**Credit Card Prediction ETL Project**

**Dataset Source:**  
[Credit Card Approval Prediction - Kaggle Dataset](https://www.kaggle.com/datasets/rikdifos/credit-card-approval-prediction)

**Objective:**

This project aims to understand how credit card applications are selected based on various demographics of an individual. The dataset contains features related to applicant’s personal details and their application status (approved or rejected). This project involves setting up a secure and scalable pipeline for data extraction, transformation, and analysis, followed by visualizing the insights using Tableau.

**Steps Involved:**

**Step 1: Data Storage in Google Cloud Platform (GCP) Bucket**

1. **GCP Bucket Setup:**
   * Create a Google Cloud Platform (GCP) storage bucket to hold our raw dataset. This is where the dataset will be stored for later processing. This bucket will act as the data source, simulating an on-premise or external data storage location
2. **Service Account & Credentials:**
   * To interact with GCP resources (like the storage bucket) programmatically, a service account needs to be created in the GCP Console.
     + Go to GCP Console → **IAM & Admin** → **Service Accounts** → **Create Service Account.**
     + Generate a **JSON** key file after creating the service account
3. **AWS Secret Manager:**
   * Store the generated GCP service account JSON credentials in **AWS Secrets Manager** to securely manage the credentials.
     + Create a new secret in AWS Secrets Manager, storing the JSON key file content.
     + Use AWS SDK(boto3) or the Secrets Manager API to retrieve these credentials securely during runtime without exposing them in the code.

**Step 2: ETL Pipeline Development**

1. **Infrastructure Setup:**
   * Launch an **EC2 instance** (t3.medium) on AWS to run your ETL pipeline.
   * Install necessary dependencies like **Apache Airflow** for workflow orchestration and **AWS SDK** for S3 interaction.
   * Ensure the EC2 instance has necessary IAM permissions to interact with **AWS S3** and **AWS Secrets Manager**.
2. **Apache Airflow Setup:**
   * Apache Airflow will be used to manage and schedule the ETL workflow.
     + Install Airflow and other necessary packages:

pip install apache-airflow boto3

* + - Set up **Airflow DAG** (Directed Acyclic Graph) for orchestrating the ETL pipeline. The DAG will consist of the following tasks

1. **ETL Pipeline Steps:**

**Extract Data:**

* + Airflow will be used to extract data from the GCP storage bucket.
    - Use google-cloud-storage Python SDK to interact with the GCP bucket.
    - Download the raw CSV data file from the GCP bucket to the EC2 instance.

**Transform Data:**

* + Use **Pandas** to clean and preprocess the dataset:
    - Convert data types appropriately (e.g., ensure numerical columns are in float/int format).
    - Handle **missing or null values** by either filling them with the mean or dropping rows/columns based on the business rules.
    - Validate the data by performing basic checks (e.g., ensuring no duplicates, checking for correct data types, etc.).

**Load Data:**

* + After transformation, use the **S3 Hook** to upload the cleaned data to an AWS **S3 bucket**.
    - The S3 bucket acts as the target storage for further analysis and visualization.

1. **DAG Configuration:**
   * Set the schedule for the DAG (e.g., run every day or weekly depending on the frequency of new data).

**Airflow Image:**

**A screenshot of a computer

Description automatically generated**

**Example DAG code:**

1. from airflow import DAG  
   from airflow.providers.google.cloud.transfers.local\_to\_gcs import LocalFilesystemToGCSOperator  
   from airflow.providers.amazon.aws.hooks.s3 import S3Hook  
   from airflow.operators.python\_operator import PythonOperator  
   from airflow.utils.dates import days\_ago  
   import pandas as pd  
   import boto3  
   import json  
   import os  
   import logging  
   from google.cloud import storage  
     
   # Configuration  
   AWS\_SECRET\_NAME = "gcp\_service\_account\_secret"  
   AWS\_REGION\_NAME = "us-east-1"  
   GCS\_BUCKET\_NAME = "data\_source\_project"  
   S3\_FINAL\_BUCKET = "mysitebucketuta"  
   GCS\_SOURCE\_FILE = "Application\_Data.csv"  
   S3\_OUTPUT\_LOCATION = f"s3://{S3\_FINAL\_BUCKET}/transformed\_data/"  
   DATABASE\_NAME = "credit\_card\_approval\_status"  
   TABLE\_NAME = "applicant\_details"  
   ATHENA\_QUERY\_OUTPUT = f"s3://{S3\_FINAL\_BUCKET}/athena-query-results/"  
     
   # Default DAG arguments  
   default\_args = {  
    'owner': 'airflow',  
    'start\_date': days\_ago(1),  
    'retries': 1,  
   }  
     
     
   # Function to fetch GCP credentials from AWS Secrets Manager  
   def get\_gcp\_credentials():  
    *"""Fetch GCP service account credentials from AWS Secrets Manager."""* client = boto3.client("secretsmanager", region\_name=AWS\_REGION\_NAME)  
    secret\_value = client.get\_secret\_value(SecretId=AWS\_SECRET\_NAME)  
    secret\_data = json.loads(secret\_value['SecretString'])  
     
    # Replace \\n with actual newline characters in the private key  
    if 'private\_key' in secret\_data:  
    secret\_data['private\_key'] = secret\_data['private\_key'].replace('\\n', '\n')  
     
    return secret\_data  
     
     
   # Initialize DAG  
   with DAG(  
    dag\_id="gcs\_to\_s3\_data\_pipeline\_with\_secrets",  
    default\_args=default\_args,  
    schedule\_interval=None,  
    catchup=False,  
    tags=['data\_pipeline', 'GCS', 'S3', 'Athena', 'SecretsManager'],  
   ) as dag:  
    def list\_gcs\_files(\*\*kwargs):  
    *"""List files in the GCS bucket."""* credentials = get\_gcp\_credentials()  
     
    # Write credentials to a temporary file  
    service\_account\_file = "/tmp/gcp\_service\_account.json"  
    with open(service\_account\_file, "w") as f:  
    json.dump(credentials, f)  
     
    # Authenticate and list files in the GCS bucket  
    client = storage.Client.from\_service\_account\_json(service\_account\_file)  
    bucket = client.get\_bucket(GCS\_BUCKET\_NAME)  
    blobs = bucket.list\_blobs()  
    print(f"Files in bucket '{GCS\_BUCKET\_NAME}':")  
    for blob in blobs:  
    print(f"Name: {blob.name}, Size: {blob.size} bytes, Updated: {blob.updated}")  
    # Download the file to /tmp with the same name as the GCS object  
    download\_path = os.path.join("/tmp", blob.name)  
     
    # Ensure that the directory exists before downloading  
    os.makedirs(os.path.dirname(download\_path), exist\_ok=True)  
     
    blob.download\_to\_filename(download\_path)  
    print(f"Downloaded {blob.name} to {download\_path}")  
     
     
    def transform\_data(\*\*kwargs):  
    *"""Reads, transforms data, and writes back to a temporary file."""* # Path to temporary file in MWAA environment  
    tmp\_file\_path = f"/tmp/{GCS\_SOURCE\_FILE}"  
     
    # Read CSV file into a DataFrame  
    df = pd.read\_csv(tmp\_file\_path)  
     
    # Strip whitespace and handle nulls  
    df = df.applymap(lambda x: x.strip() if isinstance(x, str) else x)  
     
    # Infer data types  
    data\_types = {col: str(df[col].dtype) for col in df.columns}  
     
    # Save transformed data  
    transformed\_file\_path = "/tmp/transformed\_data.csv"  
    df.to\_csv(transformed\_file\_path, index=False)  
     
    # Pass file path to next task  
    kwargs['ti'].xcom\_push(key='transformed\_file\_path', value=transformed\_file\_path)  
     
     
    # Task: List files in GCS bucket  
    list\_files\_task = PythonOperator(  
    task\_id='list\_gcs\_files',  
    python\_callable=list\_gcs\_files,  
    provide\_context=True,  
    )  
     
    # Task: Transform data  
    transform\_data\_task = PythonOperator(  
    task\_id='transform\_data',  
    python\_callable=transform\_data,  
    provide\_context=True,  
    )  
     
     
    # Task: Upload transformed data to S3  
    def upload\_to\_s3(\*\*kwargs):  
    transformed\_file\_path = kwargs['ti'].xcom\_pull(key='transformed\_file\_path')  
    S3Hook().load\_file(  
    filename=transformed\_file\_path,  
    key="transformed\_data.csv",  
    bucket\_name=S3\_FINAL\_BUCKET,  
    replace=True,  
    )  
     
     
    upload\_to\_s3\_task = PythonOperator(  
    task\_id='upload\_to\_s3',  
    python\_callable=upload\_to\_s3,  
    provide\_context=True,  
    )  
     
     
    # Create Athena Db and Table  
     
    def athena\_table\_creation(\*\*kwargs):  
    *"""Creates an Athena database and table for the dataset."""* transformed\_file\_path = kwargs['ti'].xcom\_pull(key='transformed\_file\_path')  
    df = pd.read\_csv(transformed\_file\_path)  
     
    def map\_dtype\_to\_athena(dtype):  
    *"""Maps Pandas dtypes to Athena data types."""* if "int" in dtype:  
    return "BIGINT"  
    elif "float" in dtype:  
    return "DOUBLE"  
    elif "object" in dtype:  
    return "STRING"  
    elif "datetime" in dtype:  
    return "TIMESTAMP"  
    elif "bool" in dtype:  
    return "BOOLEAN"  
    else:  
    return "STRING"  
     
    # Get column names and their Athena-compatible data types  
    columns\_and\_types = {col: map\_dtype\_to\_athena(str(dtype)) for col, dtype in zip(df.columns, df.dtypes)}  
     
    # Athena table schema  
    columns\_schema = ",\n".join([f"{col} {dtype}" for col, dtype in columns\_and\_types.items()])  
     
    # Create database query (if the database doesn't exist)  
    create\_db\_query = f"""  
    CREATE DATABASE IF NOT EXISTS {DATABASE\_NAME};  
    """  
     
    # Create table query  
    create\_table\_query = f"""  
    CREATE EXTERNAL TABLE IF NOT EXISTS {DATABASE\_NAME}.{TABLE\_NAME} (  
    {columns\_schema}  
    )  
    ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'  
    WITH SERDEPROPERTIES (  
    "separatorChar" = ",",  
    "quoteChar" = "\""  
    )  
    STORED AS TEXTFILE  
    LOCATION '{S3\_OUTPUT\_LOCATION}'  
    TBLPROPERTIES ("skip.header.line.count" = "1");  
    """  
     
    # Initialize logging  
    logging.basicConfig(level=logging.INFO)  
     
    # Run Athena query to create the database  
    athena\_client = boto3.client('athena')  
     
    try:  
    # First, create the database  
    logging.info("Executing query to create database...")  
    db\_response = athena\_client.start\_query\_execution(  
    QueryString=create\_db\_query,  
    ResultConfiguration={'OutputLocation': ATHENA\_QUERY\_OUTPUT},  
    )  
    logging.info(f"Database creation query started, execution ID: {db\_response['QueryExecutionId']}")  
     
    # Then, create the table  
    logging.info("Executing query to create table...")  
    table\_response = athena\_client.start\_query\_execution(  
    QueryString=create\_table\_query,  
    ResultConfiguration={'OutputLocation': ATHENA\_QUERY\_OUTPUT},  
    )  
    logging.info(f"Table creation query started, execution ID: {table\_response['QueryExecutionId']}")  
     
    except Exception as e:  
    logging.error(f"Error occurred while creating database or table: {str(e)}")  
     
     
    # Task: Create Athena table  
    create\_athena\_table = PythonOperator(  
    task\_id='create\_athena\_table',  
    python\_callable=athena\_table\_creation,  
    provide\_context=True,  
    )  
     
    # Task dependencies  
    list\_files\_task >> transform\_data\_task >> upload\_to\_s3\_task >> create\_athena\_table

**Step 3: Data Analysis Using Athena**

1. **Athena Table Creation:**

The Athena table and database were dynamically created using Airflow Python code by reading the column data from a CSV file.

1. **SQL Queries for Analysis:**
   * Perform analysis using SQL queries, such as:
     + Approval Rate by Applicant Demographics
     + Approval Rate by Gender
     + Count of Applicants by Job Title
   * Example SQL:

**SELECT**

**Applicant\_Gender,**

**Family\_Status,**

**COUNT(\*) AS Total\_Applicants,**

**SUM(CASE WHEN Status = 1 THEN 1 ELSE 0 END) AS Approved,**

**ROUND((SUM(CASE WHEN Status = 1 THEN 1 ELSE 0 END) \* 1.0 / COUNT(\*)) \* 100, 2) AS Approval\_Rate**

**FROM**

**credit\_card\_approval\_status.applicant\_details**

**GROUP BY**

**Applicant\_Gender, Family\_Status;**

**Athena Image:**

**A screenshot of a computer

Description automatically generated**

**Step 4: Data Visualization Using Tableau**

1. **Connect Tableau to Athena:**
   * Install the **Athena ODBC Driver** on our local machine (Tableau Desktop).
   * Copy the driver to the Tableau Driver folder to ensure Tableau can communicate with Athena.
   * In Tableau, choose **Athena** as the data source and provide the connection details (AWS region, access keys).
2. **Data Import:**
   * Once connected to Athena, use the **Custom SQL** option in Tableau to run the same sql queries.
3. **Visualizing Data:**
   * Create Tableau sheets that visualize:
     + **Demographic Analysis**: How application approval rates vary based on gender, age, education, and income.
     + **Approval Rates**: Comparison of application approval by marital status and education level.
     + **Time Analysis**: Trends in approvals/rejections over time.
   * Combine these visualizations into a dashboard for interactive exploration.
4. **Image:**

**A screen shot of a computer

Description automatically generated**

**Deliverables:**

1. **ETL Pipeline in Apache Airflow:**
   * Fully automated pipeline for extracting, transforming, and loading data into S3.
2. **Athena Database and Table:**
   * Database and table automatically created by Airflow for querying and analysis.
3. **SQL Queries for Analysis:**
   * SQL queries for performing demographic analysis, approval trends, etc.
4. **Tableau Dashboards:**
   * Interactive visualizations, providing insights into the dataset, including demographic trends and approval patterns.