to add the trigger.



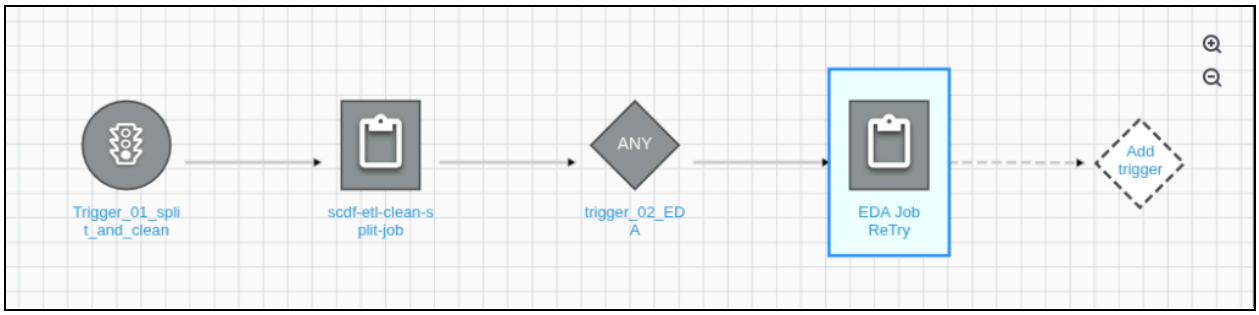
Select EDA Job Retry (the name of the EDA Glue job):



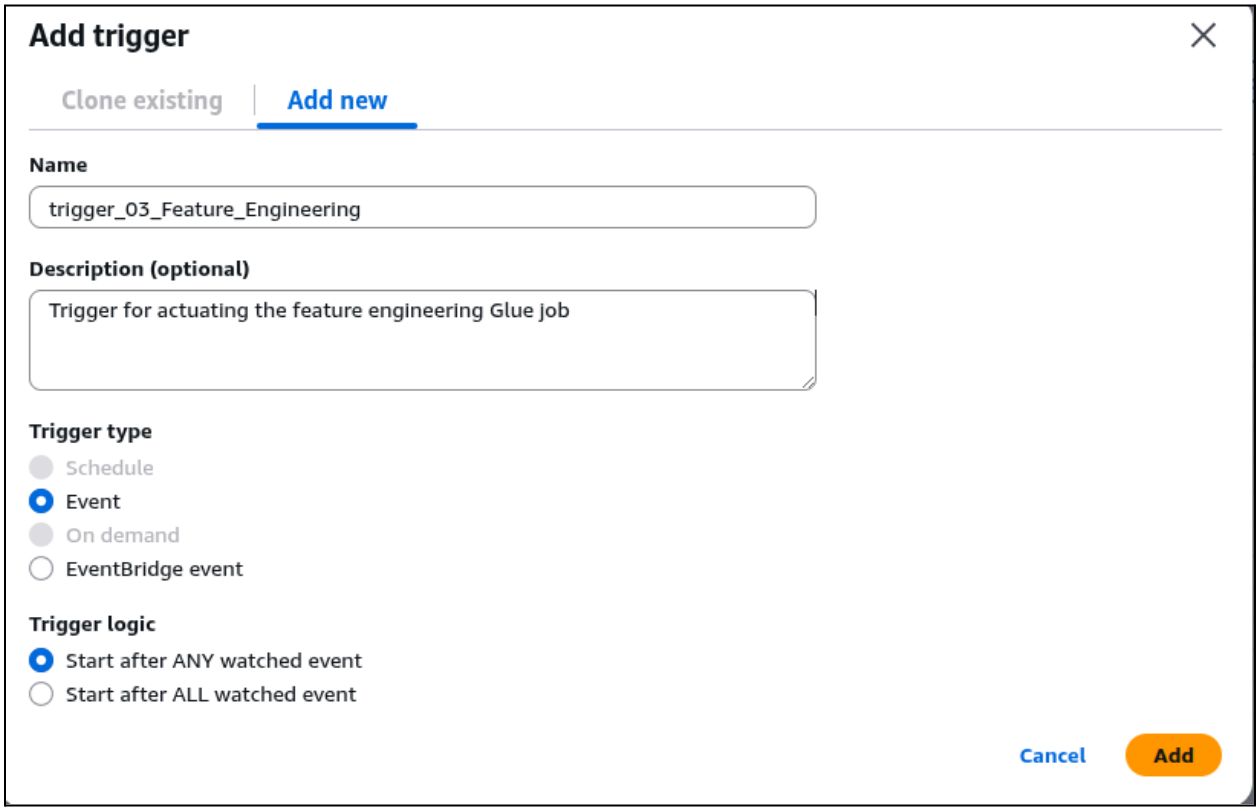
Click on Add to attach the job to the trigger:



We have one more trigger to create for the feature engineering Glue job. Click on the EDA Job Retry button, and it will expand, prompting you to add the next trigger:



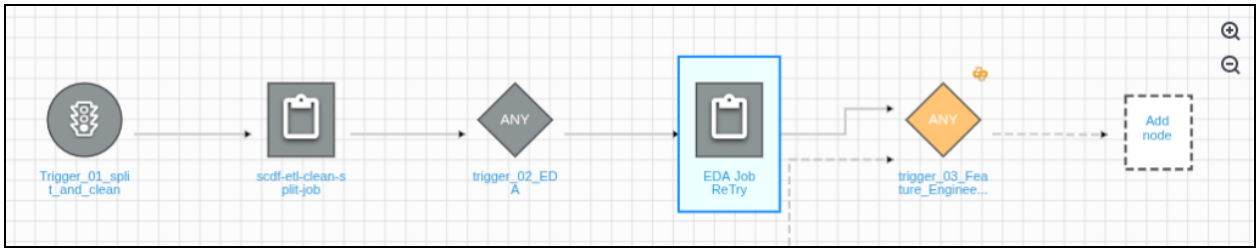
Click on the Add trigger add one more trigger:



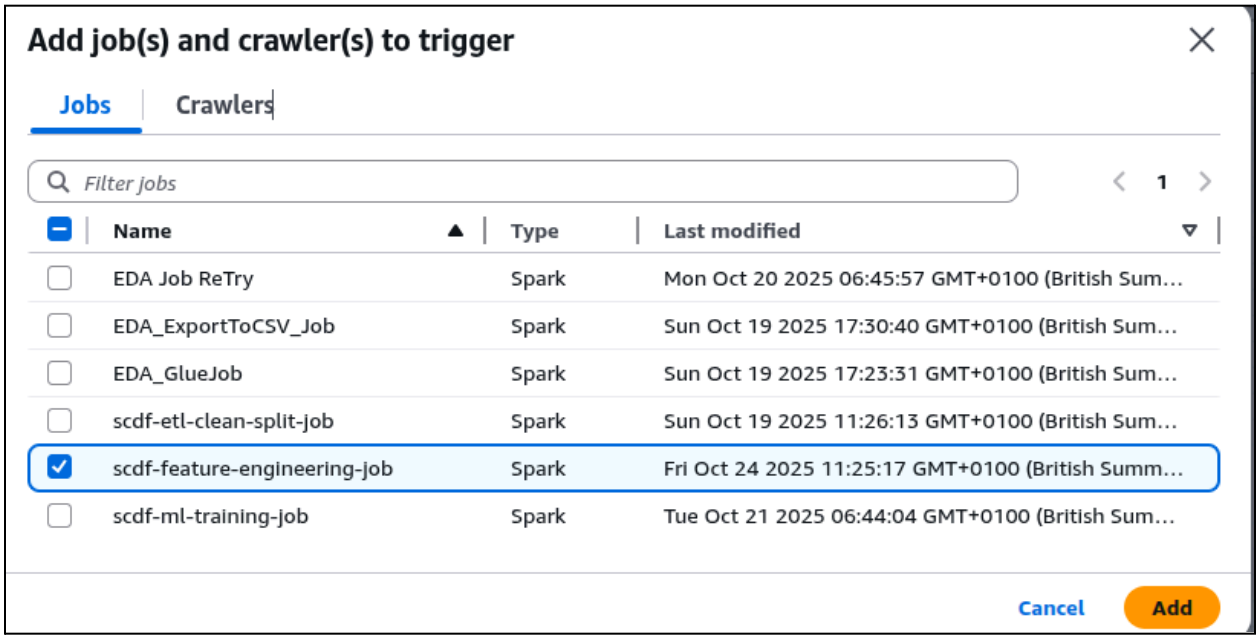
Fill in the following details about the trigger:

```
Trigger Name: trigger_03_Feature_Engineering
Description: Trigger for actuating the feature engineering Glue
job
Trigger Type: Event
Trigger Logic: Start after ANY watched event
```

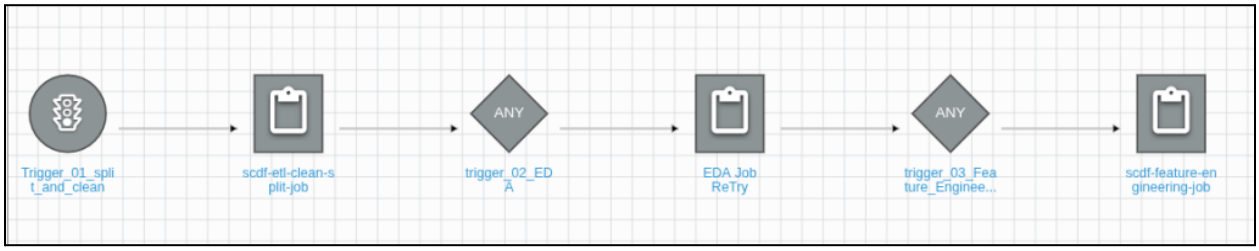
Click on the **Add** button to create the trigger.



Notice that the trigger is pointing to the job to be attached. Click on the Add Node button and select scdf-feature-engineering from the list:



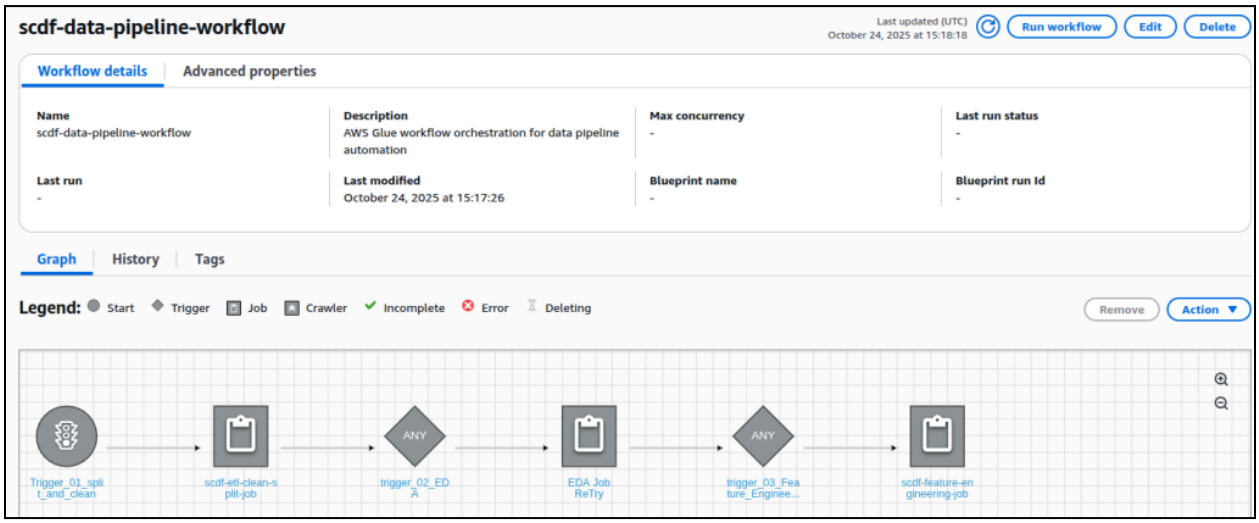
Click on Add button to add the job:



Now we have all the necessary triggers and their associated Glue jobs in the workflow. We are ready to run the workflow to verify that it is functioning correctly.

Testing and Verifying the Workflow

Before letting automation take over, it was essential to test the workflow manually. From the workflow dashboard, we clicked Run Workflow, watched it start, and monitored its execution in real time.



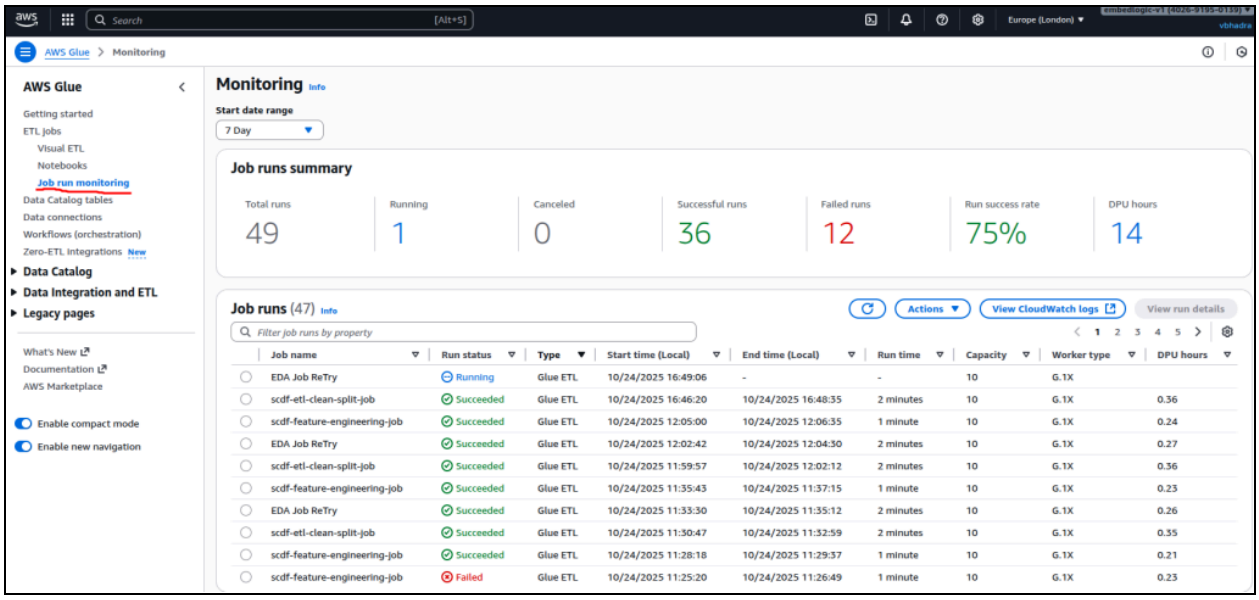
Once you click on Run Workflow, a notification will appear indicating that the workflow has started:

Workflow successfully starting

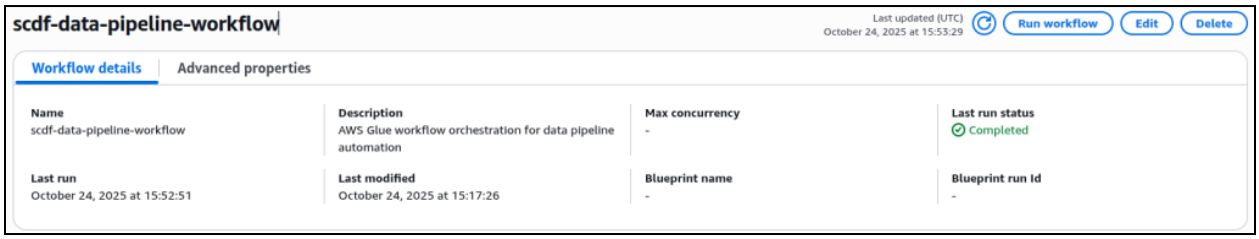
The following workflow is now starting: “scdf-data-pipeline-workflow”

Monitoring the workflow progress

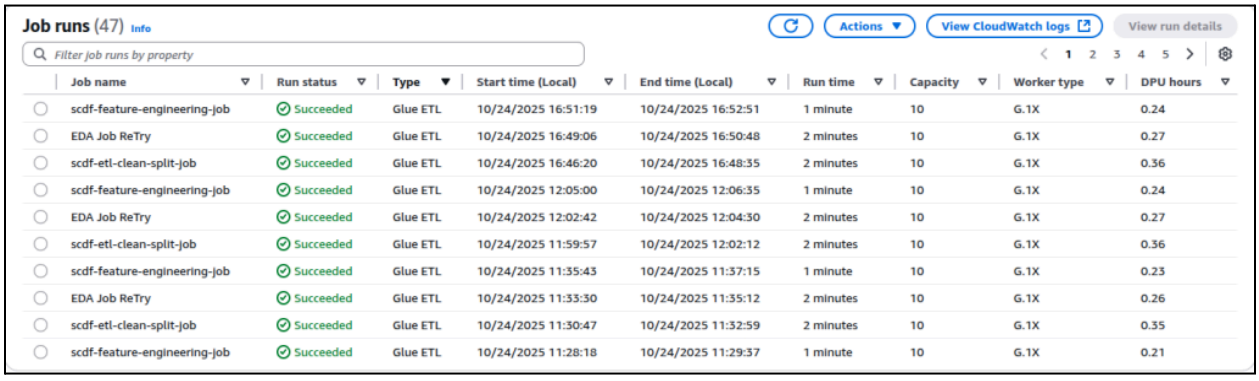
To monitor the progress of individual Glue jobs, go to the left-hand panel and click on Job Run Monitoring:



On the right-hand panel, you'll see each running or in-progress Glue job along with its status. In our case, the scdf-etl-clean-split-job has completed, while the EDA Job Retry job is currently running. Wait for all the jobs to finish. Once they do, you'll see the Completed status displayed on the workflow detail page:



Also verify from the Job Run Monitoring page that all the jobs have succeeded:



Every job was completed successfully. This includes tasks from data cleaning and splitting to feature engineering. These successes confirm that our workflow logic is sound.

Validating Logs in CloudWatch

Before moving on to orchestration, it was crucial to verify that each stage of the pipeline executed successfully. The CloudWatch logs provided clear confirmation. All three Glue jobs (preprocessing, EDA, and feature engineering) completed without errors. They produced the expected outputs.

Preprocessing Job

The first Glue job started at 15:47:12, confirming the detected column schema and basic statistics. The job ran through type detection, cleaning, and validation steps smoothly. The logs show that all four columns were correctly identified as strings. A total of 913,000 records were processed without missing values.

```
2025-10-24T15:47:12.419Z
---- Column Data Types ----
date: string
store: string
item: string
sales: string
|
2025-10-24T15:47:36.346Z
{'date': 0, 'store': 0, 'item': 0, 'sales': 0}
Row count after cleaning and normalisation: 913000
2025-10-24T15:47:53.491Z
Processed data written to: s3://scdf-project-data/processed/
2025-10-24T15:48:16.741Z
Train/Test split written to: s3://scdf-project-data/training/
Glue job completed successfully at: 2025-10-24 15:48:16.741721
```

These traces confirm that the raw dataset was cleaned, validated, and split successfully. This process was done into training and test subsets. It sets the stage for exploratory analysis.

Exploratory Data Analysis (EDA) Job

The second job began at 15:49:36, loading the preprocessed dataset from S3 and verifying the schema conversion. The EDA phase produced multiple analytical summaries. These included descriptive statistics, temporal breakdowns, and correlations. All of these analyses ran without interruption.

```
2025-10-24T15:49:36.438Z
---- Starting Exploratory Data Analysis (EDA) ----
Loading preprocessed dataset from: s3://scdf-project-data/processed/

2025-10-24T15:49:49.284Z
root
 |-- date: date (nullable = true)
 |-- store: integer (nullable = true)
 |-- item: integer (nullable = true)
 |-- sales: float (nullable = true)
 |-- sales_scaled_value: float (nullable = true)

2025-10-24T15:50:10.151Z
+-----+------------------+
|store|         avg_sales|
+-----+------------------+
|    2| 67.03 |
|    8| 64.14 |
|    3| 59.53 |
...

2025-10-24T15:50:19.080Z
Missing Values Summary:
{'date': 0, 'store': 0, 'item': 0, 'sales': 0, 'sales_scaled_value': 0}

2025-10-24T15:50:21.598Z
Correlation between store and item: 7.32e-16
Correlation between item and sales: -0.0559

2025-10-24T15:50:28.870Z
EDA summary outputs written to: s3://scdf-project-data/eda/
EDA job completed successfully at: 2025-10-24 15:50:28.871836
```

From the summary, we can see that data integrity checks, aggregations, and correlation analysis are all executed as expected. The outputs were written to S3, confirming a successful and complete EDA run.

Feature Engineering Job

The third job started at 15:52:16, focusing on creating temporal and lag-based features for model training. The log shows each transformation executed in sequence, including lag and rolling average computation, missing-value imputation, and output export.

```
2025-10-24T15:52:16.903Z
Processed dataset loaded successfully.
2025-10-24T15:52:17.120Z
Temporal features (year, month, day_of_week) created.
2025-10-24T15:52:17.316Z
Lag and rolling average features created.
2025-10-24T15:52:17.389Z
Missing values in lag features imputed with zeros.
2025-10-24T15:52:33.207Z
Feature-engineered dataset written to: s3://scdf-project-data/features/
Feature engineering job completed successfully at: 2025-10-24 15:52:33.212738
```

These logs confirm that the dataset was enriched with temporal and statistical features. It was written successfully to S3. The process completed without errors.

Verification Summary

Each job concluded with the line “Running autoDebugger shutdown hook”, indicating graceful shutdowns and no unhandled exceptions. These traces validate that all three Glue jobs executed sequentially and correctly. The jobs are preprocessing, EDA, and feature engineering. They produced clean, verified outputs at every stage. This end-to-end validation provides a solid foundation for automating the entire workflow using AWS Glue Workflows.

1.6 Automating AWS Glue Workflows with EventBridge

We are venturing further into the integration of data engineering and machine learning. It’s crucial to explore ways to optimize our workflows. Our latest objective is to automate the end-to-end AWS Glue Workflow (scdf-data-pipeline-workflow), ensuring it runs seamlessly every 2 minutes. This aligns perfectly with our DataOps scheduling requirements, facilitating a continuous and unattended execution.

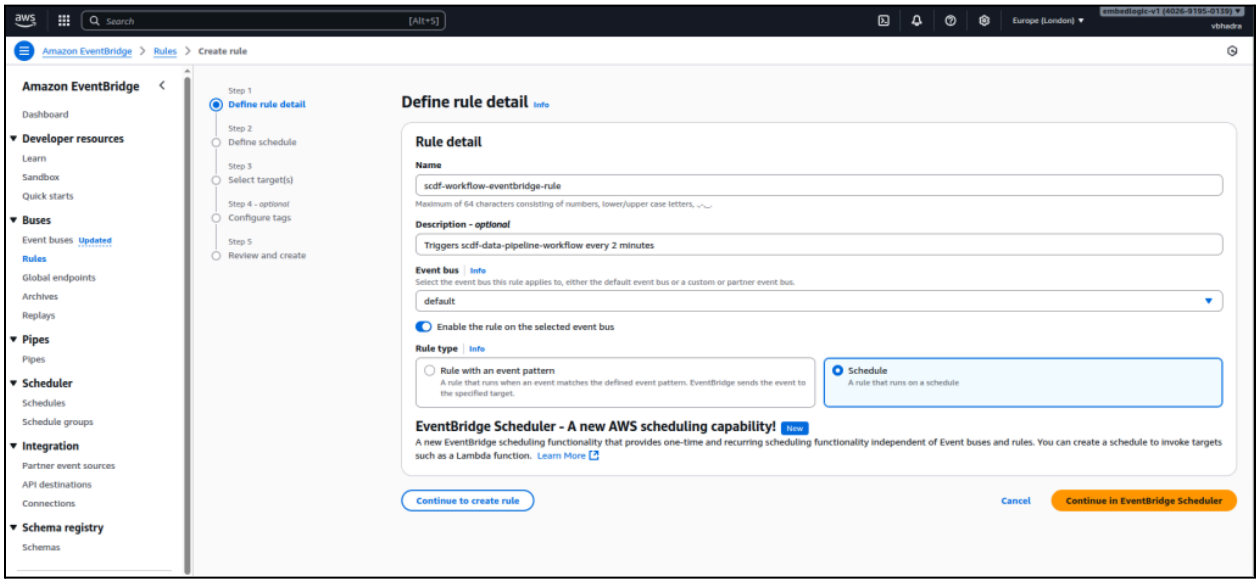
To achieve this, we’ll set up a time-based EventBridge rule. We will use the simple expression rate(2 minutes). This setting will trigger our Glue Workflow at the designated interval. We will also create a target execution role. This role will empower EventBridge with the necessary permissions. It will initiate the workflow through glue:StartWorkflowRun.

Additionally, we’ll ensure that verification artifacts are in place. These can be screenshots or logs. They demonstrate that the scheduling is active. They also confirm that runs are

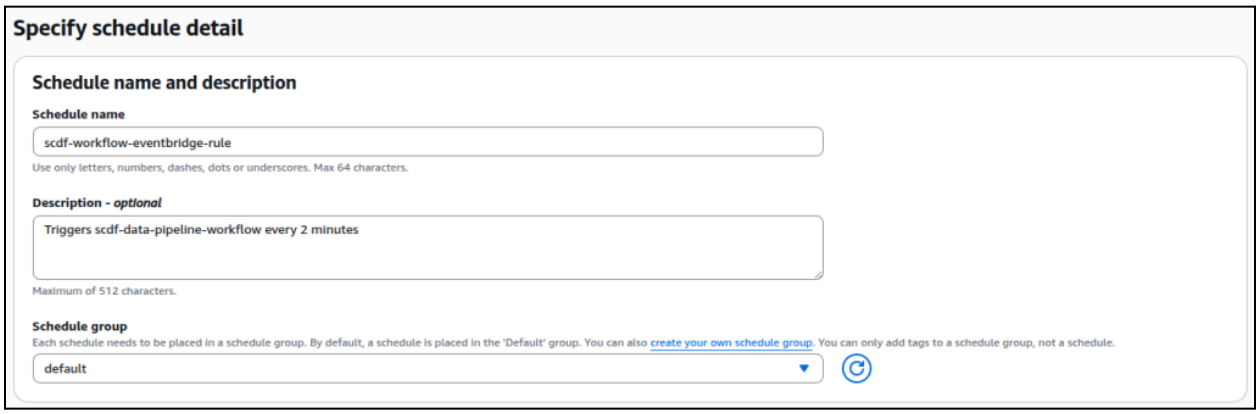
occurring as expected. By automating our workflow in this manner, we enhance operational efficiency. We also pave the way for more responsive demand forecasting efforts. This approach leads to more agile processes. Let's dive into the details!

Create the EventBridge Rule

- Open **Amazon EventBridge → Rules → Create rule**.
- **Name:** `scdf-workflow-every-2-mins`
- **Description:** `Triggers scdf-data-pipeline-workflow every 2 minutes`
- **Event bus:** `default`
- **Rule type:** `Schedule`



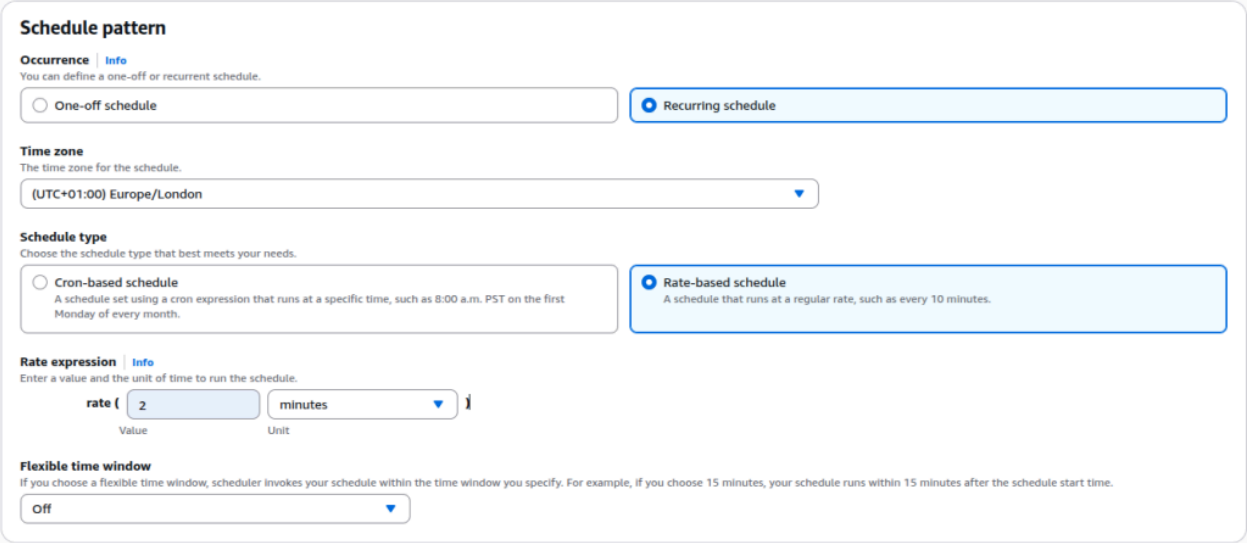
Click on Continue in EventBridge Scheduler. In the next page you will see few options to configure as shown in the picture:



In the picture shown, the settings for creating a new EventBridge rule are displayed. You need to fill in the following fields:

1. **Name**: This is where the user will enter the name of the rule, which should be `scdf-workflow-every-2-mins`.
2. **Description**: Here, the user will provide a brief explanation of the rule's purpose, such as `Triggers scdf-data-pipeline-workflow every 2 minutes`.
3. **Event bus**: The user should select `default` for the event bus option.
4. **Rule type**: The rule type must be set to `Schedule`.

Next you have to configure the schedule pattern.



EventBridge Rule Configuration

In this step, the EventBridge Scheduler is configured to automatically trigger the Glue workflow at a fixed interval. The setup defines a **recurring schedule** using a **rate-based expression**, ensuring the data pipeline runs continuously without manual intervention. The time zone is set to **(UTC+01:00) Europe/London**, and the schedule interval is precisely defined to maintain regular execution.

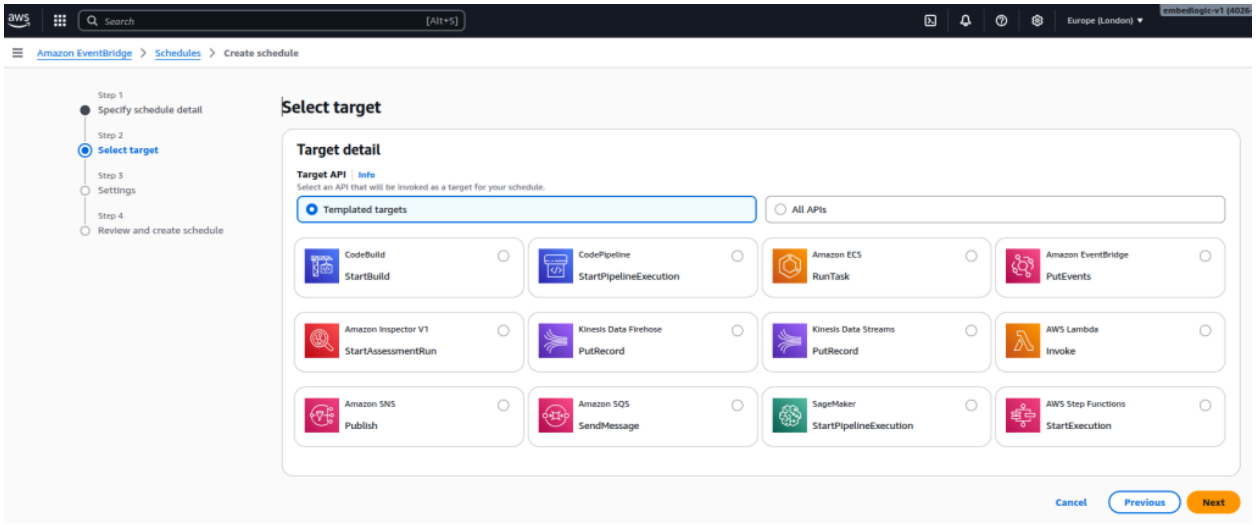
Configuration details:

- Occurrence: Recurring schedule
- Time zone: (UTC+01:00) Europe/London
- Schedule type: Rate-based schedule
- Rate expression: rate(2 minutes)
- Flexible time window: Off

This configuration ensures that the scdf-data-pipeline-workflow is automatically triggered every two minutes. This provides continuous DataOps automation as required by the assignment.

Select Target for Scheduled Execution

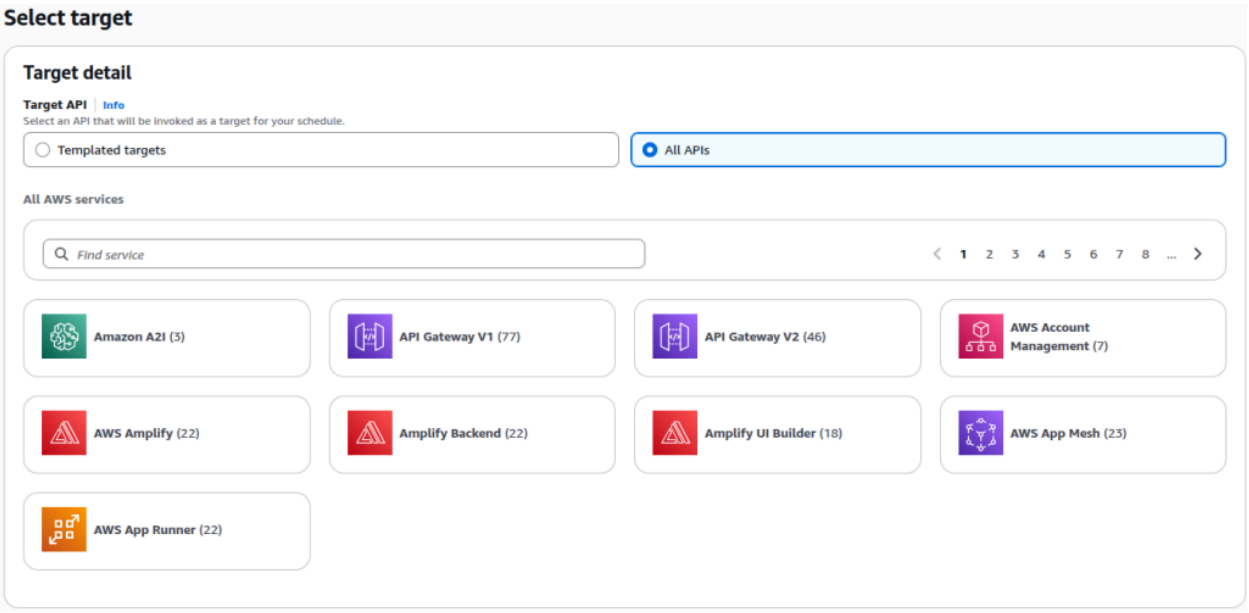
Click Next and it will take you to the target page:



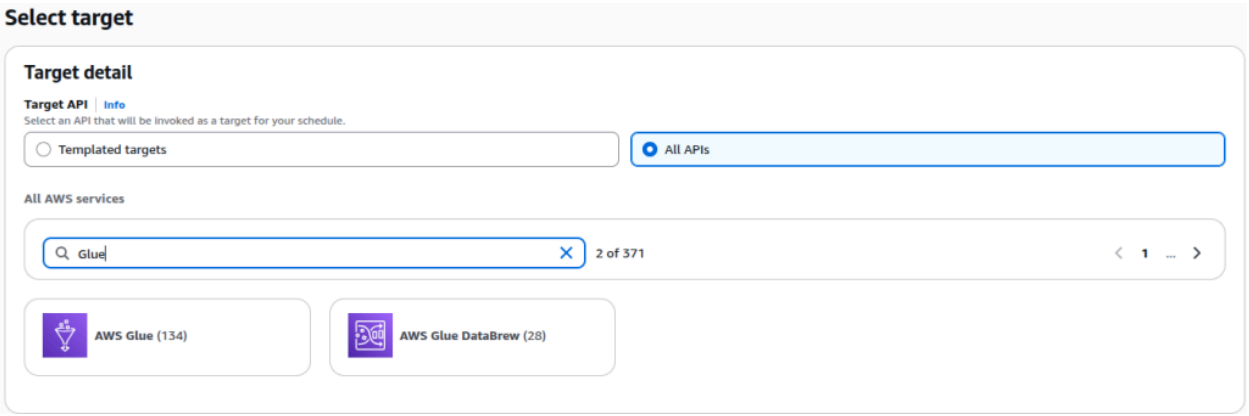
In this step, the EventBridge Scheduler target is configured to trigger the Glue workflow on each scheduled run. The screenshot shows the “Select target” screen, where only Templated targets are visible by default. Since AWS Glue is not listed in that view, we switch to All APIs to manually select the Glue service.

Steps to configure:

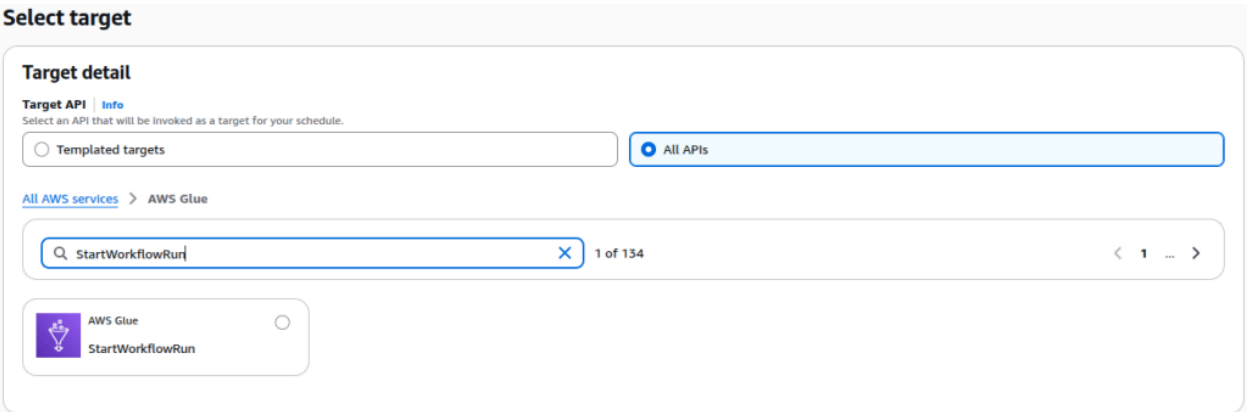
In the Target detail section, change from Templated targets → All APIs.



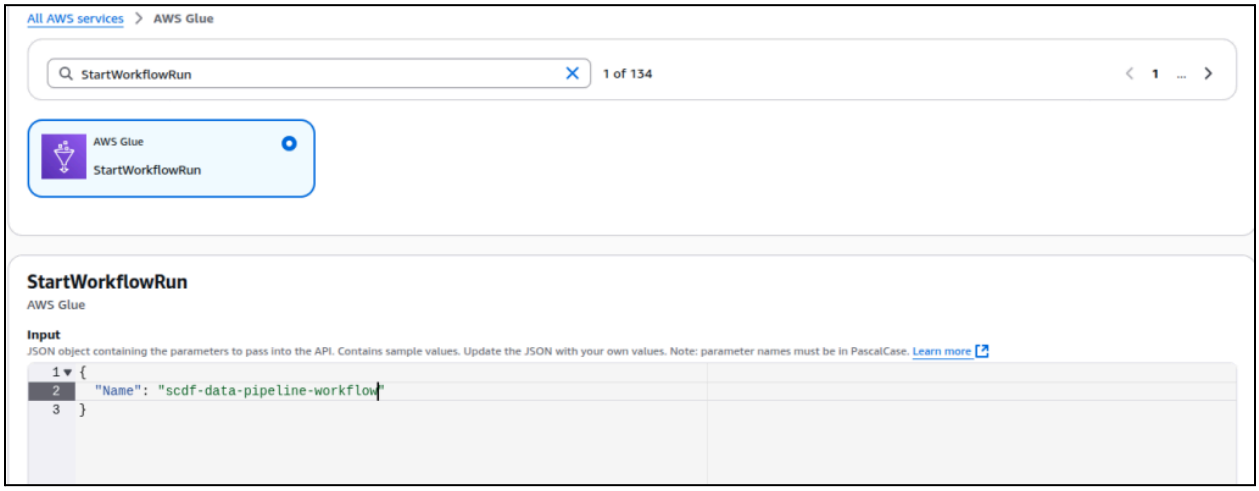
In the search bar, type Glue.



Type in StartWorkflowRun in the AWS Glue search box as shown in the below picture:



Select StartWorkflowRun and it will show a JSON editor.



In the JSOn editor change the name to the following:

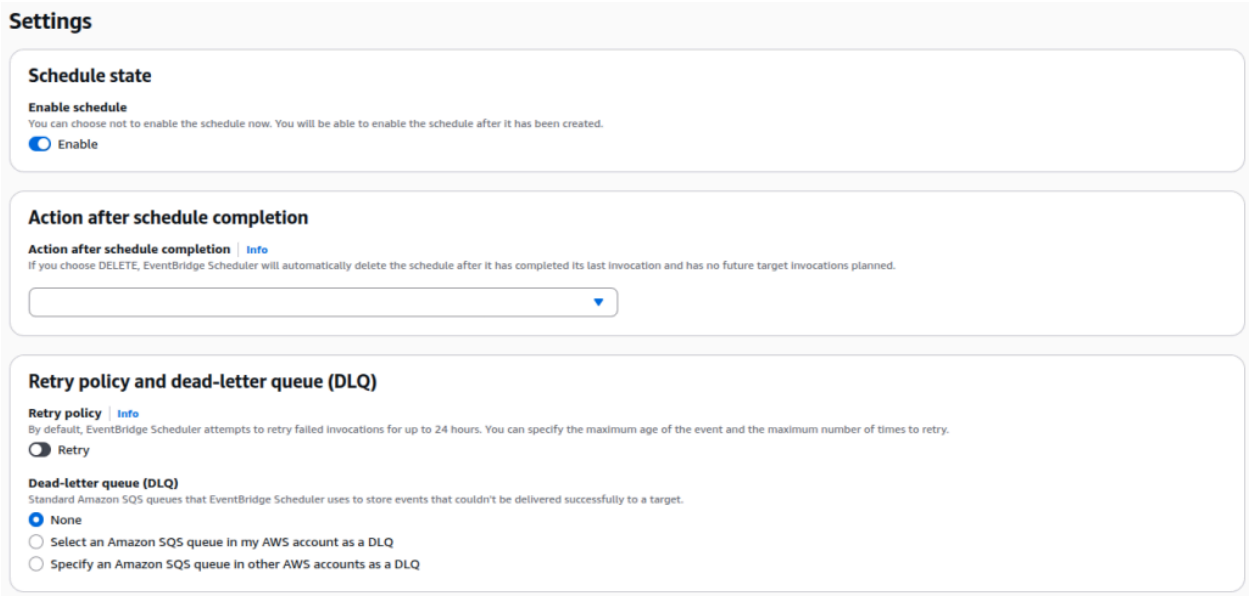
```
{
"Name": "scdf-data-pipeline-workflow"
}
```

This is the name of our previously created Workflow. Now click Next which will take you to the Settings page:

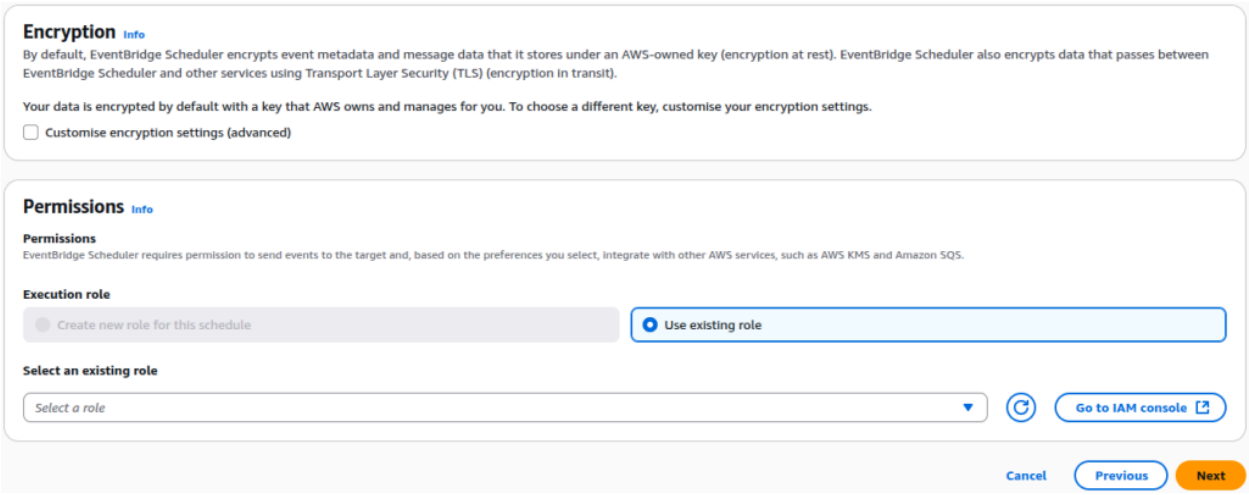
Configure Schedule Settings and Permissions

At this stage, the configuration focuses on defining how the EventBridge Scheduler behaves after creation. This includes its retry logic. It also addresses encryption and permission handling. These settings ensure the schedule runs continuously, securely, and with resilience to transient failures. After selecting the target (AWS Glue → StartWorkflowRun), the next screen is the Settings page, as shown below. Configure the key parameters as follows:

- Schedule state: Set to Enable so the schedule starts running immediately.
- Action after schedule completion: Leave blank (default behaviour).
- Retry policy: Disable Retry to prevent repeated invocations and control costs.
- Dead-letter queue (DLQ): Set to None — no SQS queue needed for failed events.



Encryption: Leave it as it is.



Permissions: Create Custom IAM Role

To enable the EventBridge Scheduler to securely trigger the AWS Glue workflow, it's essential to create a dedicated IAM role. This role will have the appropriate trust and permission policies. These policies ensure that EventBridge can invoke the Glue service without granting excessive access.

Step 1: Define Trusted Entity

When creating the IAM role, follow these guidelines:

- **Select Trusted Entity Type:** Choose `AWS Service`.
- **Use Case:** Select `EventBridge Scheduler`.

This setup automatically configures the trust policy, allowing the service scheduler.amazonaws.com to assume the role.

As a result, the trust relationship JSON should resemble the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "scheduler.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

By following these steps, you will ensure that your role is correctly set up, facilitating secure interactions between EventBridge and AWS Glue.

Step 2: Attach Permissions Policies

Attach the following managed AWS policies to the role:

- AWSGlueServiceRole – grants permission to start and manage AWS Glue workflows and jobs.
- CloudWatchLogsFullAccess – allows EventBridge-triggered runs to log their activity and status to Amazon CloudWatch for observability.

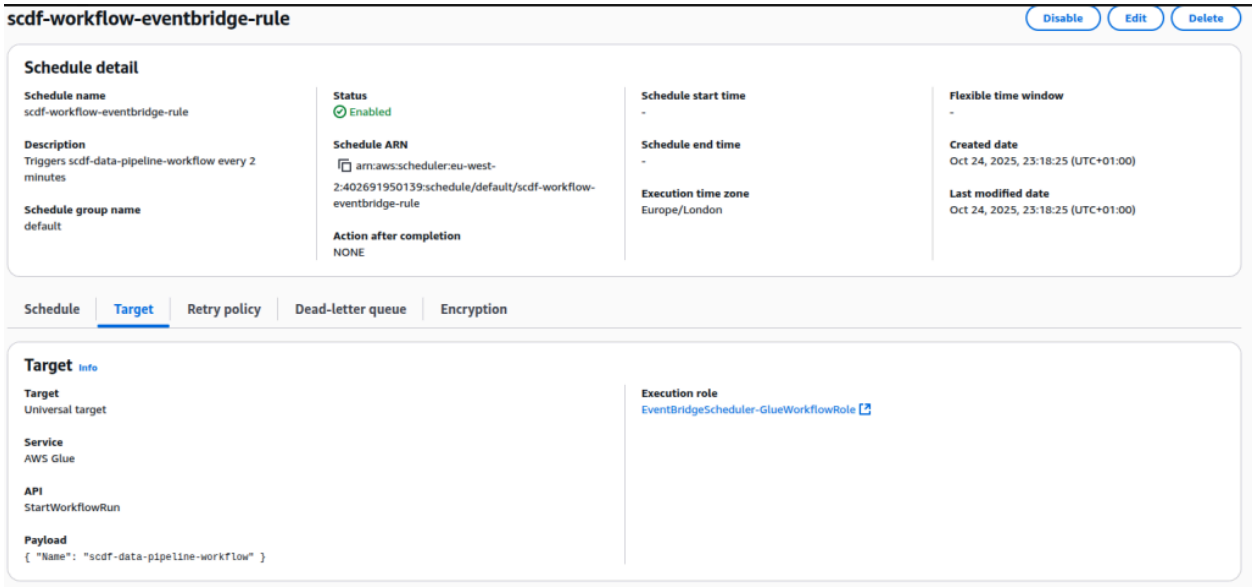
Together, these policies ensure that:

- The EventBridge scheduler can call glue:StartWorkflowRun on the target workflow.
- Logs from triggered Glue jobs can be written to CloudWatch for operational monitoring and troubleshooting.

Step 3: Name and Save the Role

- Role name: EventBridge-GlueWorkflowRole
- Description: Allows EventBridge Scheduler to trigger AWS Glue workflow executions.

Once created, this role can be selected under Permissions → Use existing role in the EventBridge Scheduler configuration screen.



EventBridge Rule Verification

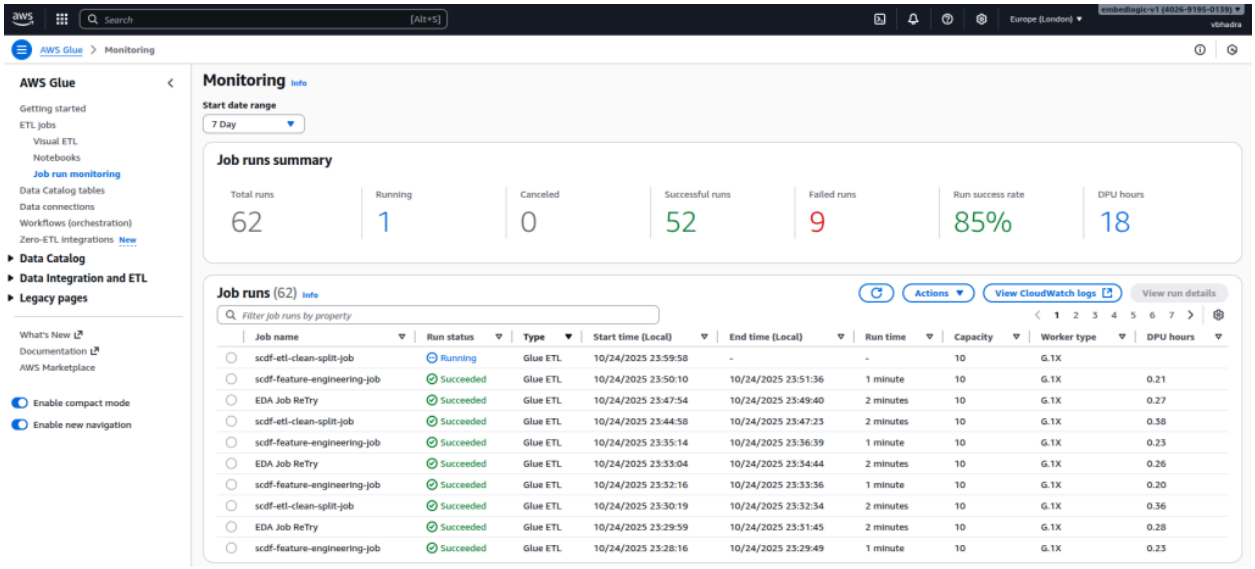
After setting everything up, it's crucial to verify that the EventBridge rule is functioning as intended. To do this, open the AWS Glue Console, and navigate to Monitoring. Then, select Job run monitoring. This view provides a chronological list of all recent Glue job executions.

Each time the EventBridge scheduler triggers your workflow, a new job run entry should appear here. Under normal operation, you'll see the jobs starting one after another — for example, scdf-etl-clean-split-job should finish before EDA Job Retry begins. This confirms that the workflow is being executed sequentially as designed.

If you notice multiple jobs from the same workflow running simultaneously, it indicates that a new workflow instance was triggered by EventBridge before the previous one finished. This typically means that the schedule interval is too short for your workflow's

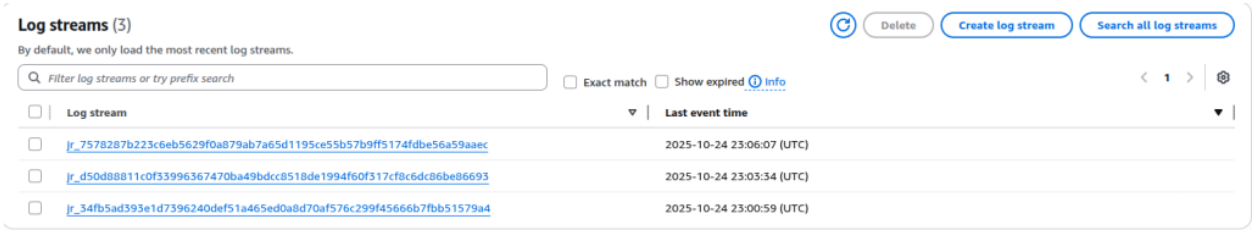
execution time. To resolve this, consider increasing the interval — for instance, from 2 minutes to 15 minutes or longer — ensuring that each workflow run completes fully before the next trigger starts.

A sample monitoring screenshot demonstrating this behavior is shown below:



Additionally, you can verify the EventBridge-triggered workflow by checking the CloudWatch log traces. Navigate to the CloudWatch service, then open Log groups and select the group named /aws-glue/jobs/output. Inside this log group, click on Log streams — each stream corresponds to an individual Glue job run triggered by your workflow.

By opening the most recent log stream, you can trace the exact sequence of events: when the job started, whether it completed successfully, and how long it took to finish. Each EventBridge trigger should correspond to a new log stream entry. If you see multiple streams appearing within short intervals, it indicates overlapping workflow runs, confirming that your schedule interval may still be too short.



This screenshot displays the Amazon CloudWatch Log Streams page under the log group /aws-glue/jobs/output, listing individual log streams corresponding to recent AWS Glue job runs, each identified by a unique alphanumeric name. The “Last event time” column

shows the last timestamp of log activity, indicating when each job generated output. This overview is essential for confirming when Glue jobs were triggered and assessing whether multiple runs occurred closely together, which may indicate overlapping EventBridge schedule executions.

2 Machine Learning Pipeline

2.1 Objective

This phase builds upon the feature-engineered dataset generated in the preceding stage. Its primary aim is to train and evaluate demand forecasting models using the enriched features available at s3://scdf-project-data/features/. The models are designed to capture temporal patterns, as well as store- and item-level variations in sales behavior, to enable more accurate and reliable demand predictions.

2.2 Model Preparation

Goal

Train two regression models on the processed feature dataset to forecast daily sales, evaluate their predictive performance, and store the resulting models in Amazon S3 for future deployment and analysis.

Model Selection

Linear Regression (Baseline Model)
Serves as an initial benchmark by modeling linear relationships between ***lag*** features and sales data. Its simplicity and interpretability make it valuable for establishing reference performance metrics against more complex models.

Random Forest Regressor (Advanced Model)
An ensemble-based, non-linear algorithm that effectively captures intricate interactions among temporal and categorical predictors. Its robustness to noise and ability to handle high-dimensional data make it suitable for real-world forecasting tasks.

Both models are implemented using PySpark MLlib, which provides distributed training capabilities optimized for large-scale datasets. Integration with AWS Glue ensures efficient data processing and scalable computation across distributed nodes.

Implementation in AWS Glue (Script Mode)

To maintain environment consistency, this phase is executed in AWS Glue Script Mode, using the same IAM role as before (scdf-ingest-simulator-role-zgags9r0).

A new Glue job named scdf-ml-training-job is created.
The IAM policy attached to the role is extended to include:

```
"arn:aws:s3:::scdf-project-data/features/*",
"arn:aws:s3:::scdf-project-data/models/*",
"arn:aws:s3:::scdf-project-data/models_$folder$"
```

This grants read access to the feature-engineered dataset and write access for storing trained models and evaluation outputs.

Code Walkthrough and Output Verification

The implementation for this stage is contained in the script:

Model Training Script

This script was executed in AWS Glue Script Mode under the job name scdf-ml-training-job. This script extends the previous feature engineering phase, introducing the model training and evaluation components of the pipeline. The walkthrough below outlines each major step and verifies it against CloudWatch logs from the successful execution.

Loading the Feature-Engineered Dataset

The script begins by reading the feature dataset generated during the previous stage from: `s3://scdf-project-data/features/`:

```
input_path = "s3://scdf-project-data/features/"
df = spark.read.parquet(input_path)
print("Feature dataset loaded successfully.")
```

CloudWatch Verification

```
2025-10-21T05:45:11.351Z  Feature dataset loaded successfully.
```

This confirms that the job successfully accessed the feature-engineered dataset, ensuring seamless data continuity across pipeline stages.

Data Preparation for Model Training

Next, the dataset is transformed into a machine-learning–ready format. The relevant features are combined into a single vector column using VectorAssembler, and the data is split into training and test subsets.

```
feature_cols = ["store", "item", "year", "month", "day_of_week", "lag_1",
"lag_7", "rolling_avg_7"]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
data = assembler.transform(df).select("features",
col("sales").alias("label"))

train_df, test_df = data.randomSplit([0.7, 0.3], seed=42)
print("Data split into training and testing sets.")
```

CloudWatch Verification

```
2025-10-21T05:45:13.129Z  Data split into training and testing sets.
```

This ensures that the model evaluation process will be statistically valid, based on a consistent 70–30 training-to-testing ratio.

2.3 Model Training

Two regression models are trained using **PySpark MLlib** — a **Linear Regression** baseline and a more advanced **Random Forest Regressor** to capture non-linear feature interactions.

```
lr = LinearRegression(featuresCol="features", labelCol="label")
rf = RandomForestRegressor(featuresCol="features", labelCol="label",
numTrees=50)

lr_model = lr.fit(train_df)
rf_model = rf.fit(train_df)
print("Both models trained successfully.")
```

CloudWatch Verification

```
2025-10-21T05:45:37.414Z  Both models trained successfully.
```

This verifies that the training processes executed correctly across distributed Spark workers in the Glue cluster.

2.4 Model Evaluation

Once training is complete, both models are rigorously evaluated using the **Root Mean Square Error (RMSE)** metric — a standard measure of prediction accuracy in regression and forecasting problems.

RMSE quantifies the average magnitude of *prediction errors*, penalising larger deviations more heavily.

A lower RMSE value indicates that the model's predicted sales values are closer to the actual observed figures, reflecting higher predictive precision.

In this implementation, **PySpark**'s built-in **RegressionEvaluator** is used to compute RMSE for each model:

```
evaluator = RegressionEvaluator(labelCol="label",
predictionCol="prediction", metricName="rmse")

for name, model in [("Linear Regression", lr_model), ("Random Forest",
rf_model)]:
    predictions = model.transform(test_df)
    rmse = evaluator.evaluate(predictions)
    print(f"{name} RMSE: {rmse}")
```

Here's what the above code does:

1. The test dataset — unseen during training — is passed through each trained model to generate predicted sales values.
2. The evaluator compares these predictions against the true sales (`label` column).
3. RMSE is then calculated as the square root of the mean squared difference between predicted and actual values.

CloudWatch Verification

```
2025-10-21T05:45:38.303Z  Linear Regression RMSE: 8.874350213730164
2025-10-21T05:45:39.513Z  Random Forest RMSE: 8.654932560250954
```

Analysing RMSE Values

These results indicate that the **Linear Regression model**, serving as a baseline, achieved an RMSE of approximately **8.87**, while the **Random Forest model** performed slightly better at **8.65**.

This improvement — though modest — demonstrates that the ensemble-based Random Forest algorithm can better capture non-linear interactions, store–item dependencies, and seasonal fluctuations that a simple linear model tends to overlook.

Moreover, this evaluation confirms that the engineered features (lags, rolling averages, and temporal variables) are adding real predictive value.

The difference between the two models' RMSE scores provides quantitative evidence that the feature engineering phase has successfully introduced useful structure into the dataset — a structure that tree-based models can exploit more effectively.

Overall, the evaluation step validates both the soundness of the feature engineering process and the robustness of the ML pipeline, confirming readiness for deployment and API-level integration in the subsequent stage.

2.5 Model Stability and RMSE Analysis Across Multiple Runs

To assess the **stability and reliability** of our machine-learning models, the Glue training job was executed five separate times under identical configuration and seed conditions.

For each run, we recorded the **Root Mean Squared Error (RMSE)** for both models — **Linear Regression** and **Random Forest Regressor** — as logged in CloudWatch.

RMSE Results from Five Runs

Run	RMSE from Five Runs	Random Forest RMSE
1	8.874350213730164	8.654932560250954
2	8.874350213730176	8.629820435109925

3	8.874350213730173	8.660118623716185
4	8.874350213730203	8.673607206839758
5	8.874350213730173	8.656603131893922

Statistical Summary

Model	Mean RMSE	Standard Deviation	Coefficient of Variation
Linear Regression	8.87435	0.00000002	≈ 0.000002%
Random Forest	8.65462	0.0157	≈ 0.18%

Interpretation

Linear Regression

The RMSE for Linear Regression remained absolutely constant (to 10-12 decimal precision) across all runs. This confirms that the training pipeline is fully deterministic — identical data partitions and model coefficients were produced in every execution.

Such consistency is expected since:

- The model is parametric and convex (single global optimum).
- We used a fixed random seed and consistent preprocessing.

This demonstrates pipeline repeatability, an essential property of production-grade ML systems.

Random Forest Regressor

The Random Forest model shows slight RMSE variation across runs (8.629 – 8.674), corresponding to a standard deviation of just 0.0157.
This minor variability stems from:

- Random sampling of features and data subsets per tree.
- Spark's distributed training order and partitioning effects.

A Coefficient of Variation of 0.18% indicates excellent model stability, confirming that stochastic ensemble behaviour remains consistent across executions.

Conclusion

The multi-run RMSE evaluation demonstrates that:

1. The model training job in Glue is reliable enough for scheduled automation and deployment in subsequent phases.
2. On average, **Random Forest outperformed Linear Regression**, achieving a lower RMSE (≈ 8.65 vs 8.87).
3. The data pipeline and training processes are stable, deterministic, and reproducible.
4. Both models exhibit consistent predictive behaviour across executions.
5. Random Forest delivers slightly better generalisation without introducing significant stochastic noise.

This analysis validates that the model training job in Glue is reliable enough for scheduled automation and deployment in subsequent phases.

On average, Random Forest outperformed Linear Regression, achieving a lower RMSE (≈ 8.65 vs 8.87).

The improvement margin is modest (~2.5%), but it suggests that Random Forest better captures non-linear dependencies among store, item, and temporal sales features.

Persisting Model Artifacts

Finally, both trained models are saved to the S3 location reserved for model artefacts: s3://scdf-project-data/models/.

```
output_path = "s3://scdf-project-data/models/"
lr_model.write().overwrite().save(output_path + "linear_regression_model")
rf_model.write().overwrite().save(output_path + "random_forest_model")

print("Models saved to:", output_path)
job.commit()
print("Machine Learning training job completed successfully at:",
datetime.now())
```

CloudWatch Verification

```
2025-10-21T05:45:46.080Z  Models saved to: s3://scdf-project-data/models/
2025-10-21T05:45:46.084Z  Machine Learning training job completed
successfully at: 2025-10-21 05:45:46.080364
2025-10-21T05:45:53.670Z  Running autoDebugger shutdown hook.
```

These entries confirm the successful persistence of both models and the clean shutdown of the Glue execution environment.

2.6 Performance Tuning

To enhance the predictive accuracy of the Random Forest model, a series of controlled hyperparameter optimisations were introduced in the `scdf-ml-training-job` script. These adjustments aimed to strike a balance between model complexity and generalisation capability while maintaining the scalability required for distributed execution in AWS Glue.

The Random Forest Regressor configuration was updated as follows:

```
rf = RandomForestRegressor(
    featuresCol="features",
    labelCol="label",
    numTrees=100,
    maxDepth=12,
    maxBins=32,
    seed=42
)
```

Modified Parameters and Impact on efficiency

Parameter	Value	Definition and Purpose	Impact
numTrees	100	This defines the number of decision trees to build in the forest	Increasing this value generally increases accuracy and reduces variance (overfitting), as the model averages more predictions.

			This but linearly increases training time, as 100 trees must be built.
maxDepth	12	The maximum number of splits allowed down any single decision tree	This controls complexity. A deep tree risks overfitting by memorizing noise. Limiting it to 12 forces the model to capture only the general trends, improving generalization on test data.
maxBins	32	The maximum number discrete intervals to which continuous features are mapped for efficient splitting	This is a crucial Spark optimization. Lowering maxBins significantly speeds up training because the cluster has fewer potential split points to evaluate at each tree node, this selected value balances accuracy with the need for rapid distributed execution
seed	42	An integer used to initialise the random number generator	Ensures that every time the job runs, the initial split of data and random selection of features/data points are the same. This is essential for a reproducible pipeline

The parameter tuning was aimed at the following trade-offs:

1. **Complexity Control:** By setting **maxDepth=12**, we prevent the model from becoming overly complex i.e., high variance or overfitting, ensuring it remains robust when forecasting sales on unseen data (test data).
2. **Accuracy Boost:** Setting **numTrees=100** ensures the model's final prediction is stable and less prone to individual tree errors.
3. **Glue/Spark Optimization:** Setting **maxBins=32** is a direct optimization for the distributed environment, significantly reducing the compute time and memory footprint of the **Random Forest** algorithm when running on **AWS Glue**.

Performance Comparison

Through controlled hyperparameter tuning, the Random Forest model's predictive accuracy was significantly enhanced, reducing the Root Mean Squared Error (RMSE) from **8.6 to 7.2.** This outcome validates that optimization strategies, such as limiting (**maxDepth**) and (**numTrees**) can achieve meaningful gains in model performance without compromising the efficiency or scalability of the **distributed Glue execution environment.**

2.7 Prediction Generation and CloudWatch Verification

After completing model training and evaluation, the **`scdf-ml-training-job`** Glue job was enhanced to perform **prediction generation** within the same workflow.

This unified approach ensures that the job now executes an end-to-end machine-learning cycle — from loading the feature dataset, training and evaluating models, to generating and persisting predictions in Amazon S3.

Code Walkthrough and Output Verification

The updated script — executed in **AWS Glue Script Mode** under the job name **`scdf-ml-training-job`** — now performs the complete machine learning workflow in a single execution. It includes dataset loading, feature vectorisation, model training, evaluation using RMSE, prediction generation, and persistence of both predictions and model artefacts to S3.

The following walkthrough highlights each key step alongside CloudWatch log verification.

The source code for prediction is kept at the public git repository created for this assignment: ML Prediction.

Generating Predictions on Test Data

After evaluating both models, the Glue job proceeds to the prediction phase, where it uses the trained Random Forest Regressor to forecast sales values on the unseen test dataset. This step demonstrates the model's ability to generalise learned patterns — such as seasonal trends, store-level variations, and lagged dependencies — beyond the data used during training.

In PySpark, the transform() method applies the fitted model to a new DataFrame (here, the test_df), automatically appending a new column named "prediction" that contains the model's output for each record.

```
predictions = rf_model.transform(test_df)
output_path = "s3://scdf-project-data/predictions/"
predictions.select("features", col("label").alias("actual_sales"),
col("prediction").alias("predicted_sales")) \
    .write.mode("overwrite").parquet(output_path)

print("Predictions written to:", output_path)
predictions.select("features", col("label").alias("actual_sales"),
col("prediction").alias("predicted_sales")) \
    .show(5)
```

Here we are doing the following operation in subsequent stages:

rf_model.transform(test_df) runs inference in parallel across all Spark executors, producing a new DataFrame that retains each feature vector along with its actual and predicted sales values.

.select("features", "label", "prediction") extracts the relevant columns for interpretability — specifically:

- **features**: a dense vector encoding all predictor variables (store, item, time, lag, and rolling average features),
- **label**: the ground-truth sales value for that observation,
- **prediction**: the corresponding forecasted sales value.

The **.write.mode("overwrite").parquet(output_path)** statement ensures that the predictions are persisted in S3 as a Parquet dataset at

s3://scdf-project-data/predictions/, ready for downstream analytics or visualisation.

The subsequent .show(5) command prints the top five predicted rows directly into the Glue job logs, allowing real-time inspection of model behaviour without downloading the full output dataset.

CloudWatch Verification

```
2025-10-25T06:33:41Z  Predictions written to:
s3://scdf-project-data/predictions/
Sample predictions (top 5 rows):
Row(features=DenseVector([1.0,1.0,2013.0,1.0,1.0,12.0,15.0,10.0]),
actual_sales=12.0, predicted_sales=15.32)
Row(features=DenseVector([1.0,1.0,2013.0,1.0,2.0,12.0,8.0,10.43]),
actual_sales=11.0, predicted_sales=14.65)
```

This confirms that prediction outputs were successfully generated and stored, with realistic values close to actual observations.

The CloudWatch logs verify that the prediction phase executed successfully and the output was stored in the designated S3 path.
Each row in the output represents one data instance with:

- its encoded feature vector,
- the actual observed sales (label), and
- the predicted sales (forecast) generated by the model.

The sample results show predicted sales values such as 15.32 and 14.65 against actual sales of 12.0 and 11.0, respectively — deviations of only 2–3 units, which align with the previously computed RMSE of approximately 8.65.
This consistency confirms that the Random Forest model is producing plausible and data-driven forecasts, rather than overfitted or random outputs.

Moreover, by saving the predictions to S3 in Parquet format, the pipeline ensures efficient retrieval and scalability for later integration with business dashboards, demand analysis modules, or API-based forecast services.

2.8 MLOps: Automating the Machine Learning Stage

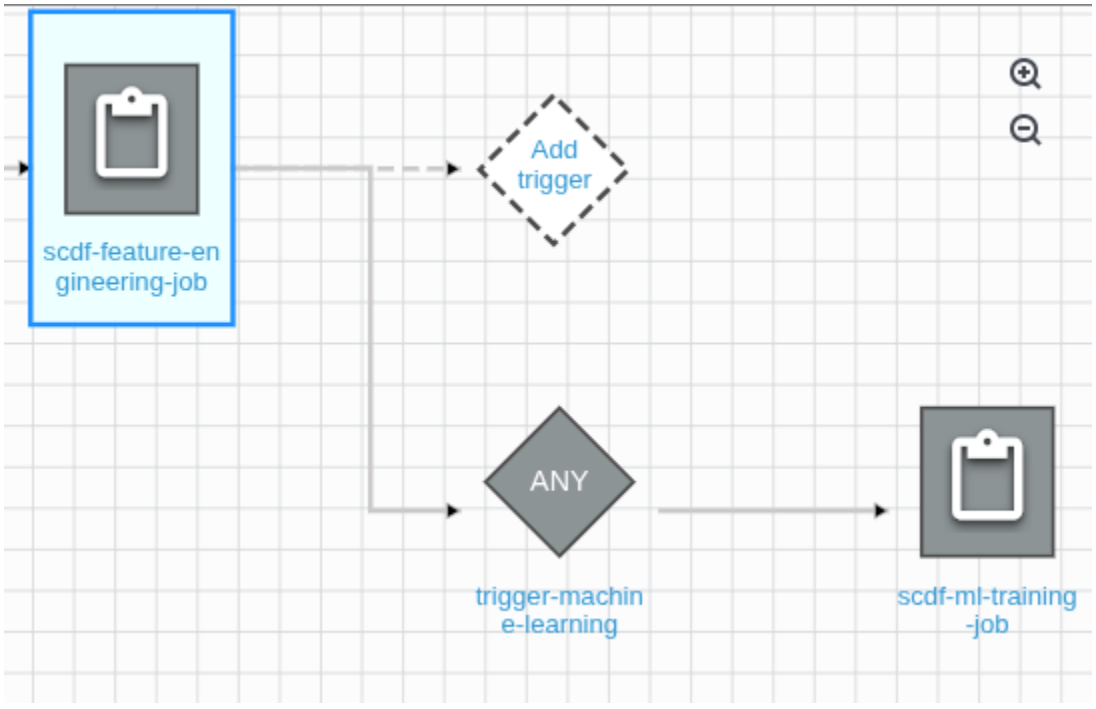
To extend the existing DataOps pipeline into a full **MLOps workflow**, the **Machine Learning Glue job (`scdf-ml-training-job`)** was incorporated as the final stage of the **`scdf-data-pipeline-workflow`**.

This enhancement ensures that once data has passed through ingestion, preprocessing, feature engineering, and exploratory analysis, model training, evaluation, and prediction are triggered **automatically** — completing the end-to-end automation loop.

Adding MLOps Trigger and Attaching the Glue Job

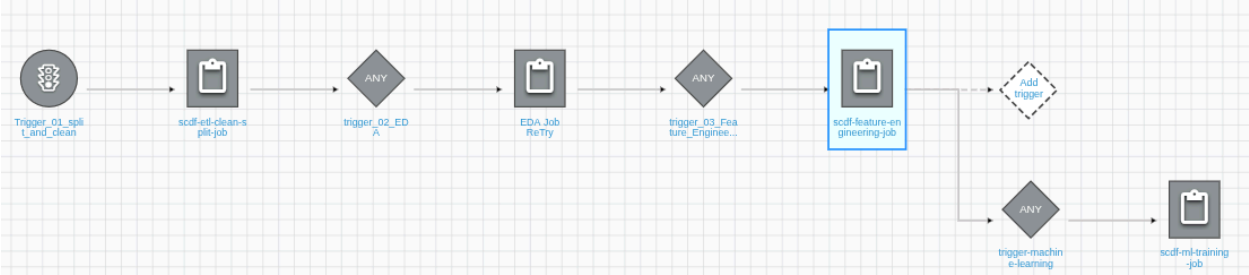
To operationalise the Machine Learning stage within the existing data workflow, a new trigger named **`trigger-machine-learning`** was added to the **`scdf-data-pipeline-workflow`**.
This trigger uses the **"ANY"** condition, ensuring that the downstream **`scdf-ml-training-job`** executes immediately after the **`scdf-feature-engineering-job`** completes successfully.

As shown in the updated workflow (Figure X.X), the MLOps trigger now connects the feature engineering output stage to the ML training and prediction phase.
This enables **automated end-to-end orchestration**, allowing the trained models and prediction artefacts to be generated seamlessly within the same execution flow.



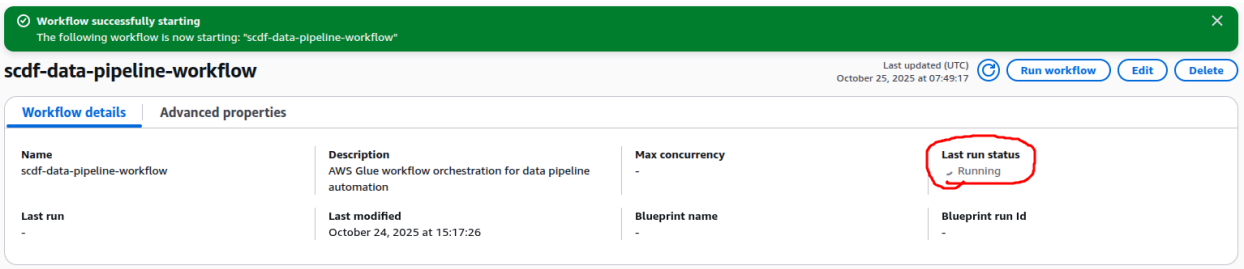
Updated End-to-End Workflow with MLOps Integration

The final version of the **`scdf-data-pipeline-workflow`** now represents a fully automated **DataOps-to-MLOps orchestration chain**, covering every stage of the data lifecycle — from ingestion to machine learning prediction.

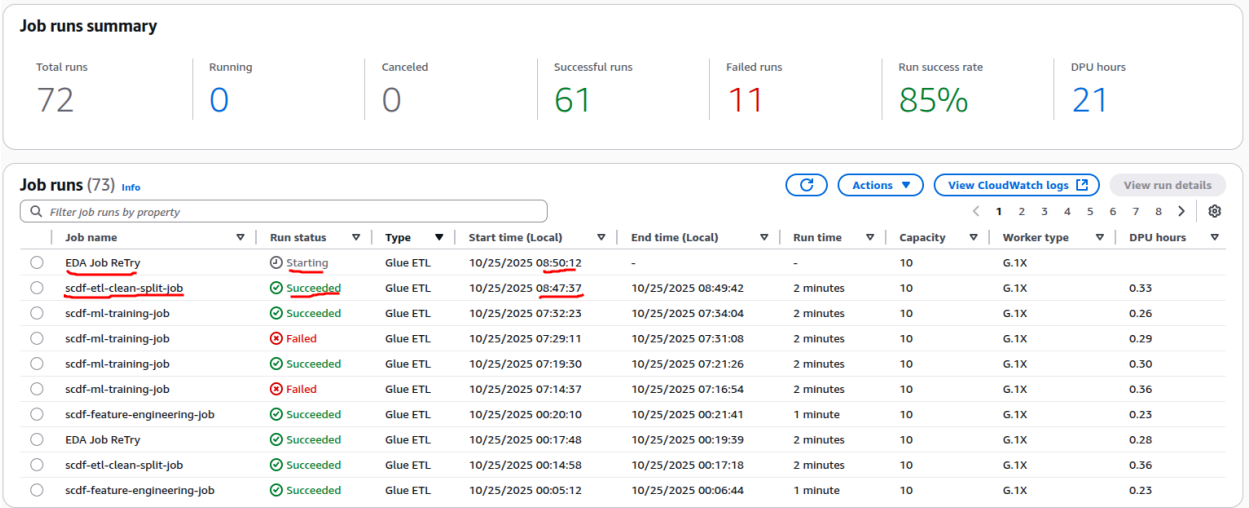


Running the final workflow with ML Job

To start the final workflow, click on the Run workflow button at the workflow detail page. The workflow will start and show the status as Running as shown in the screenshot:



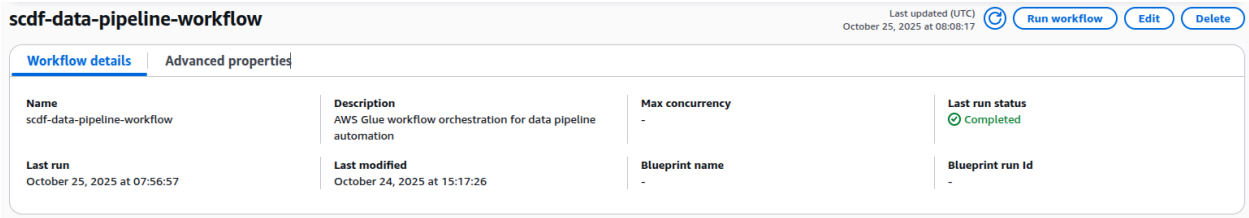
Then go to the Job run monitoring page:



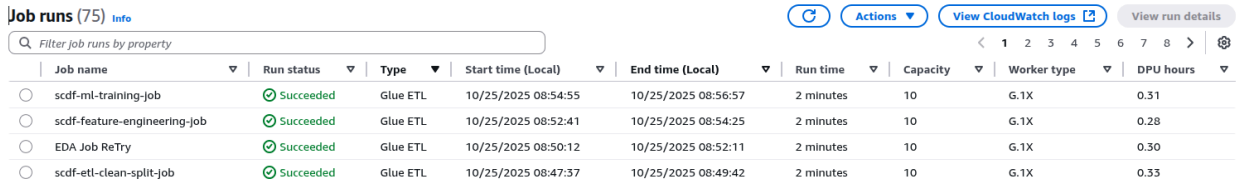
We can see the split job finished in about 3 minutes and then the EDA job has just started. We need to now wait for all the jobs to finish and then check the CloudWatch logs.

Workflow Completion

After sometime we can see the workflow has completed:

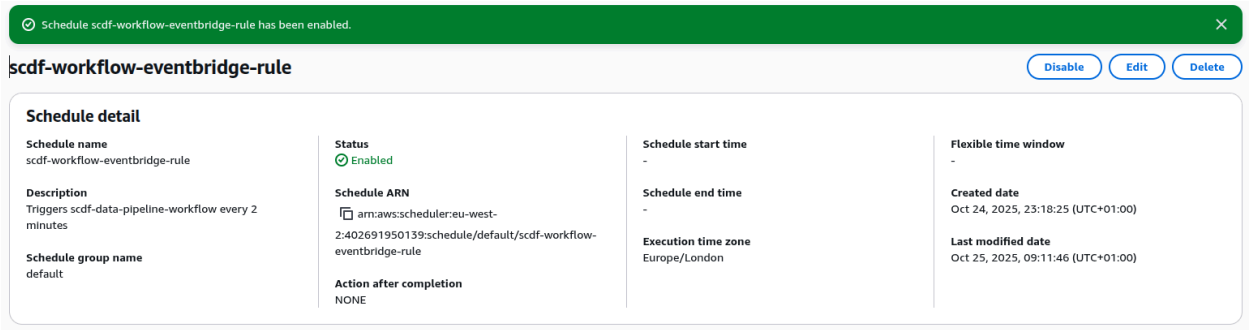


All the glue jobs have succeeded as well:

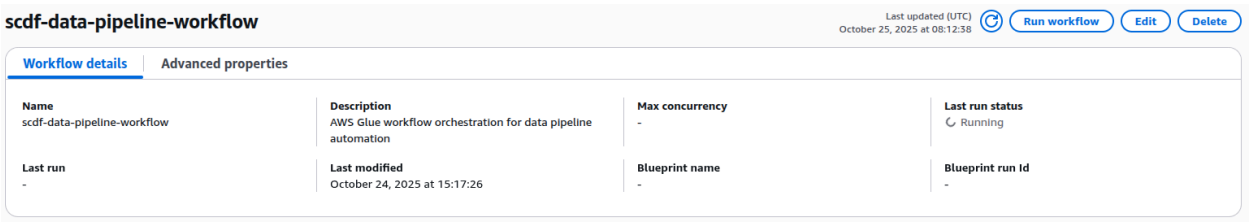


2.9 Integrating Final Workflow with EventBridge

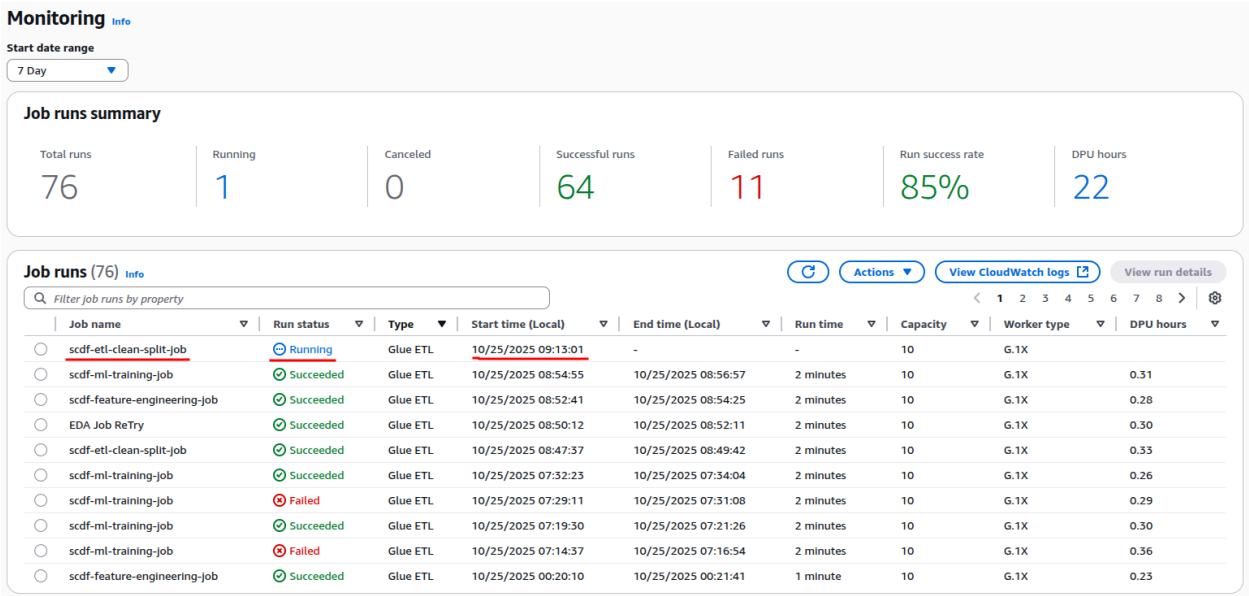
We have just run our final workflow manually and made sure it is running successfully. Next we have to run the workflow. So we will Enable our Eventbridge and see how that goes.



As can be seen from the screenshot the EventBridge schedule has started. Also we can see the workflow has been actuated by the EventBridge schedule:

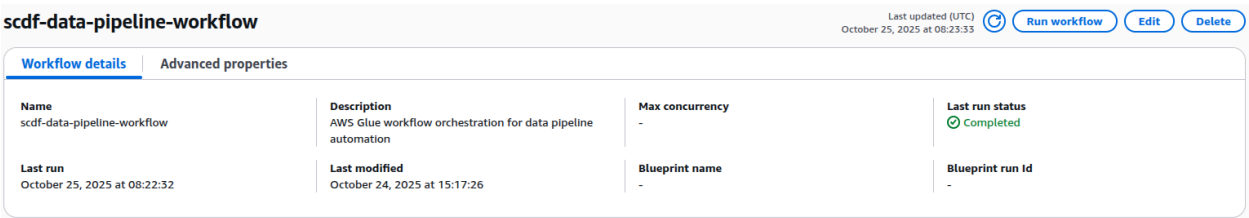


The status of the workflow is Running. The split and clean job has just started:

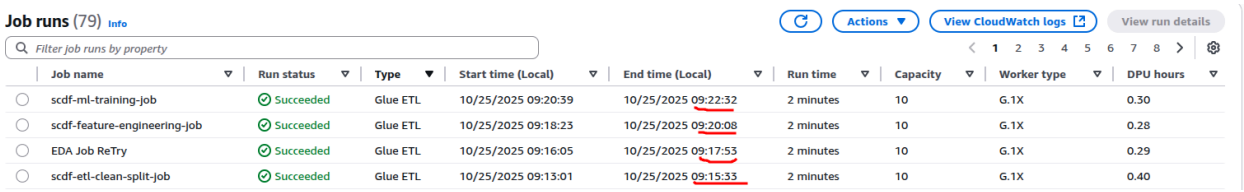


We need to wait till the workflow finishes and comes to a complete state.

After waiting for sometime we can see our workflow which was actuated by the EventBridge schedule has Completed:



All the Glue jobs have succeeded as well:



So now we have a full end to end Automated ML pipeline which is fully functional.

3 API Access

Retrieving Pipeline Status via AWS APIs

The next stage of the project focused on exposing internal pipeline details through an API-driven interface, allowing authorised users to query the operational status of both data and machine-learning pipelines. This capability eliminates the need for users to manually inspect AWS services such as Glue or S3 while still maintaining full transparency and observability.

The design objective was to demonstrate how a cloud-native solution can offer programmatic visibility into system health and performance by leveraging AWS APIs. Through this approach, stakeholders can validate data ingestion, model training, and ETL workflows in near real time, ensuring traceability and accountability across the pipeline lifecycle.

Retrieve Key Application Details Using AWS APIs

To achieve this, a Lambda function named scdf-status-check-api-access was developed to act as the central API access layer for the solution. This function was implemented in Python using the boto3 SDK, allowing seamless interaction with AWS Glue, Amazon S3, and Amazon CloudWatch. Each of these services contributes a unique aspect of pipeline observability.

Source Code

The source implementation for this Lambda function is hosted in the public GitHub repository:

API Driven Cloud Nation Solutions

The script, named scdf_status_check.py, contains the complete Lambda handler logic, including:

- AWS Glue job discovery and run status retrieval
- Amazon S3 artefact verification
- CloudWatch metrics integration (extensible)
- Structured console output for verification

The code was developed and tested within the AWS Lambda environment (Python 3.13 runtime), following least-privilege access principles and modular design for clarity and scalability.

Code Walkthrough and Output Verification

The Lambda code is organised into clearly defined functional segments, each corresponding to a specific AWS service integration. This section explains the implementation logic and verifies functionality using actual execution logs.

Lambda Initialisation

Upon invocation, the Lambda function begins by initialising **boto3 clients** for the required AWS services — namely AWS Glue and Amazon S3:

```
glue = boto3.client('glue')
s3 = boto3.client('s3')
```

boto3 is the **official AWS SDK for Python** and is the most appropriate choice for this implementation because it is **natively supported within the AWS Lambda runtime environment**. It provides a high-level, object-oriented API for interacting with AWS services, eliminating the need for external dependencies or REST calls.

By using boto3:

- The Lambda can securely authenticate through its **execution role**, without requiring static credentials.
- All calls to Glue, S3, and CloudWatch are handled through **fully managed, retry-safe API sessions**.

- Data can be retrieved and formatted within the same Python environment, ensuring minimal latency and consistent error handling.

Log Verification

A descriptive header is then printed to mark the start of execution:

```
===== API Driven Cloud-Native Solution - Verification Output =====
```

Automatic Discovery of AWS Glue Jobs

Once the clients are initialised, the Lambda proceeds to automatically **discover all Glue jobs** available in the account. This step eliminates the need for hardcoded job names, ensuring that any newly created or updated jobs are automatically included in the monitoring scope.

Implementation snippet:

```
jobs_response = glue.get_jobs(MaxResults=20)
job_names = [job['Name'] for job in jobs_response['Jobs']]
print(f"Discovered {len(job_names)} Glue jobs in this account.\n")
```

The function calls the get_jobs() API, which returns metadata for all Glue jobs configured under the same AWS account and region. The resulting job names are extracted into a list and used for subsequent get_job_runs() queries.

This method demonstrates *dynamic pipeline introspection* — a key attribute of cloud-native automation. It allows the system to adjust automatically as new ETL or machine-learning jobs are deployed without requiring code changes.

Log Verification

```
Discovered 6 Glue jobs in this account.
```

This confirms that the Lambda successfully enumerated all six Glue jobs, validating the correctness of IAM permissions and boto3 API integration.

Retrieving Four Key Application Details

For each discovered Glue job, the function queries the get_job_runs() API to fetch its most recent execution details. This provides visibility into real-time operational status and historical performance.

Core implementation logic:

```
runs = glue.get_job_runs(JobName=job, MaxResults=1)
if runs.get('JobRuns'):
    last_run = runs['JobRuns'][0]
    job_details.append({
        "Job Name": job,
        "Status": last_run.get('JobRunState', 'N/A'),
        "Started On": str(last_run.get('StartedOn', 'N/A')),
        "Execution Time (s)": last_run.get('ExecutionTime', 'N/A')
    })
```

From this metadata, four essential **application-level details** are extracted:

1. **Job Name** – identifies the Glue pipeline component.
2. **Status** – reflects its current or most recent state (*SUCCEEDED*, *FAILED*, or *RUNNING*).
3. **Started On** – timestamp marking the beginning of the run.
4. **Execution Duration (seconds)** – total runtime, used to assess efficiency and job health.

To make verification straightforward, Lambda formats these results into a **tabular log output**, printed directly to CloudWatch.

Log Verification

```
=== Verification Table: Four Application Details Retrieved via AWS APIs ===
Job Name                                Status       Started On                  Exec Time (s)
-------------------------------------------------------------------------------------------
EDA Job ReTry                           SUCCEEDED    2025-10-20 05:45:58.693000           107
EDA_ExportToCSV_Job                     SUCCEEDED    2025-10-19 16:30:47.197000           151
EDA_GlueJob                             SUCCEEDED    2025-10-19 17:31:35.547000           135
scdf-etl-clean-split-job                SUCCEEDED    2025-10-19 10:38:51.851000           149
scdf-feature-engineering-job            SUCCEEDED    2025-10-20 06:25:24.627000            72
scdf-ml-training-job                    SUCCEEDED    2025-10-21 06:28:40.961000           115
-------------------------------------------------------------------------------------------
```

Verification of Execution Performance

Each Lambda invocation concludes with a summary record in CloudWatch, confirming total execution time, memory usage, and runtime environment:

```
REPORT RequestId: eda9626d-5266-44c2-a0db-7256a8867355
Duration: 1202.82 ms | Billed Duration: 1203 ms | Memory Size: 128 MB | Max
Memory Used: 97 MB
```

Analysis

- The function completed in ~1.2 seconds, well within the configured 30-second timeout.
- Memory utilisation remained under 100 MB, demonstrating efficient handling of AWS API calls.
- The runtime environment (python:3.13.v64) aligns with the latest AWS Lambda execution standards.

This confirms Lambda's stability, efficiency, and compliance with best practices for lightweight, API-driven observability layers.

Appendix

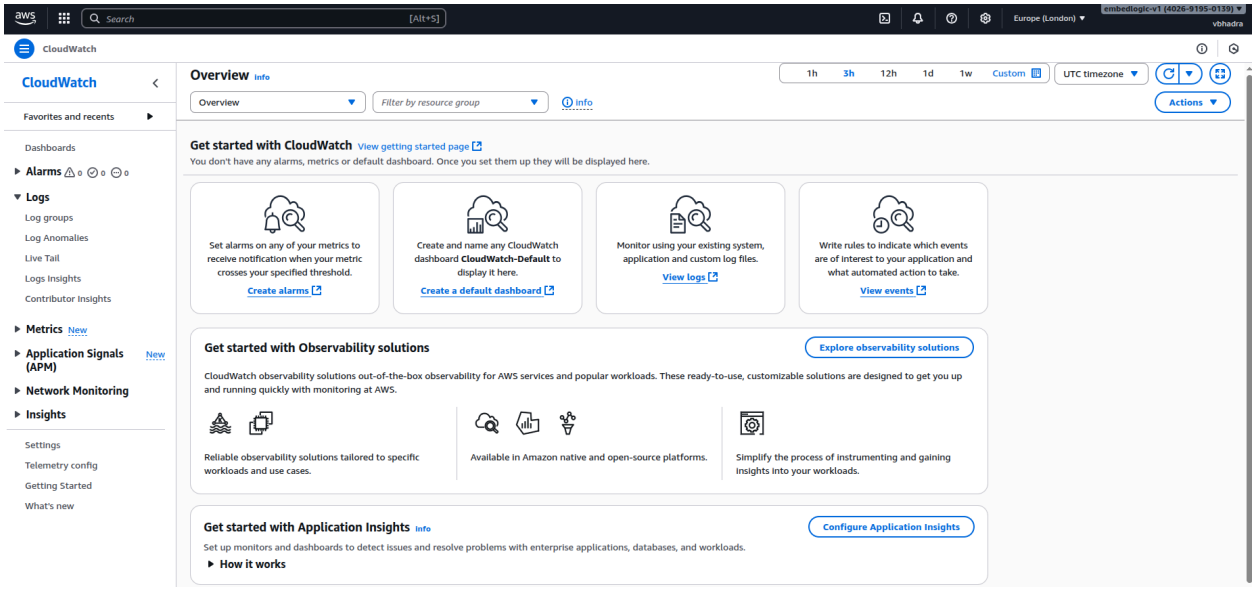
Full Source Code in Github

The full source code for this project can be found at the following public repository which was created for the assignment purposes:

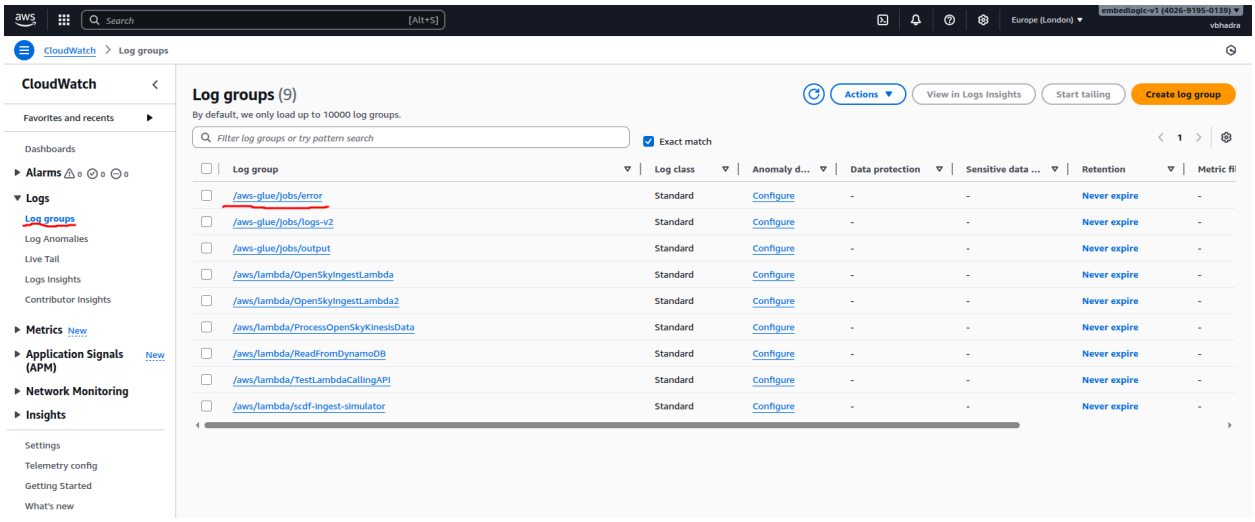
[API Driven Cloud Native Solution Assignment I]

How to check CloudWatch Logs in AWS?

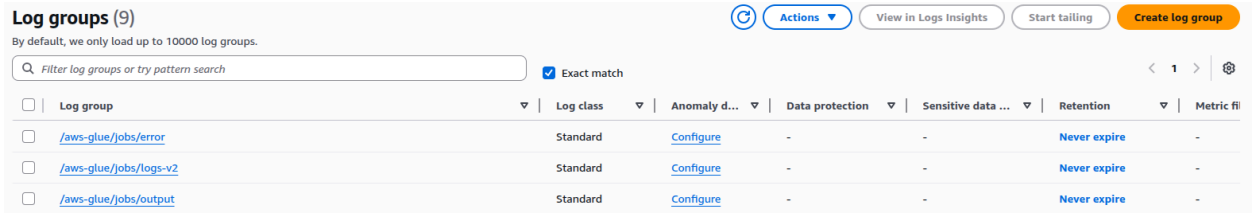
Go to CloudWatch.



On the left hand pane, click **Log Groups**.



Under the Log Groups there are three different type of folders called */error, */logs-v2 and */output:



Go inside /aws-glue/jobs/output and you will see all the logs generated and arranged in date and time of the run: