Lead Conversion Prediction

- 4.2. The Ingestion Pipeline: S3 to Redshift
    - 4.2.1. Creating the Glue Connection to Redshift
    - 4.2.2. The Glue Crawler for Automatic Schema Discovery
    - 4.2.3. Building the Visual ETL Job (s3ToRedshift)

---

5. ML Development in Amazon SageMaker

- 5.1. Setting up the SageMaker Environment
    - 5.1.1. Creating a SageMaker Domain
    - 5.1.2. Launching SageMaker Studio & JupyterLab
- 5.2. ML Development Workflow
    - 5.2.1. Data Ingestion from Redshift
    - 5.2.2. Data Exploration, Cleaning, and Feature Engineering
    - 5.2.3. Building the Preprocessing Pipeline
    - 5.2.4. Data Preprocessing and Class Balancing with SMOTE
    - 5.2.5. Model Training, Logging, and Explanation with SHAP
    - 5.2.6. Model Selection and Registration

---

6. MLOps, Deployment & Monitoring

- 6.1. Experiment Tracking with MLflow
    - 6.1.1. Setting Up the MLflow Tracking Server
    - 6.1.2. Integrating MLflow into the Training Pipeline
- 6.2. Model Registration and Lifecycle Management
    - 6.2.1. MLflow Model Registry for Versioning and Staging
- 6.3. Model Deployment and Serving
    - 6.3.1. Loading the Production Model from the Registry
    - 6.3.2. Real-time Prediction with Flask and Ngrok
- 6.4. Drift Analysis with Evidently & MLflow

---

7. Full Automation with MWAA (Managed Workflows for Apache Airflow)

- 7.1. Orchestrating the MLOps Lifecycle
- 7.2. Setting up the MWAA Environment
- 7.3. The Automated Retraining DAG (Directed Acyclic Graph)
    - 7.3.1. Preparing and Uploading the DAG File
    - 7.3.2. Workflow Logic and Conditional Retraining
    - 7.3.3. Monitoring DAG Runs in the Airflow UI

---

8 .Technology & Libraries Stack

---

9. Code Repository & Resources

---

1. Introduction

1.1 Problem Statement

- Companies get a lot of leads every day, but not all of them turn into customers. Right now, sales teams often guess which leads are good, which wastes time and effort.

1.2 Project Goal

To develop a robust, explainable machine learning system that accurately predicts the likelihood of lead conversion, enabling sales and marketing teams to prioritize high-potential leads and maximize conversion rates, efficiency, and return on investment.

1.3 Project Objectives

1. **Maximize Conversion Rate**
   Use predictive analytics to identify and prioritize leads with the highest probability of conversion, thereby increasing the proportion of leads that become customers.
2. **Minimize Resource Wastage**
   Reduce time and effort spent on low-potential leads by automating the lead scoring process, allowing sales teams to focus on leads most likely to convert.

3. **Maximize Sales & Marketing ROI**
   Optimize allocation of sales and marketing resources towards leads offering the greatest revenue potential, improving overall campaign effectiveness.
4. **Enable Data-Driven Decision-Making**
   Provide an interpretable, reliable lead scoring model, supported with explainability tools (e.g., SHAP), so business users understand and trust the predictions.
5. **Maintain Consistent Model Performance**
   Continuously monitor for model/data drift and retrain as necessary to ensure sustained accuracy and business impact as lead profiles and market conditions evolve.

1.4. Success Criteria

1.4.1 Bussiness Success Criteria

1. **Lead-to-Sale Conversion Rate ≥ 5%**
   This means at least 5% of generated leads convert into paying customers, outperforming typical industry averages of 2–5%.
2. **Reduction in Cost per Conversion by 10–20%**
   Lower marketing and sales spend per converted lead within the first 6–12 months, demonstrating improved efficiency and return on investment.

1.4.2 MLSuccess Criteria

1. ROC-AUC > 0.90, F1-score > 0.70.
2. Drift detection/alert within 7 days of data change.
3. Model results interpretable by SHAP feature importance.
4. Serve scoring API responses <500ms.

1.5 Data Description

Feature Name	Explanation
id	Unique identifier for each lead
Lead Origin	How the lead originally entered the system (e.g., Landing Page, API, etc.)
Lead Source	Source/channel through which the lead was acquired (Google, Direct

Feature Name	Explanation
	Traffic, etc.)
Do Not Email	Indicates if the lead opted out of email communications (Yes/No)
Do Not Call	Lead’s preference for receiving calls (Yes/No)
Converted	**Target:** Whether the lead converted (1 = Yes, 0 = No)
TotalVisits	Number of times the lead visited the website
Total Time Spent on Website	Cumulative minutes the lead spent browsing
Page Views Per Visit	Average number of pages viewed per session
Last Activity	Last recorded interaction by the lead
Country	Lead's country
Specialization	Area of interest or specialty indicated by the lead
How did you hear about X	How the lead heard about the organization
What is your current occupation	Current profession of the lead
What matters most to you in choosing a course	Lead's priority when selecting a program (e.g., "Better Career Prospects")
Search	Did the lead use the website search feature? (Yes/No)

Feature Name	Explanation
Magazine	Did the lead find out through a magazine? (Yes/No)
Newspaper Article	Whether the lead read about it in a newspaper article (Yes/No)
X Education Forums	Interaction through education forum (Yes/No)
Newspaper	Other newspaper-based engagement (Yes/No)
Digital Advertisement	Clicks or engagement via digital ads (Yes/No)
Through Recommendations	Is the lead referred by someone else? (Yes/No)
Receive More Updates About Our Courses	Whether the lead wants course updates (Yes/No)
Tags	Categorized tag based on lead's status or behavior (e.g., "Interested in XYZ
Lead Quality	Human/categorical assessment of lead quality (e.g., "High", "Low")
Update me on Supply Chain Content	Subscribed to supply chain content updates? (Yes/No)
Get updates on DM Content	Subscribed to digital marketing content updates? (Yes/No)
City	City of the lead
Asymmetrique Activity Index	Behavioral activity score calculated by a third-party scoring tool
Asymmetrique Profile Index	Profile matching score (alignment with target persona)

Feature Name	Explanation
Asymmetrique Activity Score	Detailed activity engagement score
Asymmetrique Profile Score	Detailed profile fit score
I agree to pay the amount through cheque	Consent to pay by cheque (Yes/No)
A free copy of Mastering The Interview	Did the lead request a free eBook/resource? (Yes/No)
Last Notable Activity	The most significant recent action taken by the lead

---

2.AWS ML Pipeline Architecture

2.1. Architectural Overview

The system is designed as a modular, event-driven architecture that ensures scalability, security, and maintainability.

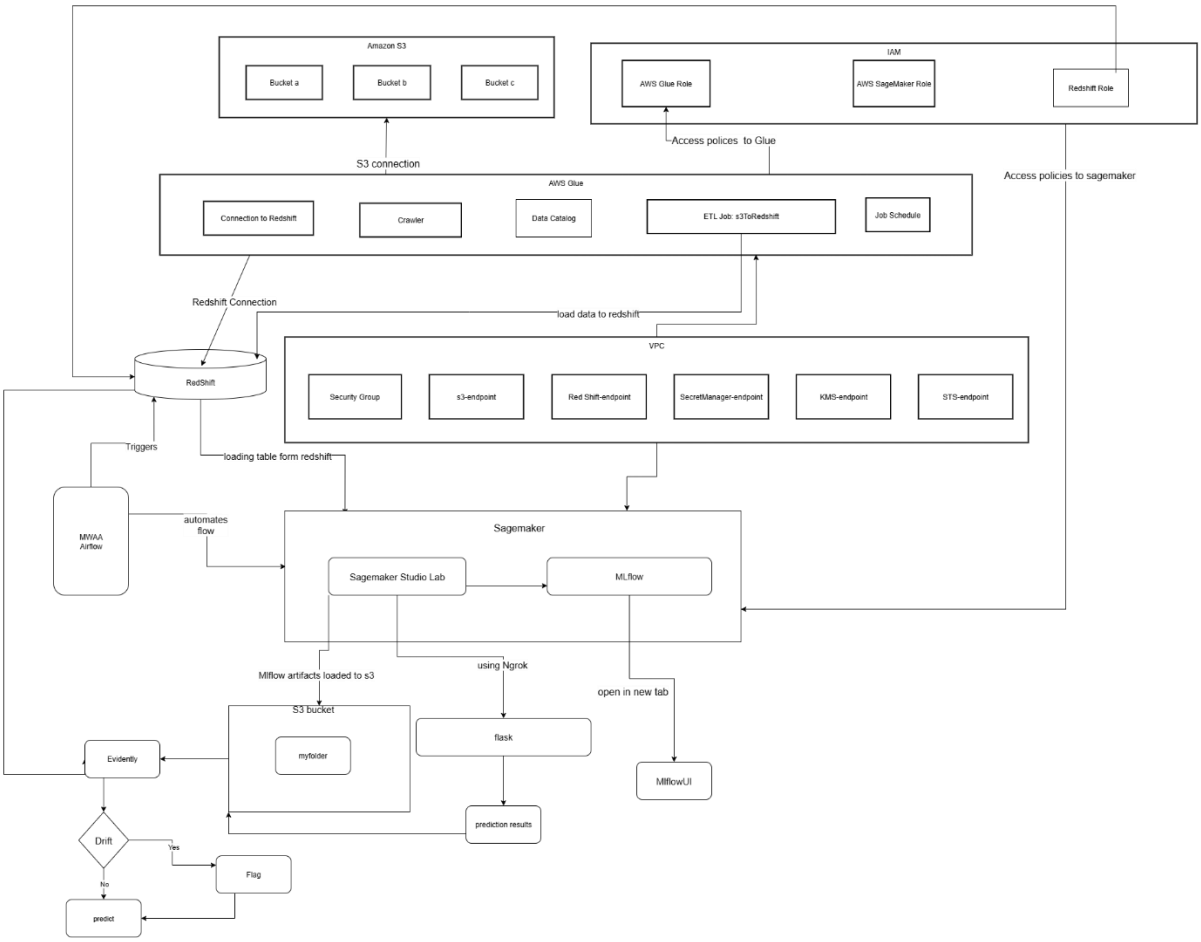
- **Left (Ingestion):** Raw data lands in an Amazon S3 "raw" bucket.
- **Middle (ETL & Warehousing):** AWS Glue components (Crawler, ETL Job) are triggered. The Crawler populates the Glue Data Catalog. The ETL job reads from the raw S3 bucket, transforms the data, and loads it into an Amazon Redshift data warehouse. All services communicate privately within a VPC via Endpoints, secured by an IAM Role and Security Group.
- **Right (Machine Learning):** Amazon SageMaker Studio Lab accesses the clean data from Redshift. It uses MLflow (running on a separate EC2 instance within the VPC) for experiment tracking. Trained model artifacts are saved to a dedicated S3 "artifacts" bucket.

Bottom (Serving & Monitoring): A Flask application, running locally or within SageMaker, loads the production model from the MLflow registry. Ngrok exposes

- this Flask API to the web for testing. MLflow UI provides a dashboard for monitoring.

2.2 Key Component Breakdown

- **Data Storage (Amazon S3):** Serves as the foundational storage layer. Different buckets/prefixes are used for raw-data, processed-data, and model-artifacts, ensuring a clean separation of concerns.
- **IAM (Identity and Access Management):** Provides the security backbone.
    - **AWS Glue Role:** Grants AWS Glue permissions to access S3 data and connect to the Redshift cluster.
    - **AWS SageMaker Role:** Allows SageMaker to read data from Redshift/S3 and write model artifacts back to S3.



- **AWS Glue (ETL Platform):** The serverless data integration engine.
    - **Connection to Redshift:** A pre-configured connector allowing Glue jobs to read from and write to Redshift.

    - **Crawler & Data Catalog:** Automatically scans and catalogs data schemas, making data discoverable and queryable.
    - **ETL Job (s3ToRedshift):** The core job that extracts raw S3 data, cleans it, and loads it into Redshift.
- **Networking (VPC):** Creates a private, isolated network for our resources.
    - **Security Group:** A virtual firewall controlling traffic to VPC endpoints.
    - **Endpoints:** Provide secure, private connections for S3, Redshift, Secrets Manager, KMS, and STS, preventing data exposure to the public internet.
- **Data Warehouse (Amazon Redshift):** The central repository for structured, analytics-ready data. It serves as the single source of truth for the machine learning model.
- **SageMaker (ML Platform):** The integrated development environment for all ML tasks.
    - **SageMaker Studio Lab:** Used for data preparation, interactive development, and model building.
    - **Artifact Management:** All trained models are saved to S3 for persistence and versioning.
- **Serving and Monitoring Stack:**
    - **MLflow:** Integrated for comprehensive experiment tracking, model logging, and lifecycle management via the Model Registry.
    - **Flask:** A lightweight web framework used to create a simple API to serve the model's prediction endpoint.
    - **Ngrok:** A utility to create a secure tunnel, exposing the local Flask API for testing and integration.
    - **MLflowUI:** The web dashboard for tracking, comparing, and visualizing ML experiments.

2.3 End-to-End Data and Model Flow

1. **Raw Data Ingestion:** Raw lead data (.csv) is uploaded to the designated Amazon S3 bucket.
2. **ETL with AWS Glue:** The Glue Crawler scans the S3 bucket and updates the Data Catalog. A scheduled Glue ETL job then extracts the data, transforms it, and loads it into Amazon Redshift.

3. **Redshift Data Access:** Redshift now holds clean, structured tables, ready for machine learning.
4. **Data Access for ML:** Amazon SageMaker securely accesses the Redshift data via VPC endpoints and its IAM role. Training, validation, and test sets are prepared.
5. **ML Development in SageMaker:** Models are built, trained, and evaluated in SageMaker Studio Lab. MLflow tracks all experiments and metrics. The final model artifacts are saved to S3.
6. **Model Logging & Serving:** The best model is registered in the MLflow Model Registry and promoted to "Production." It is then served as an API via a Flask application, exposed for testing with Ngrok.
7. **Feedback Loop & Automation:** Glue schedules ensure continuous data ingestion. The MLOps framework supports systematic retraining and versioning.

---

3. Foundational AWS Infrastructure Setup

3.1 Identity & Access Management (IAM)

The Principle of Least Privilege

Security is paramount. Our architecture adheres to the principle of least privilege, meaning every component is granted only the permissions essential for its designated function. This is achieved by creating highly specific IAM roles that services like Glue and SageMaker assume to perform their tasks, minimizing potential security risks.

3.1.1 Creating the Core Service Roles

We create two primary roles: one for Glue and one for SageMaker. The process below details the creation of the Glue role, which is the more complex of the two.

Console Walkthrough: Creating glueaccessrole

1. **Navigate to IAM:** In the AWS Management Console, go to the IAM service.
2. **Create Role:** Click on **Roles** in the left navigation pane, then click **Create role**.
3. **Select Trusted Entity:**
    - **Trusted entity type:** Select **AWS service**.
    - **Use case:** Choose **Glue** from the dropdown menu. This automatically creates a trust policy allowing the AWS Glue service (glue.amazonaws.com) to assume this role.

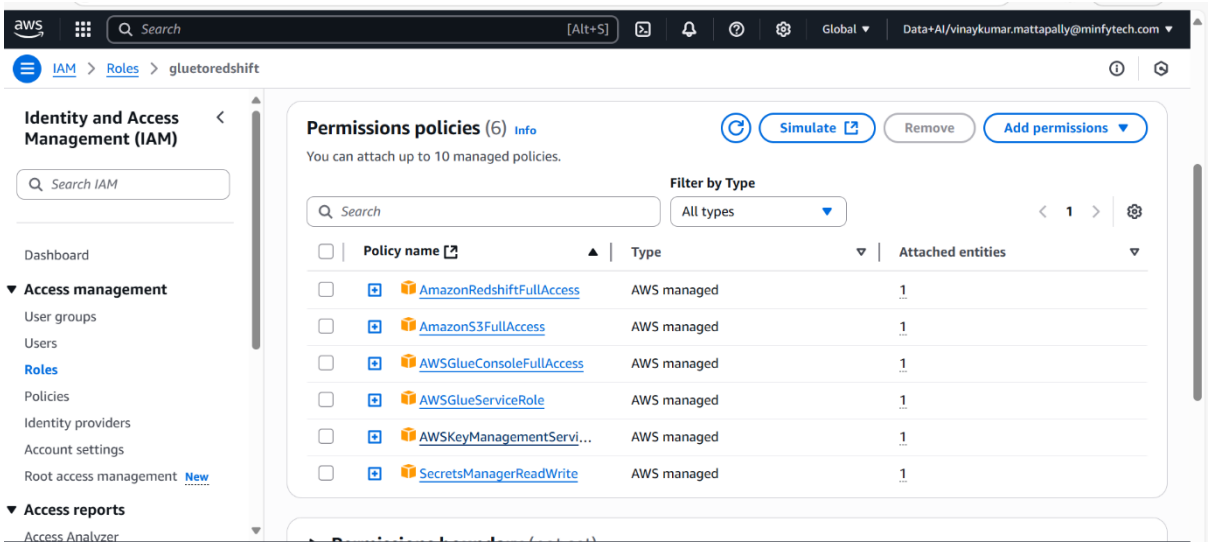
- Click **Next**.

4. **Add Permissions:** On the "Add permissions" page, search for and attach the following AWS managed policies. For a production environment, these should be replaced with more fine-grained, custom policies.
    - AmazonS3FullAccess
    - AmazonRedshiftFullAccess
    - AWSGlueConsoleFullAccess
    - AWSGlueServiceRole
    - SecretsManagerReadWrite
    - AWSKeyManagementServicePowerUser
    - Click **Next**.
5. **Name and Review:**
    - **Role name:** glueaccessrole
    - Review the attached policies and the trust policy. The trust policy JSON should look like this:

JSON

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

- Click **Create role**.



A similar process is followed to create the SageMaker role, selecting **SageMaker** as the use case in Step 3.

3.2 Networking & Security (VPC)

3.2.1 VPC and Subnet Strategy

To ensure all inter-service communication is secure and isolated from the public internet, all resources are deployed within a Virtual Private Cloud (VPC). We utilize the default VPC for this guide, but a custom VPC is recommended for production. The VPC contains multiple subnets across different Availability Zones for high availability.

3.2.2 Creating VPC Endpoints for Secure Communication

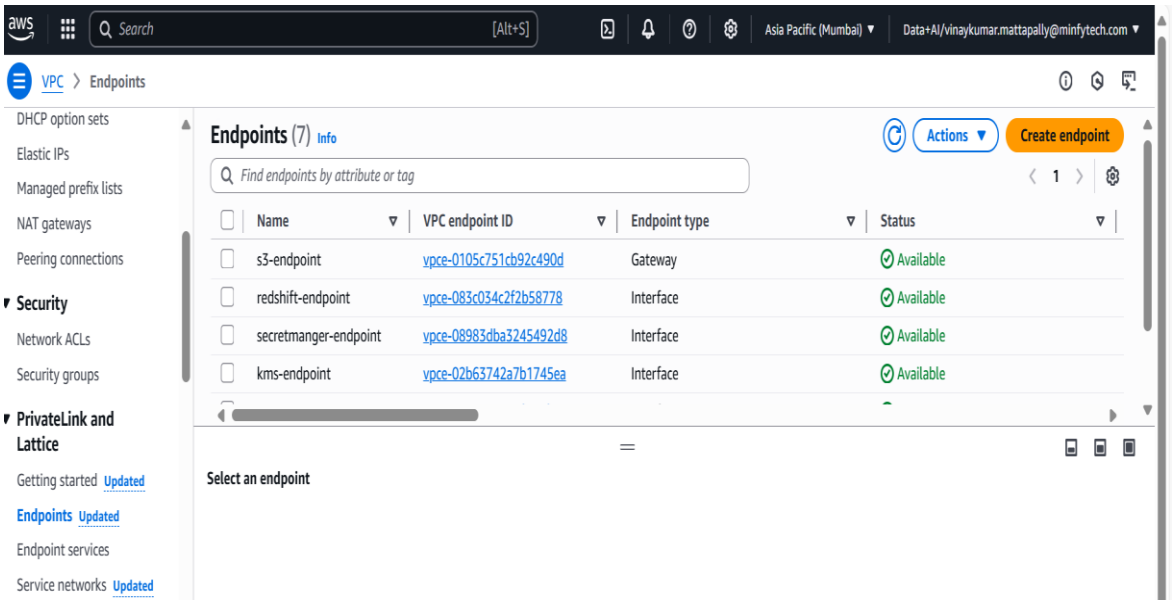
VPC Endpoints allow services within the VPC to communicate with other AWS services as if they were on the same network, without traffic ever leaving the AWS backbone.

Console Walkthrough: Creating Endpoints

1. Navigate to the **VPC** service in the AWS Console.
2. Select **Endpoints** and click **Create endpoint**.
3. **Endpoint 1: S3 (Gateway)**
    - **Service category:** AWS services
    - **Service name:** Search for s3 and select the service with Type: Gateway.
    - **VPC:** Select your target VPC.

- **Route tables:** Select the route tables for the subnets where your resources reside. This adds a route to the table, directing S3 traffic through the gateway.
- Click **Create endpoint**.

4. **Endpoint 2: Redshift (Interface)**
   - Click **Create endpoint** again.
   - **Service category:** AWS services
   - **Service name:** Search for redshift and select the service com.amazonaws.<region>.redshift.
   - **VPC:** Select your target VPC.
   - **Subnets:** Select at least two subnets in different Availability Zones.
   - **Security groups:** Select the security group created in the next section.
   - Click **Create endpoint**.
5. **Endpoints 3, 4, 5: Secrets Manager, KMS, and STS (Interface)**
   - Repeat the process for the following services, selecting the same VPC, subnets, and security group for each:
     - com.amazonaws.<region>.secretsmanager
     - com.amazonaws.<region>.kms
     - com.amazonaws.<region>.sts

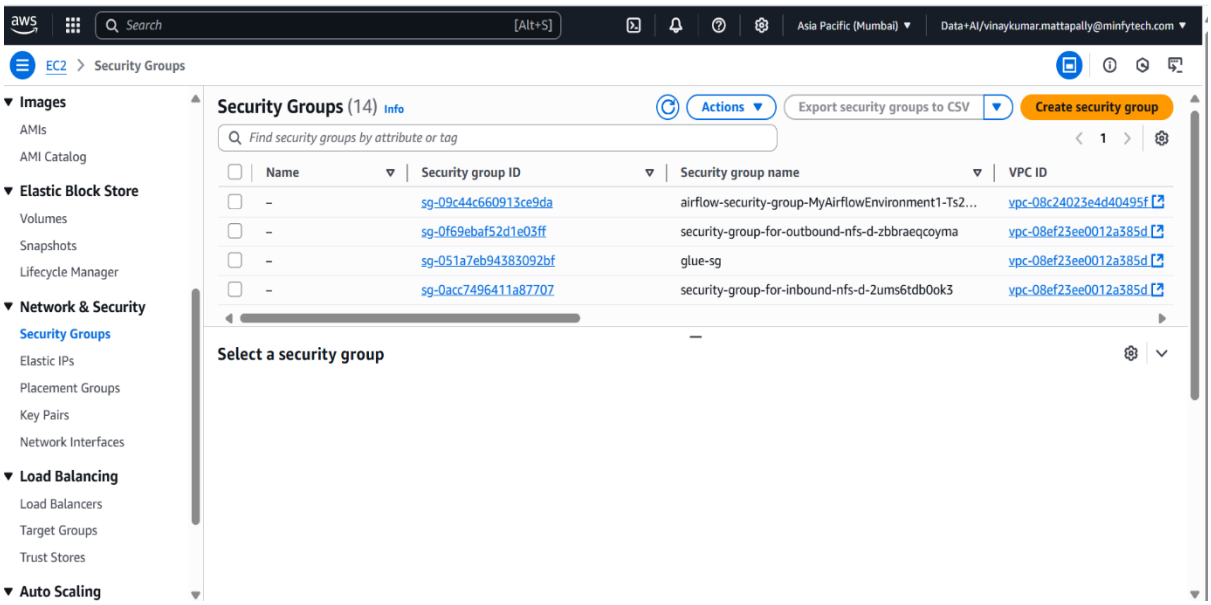


3.2.3 Configuring the Security Group Firewall

A security group acts as a stateful firewall for resources, controlling inbound and outbound traffic.

Console Walkthrough: Configuring the Security Group

1. Navigate to **VPC > Security Groups** and click **Create security group**.
2. **Name and VPC:** Name it gluesg and associate it with your VPC.
3. **Configure Inbound Rules:** Select the new security group and click the **Inbound rules** tab, then **Edit inbound rules**.
    - **Rule 1 (For Redshift Client Access):**
        - **Type:** Redshift
        - **Protocol:** TCP
        - **Port range:** 5439
        - **Source:** My IP (to allow access from your local machine).
    - **Rule 2 (For Internal Communication):**
        - **Type:** All TCP
        - **Protocol:** TCP
        - **Port range:** 0 - 65535
        - **Source:** Custom. Start typing the ID of the gluesg group itself (sg-xxxxxxxx). This self-referencing rule allows all resources within the security group to communicate with each other freely. This is critical for Glue, SageMaker, and Redshift to interact.
4. **Save Rules:** Click **Save rules**. The default outbound rule, which allows all traffic out, is sufficient for this project.



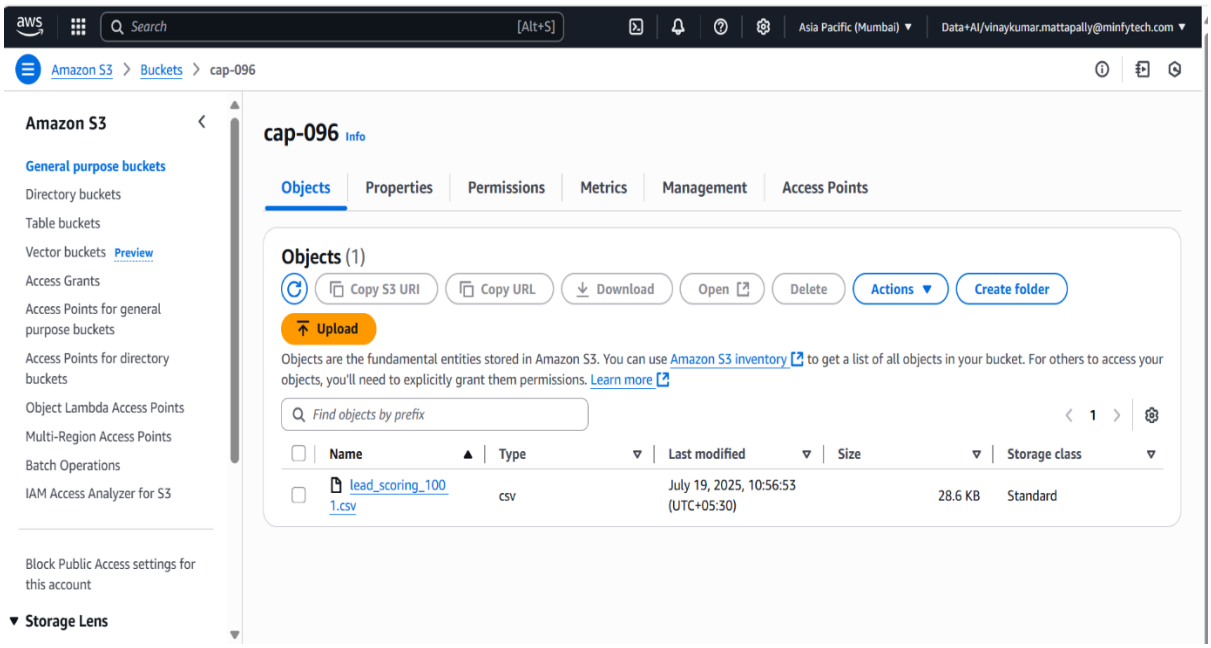
4.Data Engineering & Warehousing

4.1 Data Lake & Warehouse Setup

4.1.1 S3 Bucket Configuration

Amazon S3 is the foundation of our data storage. We use a single bucket with a logical prefix structure to organize data by stage.

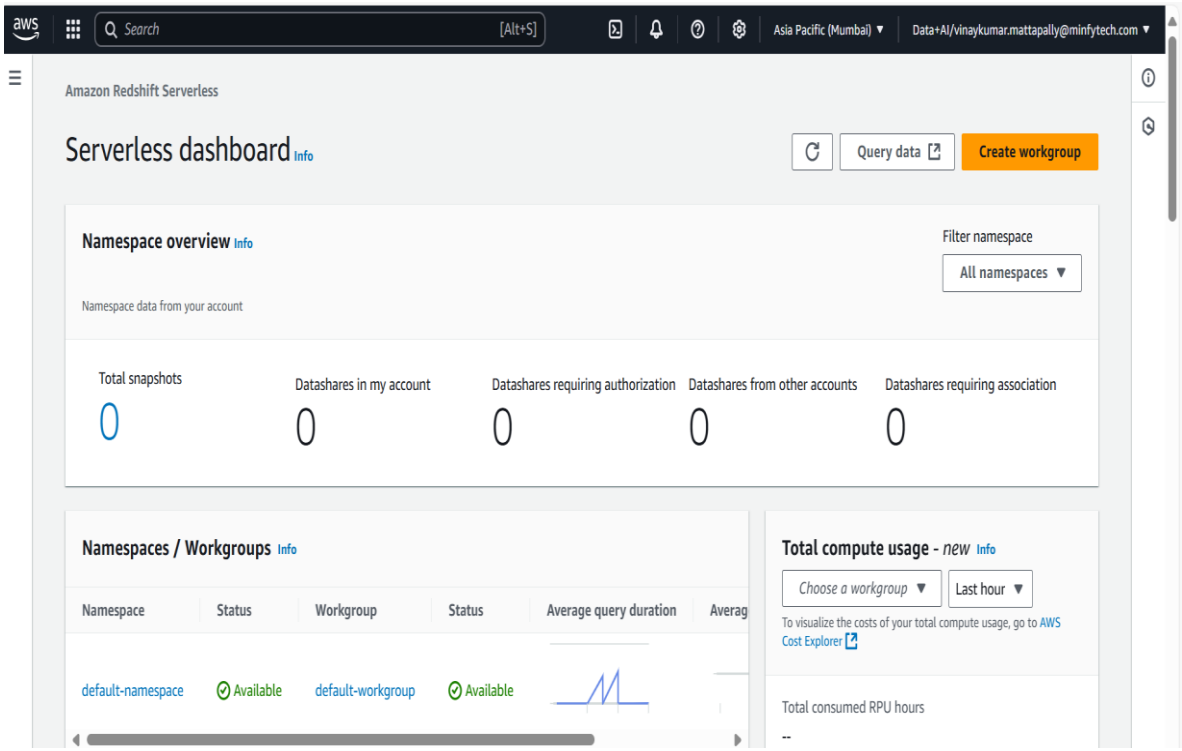
- **Bucket Name:** lead-conversion-project-data
- **Prefixes:**
    - s3://lead-conversion-project-data/raw-data/: For incoming, unaltered lead data.
    - s3://lead-conversion-project-data/processed-data/: For data cleaned and feature-engineered by Glue.
    - s3://lead-conversion-project-data/model-artifacts/: For storing trained model binaries, SHAP plots, and other ML assets.



4.1.2 Amazon Redshift Serverless Provisioning

Redshift Serverless provides a simple, auto-scaling data warehouse.

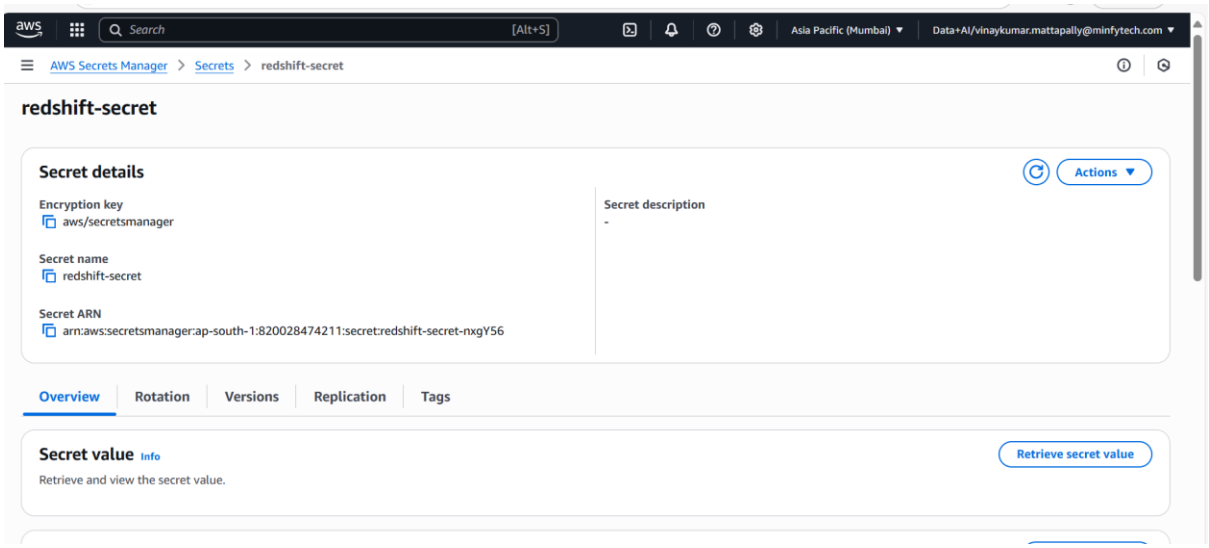
1. Navigate to the **Amazon Redshift** service and select **Try Redshift Serverless**.
2. Configure the workgroup and namespace, accepting the defaults.
3. In the **Network and security** section, associate the workgroup with your VPC and the gluesg security group.
4. Review and create the serverless endpoint. This will take several minutes.



4.1.3 Securing Credentials with AWS Secrets Manager

We avoid hardcoding database credentials by storing them securely in Secrets Manager.

1. In the Redshift console, set a password for the default awsuser.
2. Navigate to **AWS Secrets Manager** and click **Store a new secret**.
3. **Secret type:** Select Credentials for Redshift cluster.
4. Enter the awsuser username and the password you set.
5. **Secret name:** Use a descriptive name like prod/redshift/credentials.
6. Complete the process, disabling automatic rotation for this guide. Note the ARN of the created secret.



4.2 The Ingestion Pipeline: S3 to Redshift

ETL with AWS Glue: Overview

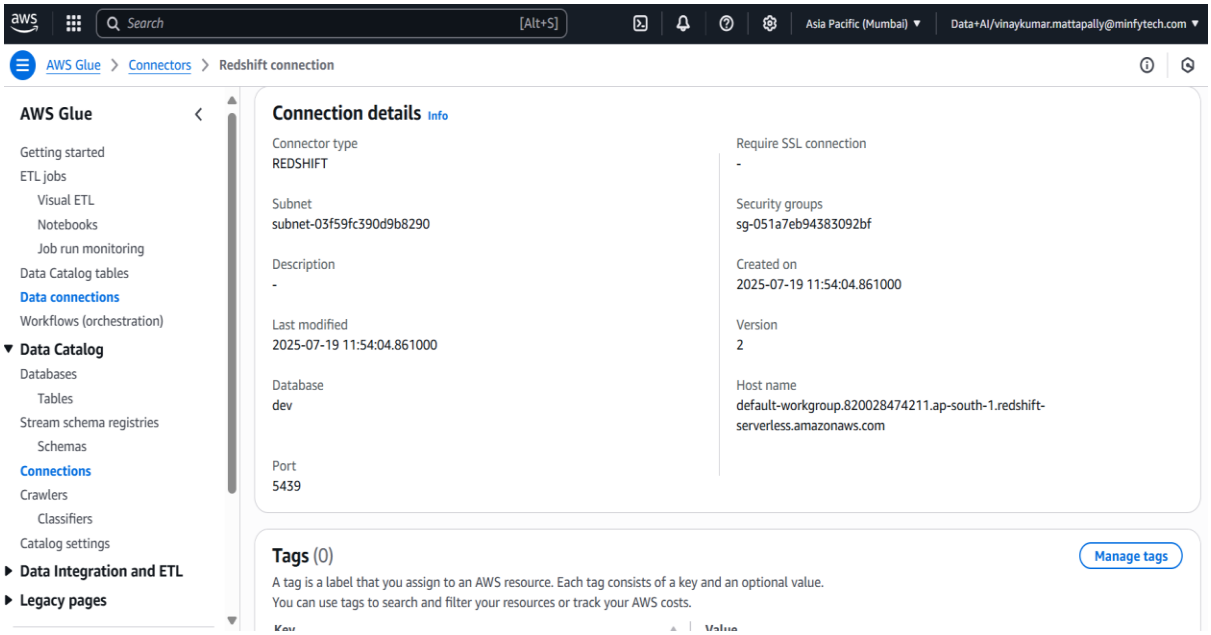
AWS Glue provides a serverless framework to extract data from S3, transform it, and load it into Redshift. This process relies on a Glue Connection to securely access Redshift and a Glue Crawler to understand the data's structure.

4.2.1 Creating the Glue Connection to Redshift

1. Navigate to **AWS Glue** and select **Connections**.
2. **Create connection:**
    - **Name:** redshift_connector

- **Type:** Amazon Redshift
- Select Connect to a Redshift Serverless workgroup and choose your workgroup.
- For **Authentication**, select the AWS Secret (prod/redshift/credentials) you created.

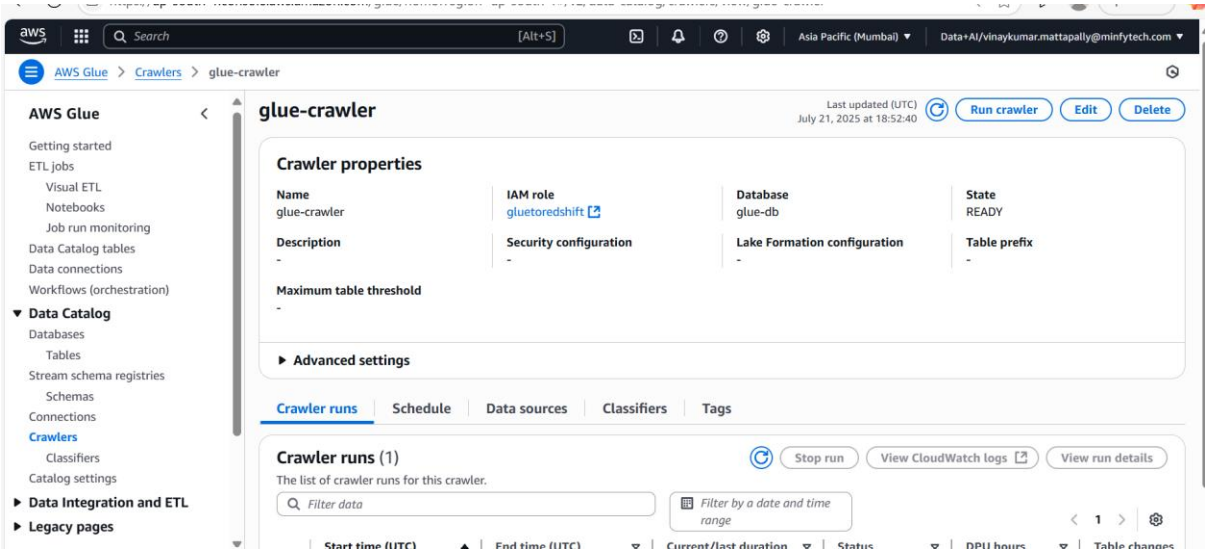
3. **Test Connection:** After creating, select the connection and click **Actions > Test connection**. Choose the glueaccessrole. A "Success" message confirms that your networking and permissions are correct.



4.2.2 The Glue Crawler: Automatic Schema Discovery

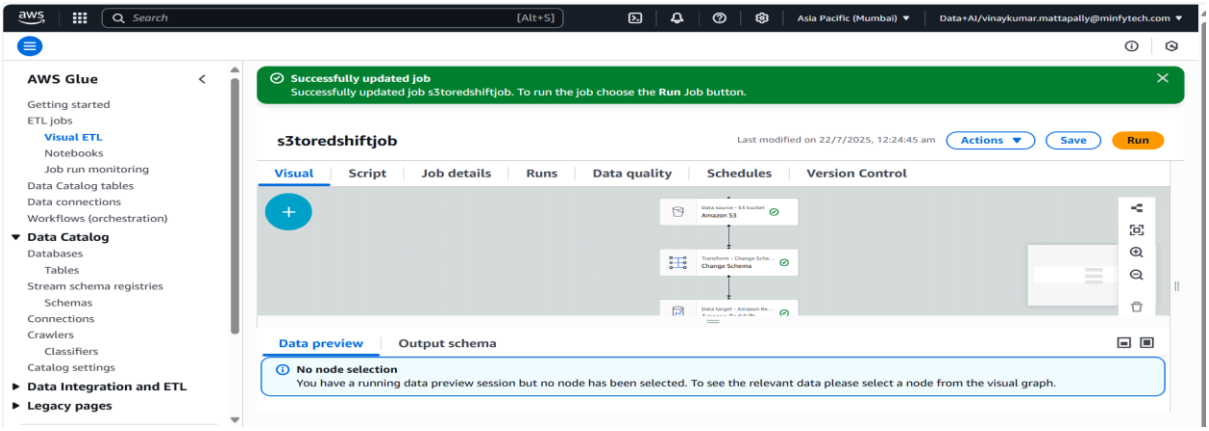
The crawler scans our raw data and creates a metadata table in the Glue Data Catalog.

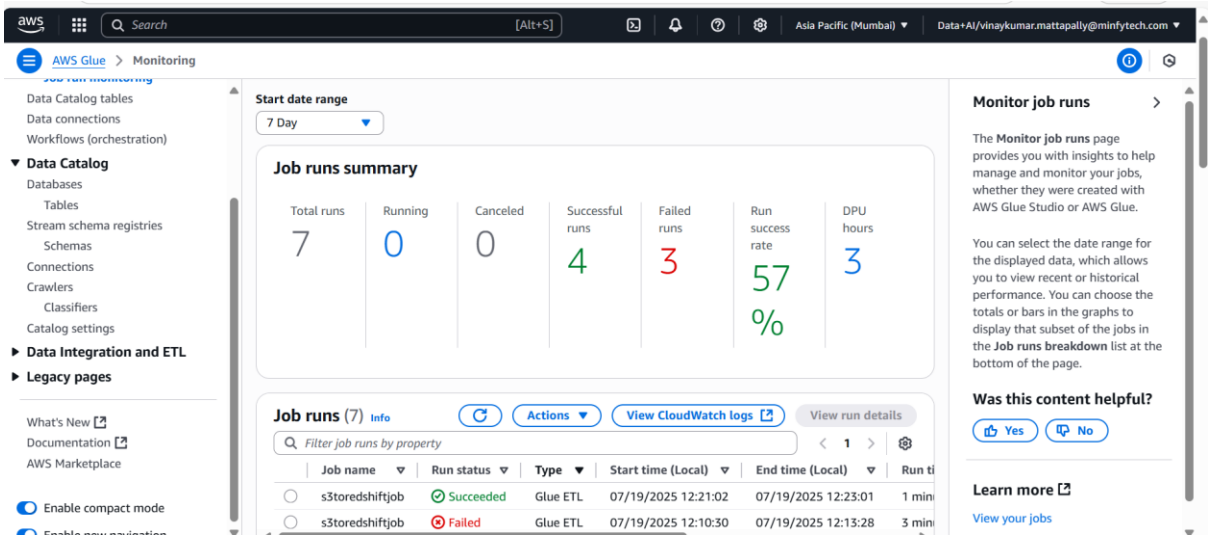
1. In Glue, select **Crawlers** and **Create crawler**.
2. **Name:** s3_leads_crawler.
3. **Data source:** Point it to the S3 path s3://lead-conversion-project-data/raw-data/.
4. **IAM Role:** Select glueaccessrole.
5. **Output:** Create a new database in the Glue Data Catalog named lead_conversion_db.
6. Run the crawler. Upon completion, a new table will be visible in the lead_conversion_db database.

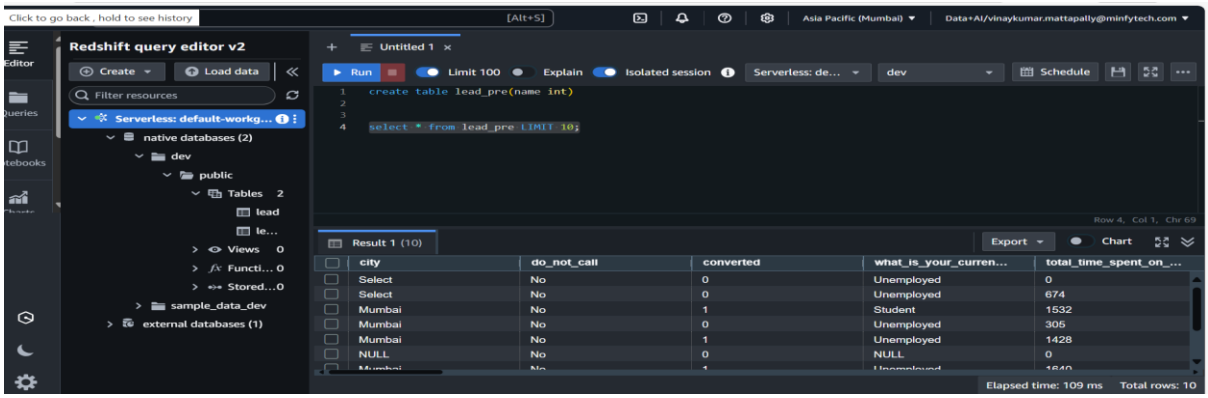


4.2.3 Building the Visual ETL Job (s3ToRedshift)

1. In Glue, navigate to **Glue Studio** and create a new **Visual ETL** job.
2. **Source Node:** Select **AWS Glue Data Catalog** and choose the table created by the crawler.
3. **Transform Node:** Add any desired transformations. For a simple ingestion, you might use the **Change Schema** transform to ensure data types are correct.
4. **Target Node:** Select **Amazon Redshift**.
    - **Connection:** Choose redshift_connector.
    - **Target options:** Specify the target database (dev) and table name (public.leads_raw).
5. **Job Details:** Name the job S3_to_Redshift_ETL and assign the glueaccessrole.
6. Save and **Run** the job. After it succeeds, you can query the public.leads_raw table in Redshift.







5.Amazon SageMaker: The ML Development Environment

5.1. Setting up the SageMaker Environment

All machine learning development for this project is conducted within **Amazon SageMaker**, a fully managed service that provides every developer and data scientist with the ability to

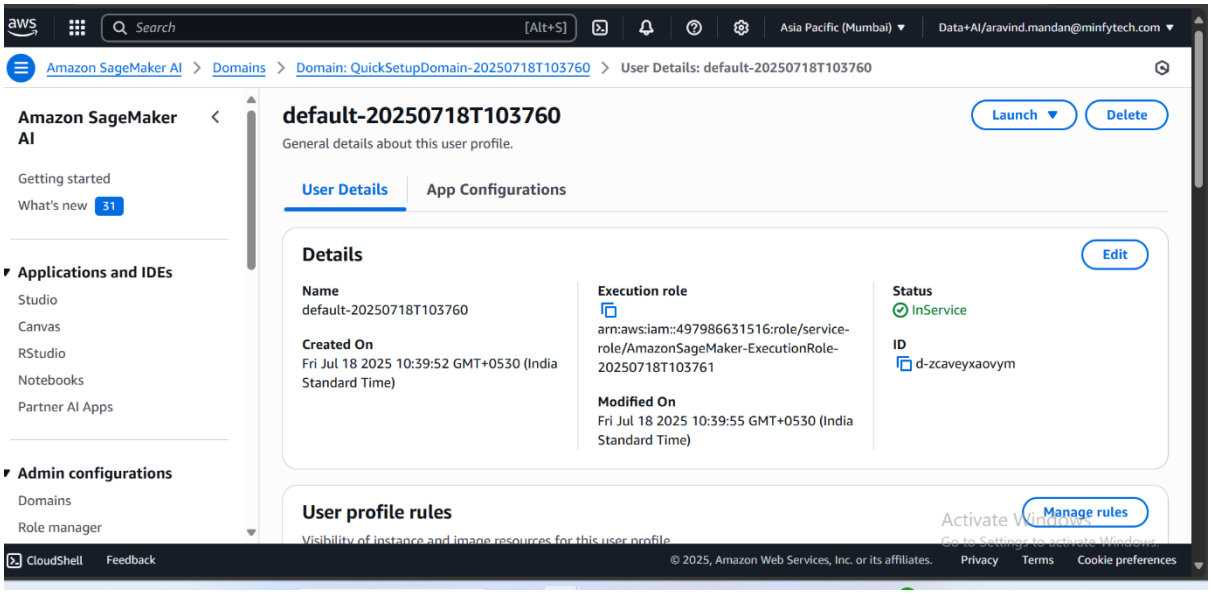
prepare, build, train, and deploy high-quality ML models quickly. We specifically use **SageMaker Studio**, an integrated development environment (IDE) for machine learning.

5.1Creating a SageMaker Domain

A SageMaker Domain is the primary entry point for using SageMaker Studio. It consists of a list of authorized users, a variety of security controls, and an Amazon Elastic File System (EFS) volume that contains the data, notebooks, and other artifacts for the users in the Domain.

Console Walkthrough: Creating the Domain

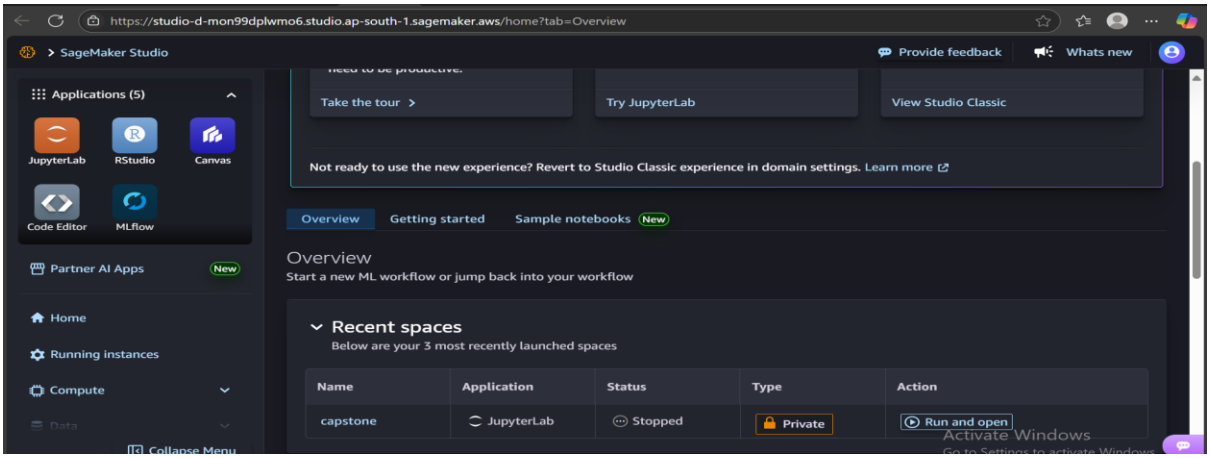
1. **Navigate to SageMaker:** In the AWS Management Console, go to the **Amazon SageMaker** service.
2. **Create a Domain:** On the SageMaker dashboard, click on **Domain** in the left navigation pane, then click **Create Domain**.
3. **Setup:** Choose the **Standard setup** option.
4. **Authentication:** For this guide, select **AWS Identity and Access Management (IAM)** as the authentication method.
5. **Configure Network and Storage:**
    - **VPC:** Select the same Virtual Private Cloud (VPC) that your Redshift cluster and other resources are in. This is critical for secure communication.
    - **Subnet(s):** Choose the private subnets associated with your VPC. Selecting multiple subnets across different Availability Zones ensures high availability.
    - **Security Group:** Attach the `mlops-sg` security group that was configured in Part 2. This allows the SageMaker environment to communicate with other services like Redshift and the MLflow server through the private VPC endpoints.
6. **IAM Role:** Create a new IAM role or use an existing one that has the necessary permissions. This role should have, at a minimum:
    - `AmazonSageMakerFullAccess` policy.
    - Permissions to access the S3 bucket (`leadconversiondata`).
    - Permissions to interact with the Redshift Data API.
7. **Create User Profile:** Within the domain setup, add a default user profile. Give it a name like `ml-developer`.
8. **Review and Submit:** Review all configurations and click **Submit** to create the Domain. This process can take several minutes.



5.1.2 Launching SageMaker Studio

Once the Domain status is "Ready," you can launch the Studio IDE for your user.

1. From the SageMaker Domain page, find your user profile (e.g., `ml-developer`) in the user list.
2. On the right side of the user row, click **Launch** and select **Studio**.
3. A new browser tab will open, and the SageMaker Studio environment will begin to load. This can take a minute or two on the first launch as it provisions the necessary resources.

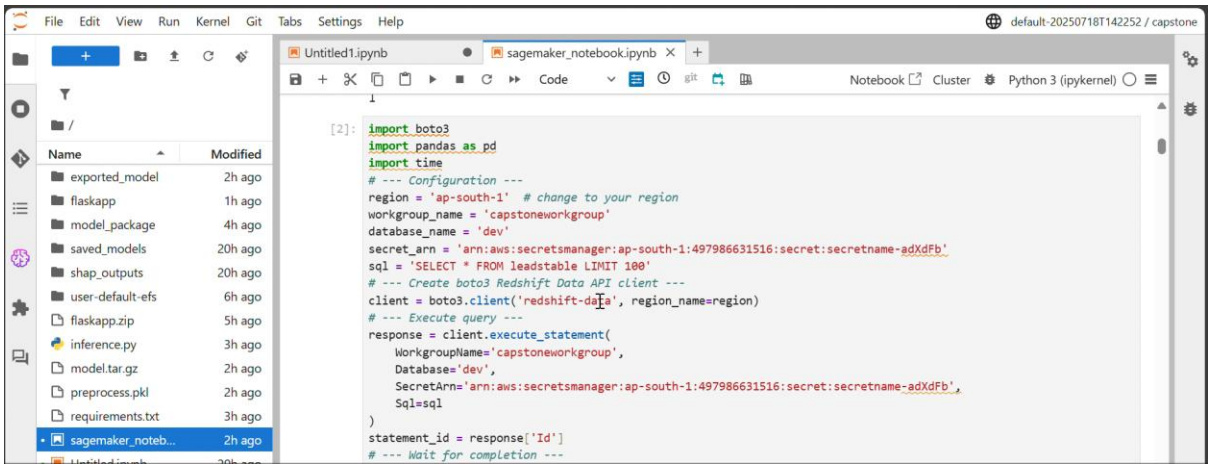


Opening the JupyterLab Notebook

SageMaker Studio provides a fully managed JupyterLab environment. This is where all coding, from data exploration to model training, will take place.

1. **Open Launcher:** Once Studio has loaded, you will be presented with a launcher screen. If not, you can open it from the **File > New Launcher** menu.
2. **Create Notebook:** Under the "Notebooks and compute resources" section, click on **Notebook**.
3. **Select Kernel:** A dialog will pop up asking you to select an image and kernel. Choose the **Data Science** image and the **Python 3** kernel. This provides a Python environment pre-loaded with common data science libraries like pandas, NumPy, and scikit-learn.
4. The new notebook file (e.g., `Untitled.ipynb`) will open, and you are now ready to begin the ML development process.

With the SageMaker Studio environment running, the first step in our ML workflow is to load the data from Redshift and begin the exploratory data analysis.



5.2 ML Development, Model Logging, Serving, and Monitoring in SageMaker

Import Libraries

1. Data Handling & Computing

- **pandas:** Advanced tabular data manipulation with DataFrames.
- **numpy:** Efficient numerical operations on arrays and matrices.

2. Visualization

- **matplotlib.pyplot:** Basic and advanced data visualizations.
- **seaborn:** Statistical, attractive plots built on matplotlib.

3. Model Utilities

- **joblib:** Save/load ML models and pipelines.
- **os:** Manage file paths and directories.

4. Scikit-learn Machine Learning

- **Pipeline, ColumnTransformer:** Build and combine data processing and modeling steps.
- **Preprocessing tools (e.g., MinMaxScaler, OneHotEncoder):** Scale/encode features.
- **SimpleImputer:** Handle missing data.
- **train_test_split, GridSearchCV:** Split data and tune hyperparameters.
- **ML models:** LogisticRegression, DecisionTree, RandomForest.
- **Metrics:** Evaluate classification model performance.

5. Advanced Preprocessing

- **feature_engine.outliers.Winsorizer:** Handle outliers.
- **imblearn.SMOTE:** Fix class imbalance.
- **xgboost.XGBClassifier:** Gradient boosting classifier.

6. MLOps & Experiment Tracking

- **mlflow:** Track, log, register, and deploy models.

7. Model Explainability

- **shap:** Understand feature importance and model decisions.

8. Data Drift & Validation

- **evidently:** Monitor dataset shifts and feature stability.

9. Miscellaneous

- **re:** String/column filtering with regular expressions.
- **warnings:** Control unwanted Python warning messages.

5.2.1 Data Ingestion:

Step 1: Configuration

region = 'ap-south-1'

workgroup_name = 'default-workgroup'

database_name = 'dev'

secret_arn = '<your-secret-arn>'

sql = 'SELECT * FROM lead_pre'

- Sets AWS region, Redshift **workgroup name**, database name, secret ARN for authentication, and SQL query.
- secret_arn is created using **AWS Secrets Manager** and stores Redshift credentials securely.

Step 2: Create Redshift Data API Client

client = boto3.client('redshift-data', region_name=region)

- Uses boto3 to create a Redshift Data API client.
- This is how you interact with Redshift **without needing a JDBC connection**.

Step 3: Run the SQL Query

response = client.execute_statement(

WorkgroupName=workgroup_name,

Database=database_name,

SecretArn=secret_arn,

Sql=sql

)

- Executes the given SQL query on the Redshift Serverless **workgroup**.
- Returns a statement_id used to track the query.

Step 4: Wait for Query Completion

statement_id = response['Id']

desc = client.describe_statement(Id=statement_id)

while desc['Status'] not in ['FINISHED', 'FAILED', 'ABORTED']:

time.sleep(1)

desc = client.describe_statement(Id=statement_id)

- Repeatedly checks the query status.
- Waits (sleep) until the query either **FINISHES** or **FAILS**.

Step 5: Handle Failures

if desc['Status'] != 'FINISHED':

raise Exception(f"Query failed: {desc}")

- If the query fails, aborts the process with an error.

Step 6: Retrieve Query Results

result = client.get_statement_result(Id=statement_id)

- Retrieves the actual data after the query is finished.
- Returns both **column names** and **row data**.

Step 7: Parse the Results

columns = [col['name'] for col in result['ColumnMetadata']]

rows = result['Records']

data = []

for row in rows:

data.append([list(col.values())[0] if col else None for col in row])

- Extracts **column names** from metadata.
- Iterates over each record and extracts values (since each value is a dictionary like {'stringValue': 'abc'}).

Step 8: Convert to Pandas DataFrame

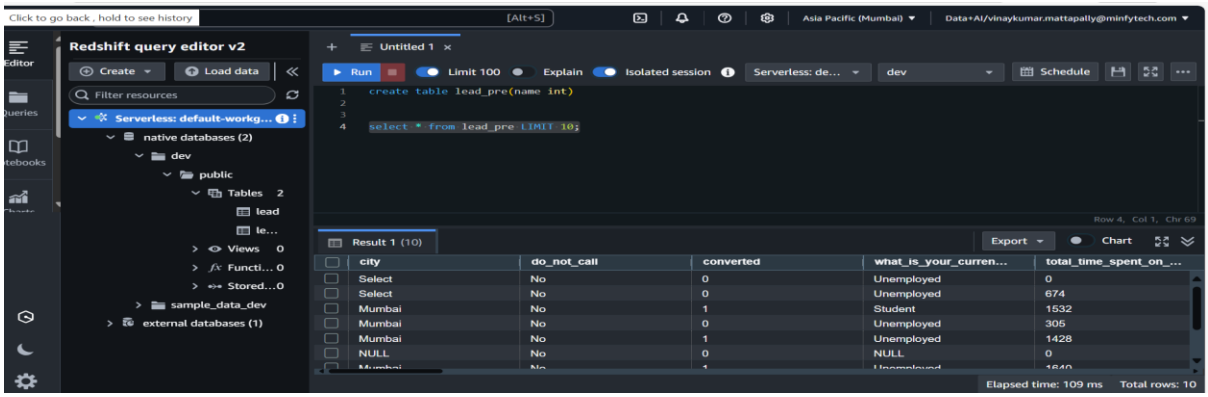
df = pd.DataFrame(data, columns=columns)

- Converts the extracted data and columns into a standard Pandas DataFrame for analysis or ML input.

Step 9: Display the Result

print(df.head())

- Prints the first 5 rows to check the output.



5.2.2 Data Exploration (EDA)and Cleaning:

a. Data Inspection

- df.head(): Shows the first 5 rows to preview your data.
- df.info(): Displays summary info—column types, non-null counts.
- df.describe(): Offers statistics for numerical columns (mean, std, etc.).
- df.isnull().sum(): Counts missing (null) values per column.
- df.duplicated().sum(): Counts duplicate rows.
- df.columns: Lists all column names.

Purpose:

- checking dataset shape, types, missingness, and possible duplicates—standard in the exploratory data analysis (EDA) phase.

b. Data Cleaning & Tidying

df.columns = df.columns.str.strip().str.lower().str.replace(" ", "_"):

- Standardizes column names (lowercase, stripped, underscores instead of spaces), which is crucial for consistency and code reliability.

df.drop(['prospect_id', 'lead_number'], axis=1, inplace=True):

- Removes unique ID columns, as they don't contain predictive information for modeling.

df.replace(["Select", "", None], np.nan, inplace=True):

- Replaces placeholder values ("Select", empty strings, None) with proper np.nan, so that missing value treatment methods can work correctly.

Feature Engineering

a. Mapping Binary Columns

- You have a list of columns (binary_cols) that encode 'Yes'/'No' answers. These are converted to binary numeric representation (1 and 0), e.g., "Yes" → 1, "No" → 0.

Why?

- Many ML models require numerical data. Mapping binary categorical columns makes them machine-readable without introducing spurious order (as would happen with ordinal encoding).

Implementation:

```
binary_map = {"Yes": 1, "No": 0}

for col in binary_cols:

  if col in df.columns:

    df[col] = df[col].map(binary_map)
```

- This is defensive: it only maps columns present in the DataFrame.

Separate Features and Target

X = df.drop(columns=["converted"]):

- All columns except the target (converted) become features.

y = df["converted"]:

- The target label is isolated to y.

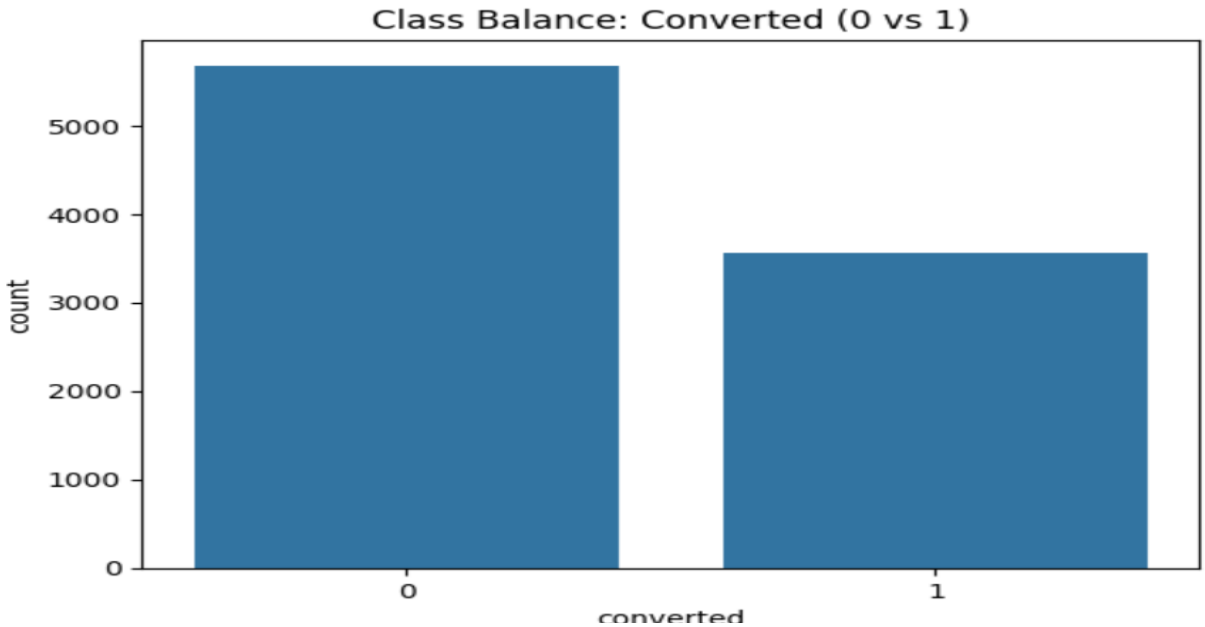
Class Imbalance Visualization

- Before further processing, you check how balanced your target variable is:

  Code:

```
print("\n 📊 Class distribution:")

print(y.value_counts())

sns.countplot(x=y)

plt.title("Class Balance: Converted (0 vs 1)")

plt.show()
```

- Prints and plots the number of positive/negative samples (converted = 1 or 0).



Checking class imbalance is crucial: If much fewer "converted" cases exist, you may need resampling methods (like SMOTE) to balance the classes for model training.

Prepare for Model Training

a. Train-Test Split (Before SMOTE)

X_train_raw, X_test_raw, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

- Splits the data into training (80%) and testing (20%) sets to enable unbiased final model evaluation.

- **Important:** SMOTE (which synthetically balances classes) should only be fit/applied on the **training** set to prevent data leakage.

b. Inspect Class Balance Before SMOTE

Code

print("\n 📊 Class distribution before SMOTE:")

print(y_train.value_counts())

- This lets you see the imbalance in the training set only (what the model will actually train on).

```
📊 Class distribution before SMOTE:
converted
0    51
1    29
Name: count, dtype: int64

🔪 Shape before SMOTE:
X_train_raw: (80, 36)
y_train: (80,)
```

c. Print Dataset Shapes

print(f"\n 🔪 Shape before SMOTE:\nX_train_raw: {X_train_raw.shape}\ny_train: {y_train.shape}")

5.2.3. Building the Preprocessing Pipeline

1. Identify Feature Types

Code:

numeric_features = X.select_dtypes(include=['number']).columns.tolist()

categorical_features = X.select_dtypes(include=['object', 'category']).columns.tolist()

- Detects which columns are numeric and which are categorical based on dataframe data types.

2. Define Ordinal Features

Code:

ordinal_features = list(set(ordinal_features) & set(categorical_features))

categorical_features = list(set(categorical_features) - set(ordinal_features))

- Ordinal features have a meaningful order (e.g., 'Low', 'Medium', 'High').
- Ensures only true categorical-and-in-data ordinal features are used.
- Removes them from the general categorical list to avoid double processing.

3. Build Sub-Pipelines

a) Numeric Features

Code:

```
numeric_pipeline = Pipeline([
  ('imputer', SimpleImputer(strategy='median')),
  ('scaler', MinMaxScaler()),
])
```

- Imputes missing values with the median (robust to outliers).
- Scales features to range for uniformity (important for ML algorithms).

b) Nominal Categorical Features

Code:

```
categorical_pipeline = Pipeline([
  ('imputer', SimpleImputer(strategy='most_frequent')),
  ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])
```

- Imputes missing values with the mode (most common category).
- One-hot encodes (makes each category into its own column, avoids order issues).
- Ignores unknown categories at inference (prevents errors during prediction/deployment).

c) Ordinal Categorical Features

Code:

```
ordinal_pipeline = Pipeline([
  ("imputer", SimpleImputer(strategy="most_frequent")),
  ("ordinal", OrdinalEncoder(categories=ordinal_mapping))
  ])
```

- Imputes with most frequent value.
- OrdinalEncoder uses your provided mapping to map categories to integers preserving their order.

4. Combine Pipelines with ColumnTransformer

Code:

```
preprocessor = ColumnTransformer([
  ("num", numeric_pipeline, numeric_features),
  ("cat", categorical_pipeline, categorical_features),
  ("ord", ordinal_pipeline, ordinal_features)
  ])
```

- Each sub-pipeline is applied to its respective columns without interfering with the others.
- The output is a single, joined, transformed feature set.

5. Save the Pipeline

Code:

```
joblib.dump(preprocessor, pipeline_path)
print(f"✅ Preprocessing pipeline saved at: {pipeline_path}")
```

- Serializes and saves the pipeline to disk for consistent use in training, validation, and production.
- Pipeline can be loaded and applied to new data with joblib.load('preprocess.pkl').

6. Return Results

Returns:

- The constructed pipeline object
- The final list of numeric features
- The final list of (nominal) categorical feature

5.2.4. Data Preprocessing and Class Balancing with SMOTE

This section explains your approach to preparing data for machine learning, focusing on preprocessing with ordinal and categorical features, and using SMOTE for class balancing.

1. Defining Ordinal Features and Their Order

- identified the following as ordinal features:

asymmetrique_activity_index

asymmetrique_profile_index

Both are mapped using the order:

'03.Low' < '02.Medium' < '01.High'

- This implies '01.High' is the highest and '03.Low' is the lowest category, which is necessary for any algorithm that utilizes the natural order in these features.

2. Creating and Fitting the Preprocessing Pipeline

Purpose: Transform raw training features into a form suitable for machine learning models.

How:

- Numeric columns are imputed (median) and scaled (MinMax).
- Nominal categorical columns are imputed (most frequent) and one-hot encoded.
- Ordinal columns are imputed (most frequent) and ordinal encoded with your specified order.
- Saving: The pipeline is saved, making it reusable in future inference or production scenarios.

Code:

```
preprocessor, num_cols, cat_cols = build_preprocessing_pipeline(
  X_train_raw, ordinal_features, ordinal_mapping
)
X_train_processed = preprocessor.fit_transform(X_train_raw)
X_test_processed = preprocessor.transform(X_test_raw)
```

Note: You fit only on training data for proper generalization and to avoid data leakage.

3. Addressing Class Imbalance with SMOTE

Issue: If your target has far more of one class (e.g., many 0's and few 1's), the model may learn to ignore the minority class.

Solution:

Synthetic Minority Over-sampling Technique (SMOTE) is applied after preprocessing and only on the training set.

SMOTE generates new synthetic examples in the minority class, balancing the dataset.

Code:

smote = SMOTE(random_state=42)

X_train_balanced, y_train_balanced = smote.fit_resample(X_train_processed, y_train)

4. Checking Results

After balancing, you print the new class distribution in the training set:

Code:

print(pd.Series(y_train_balanced).value_counts())

```
✅ Preprocessing pipeline saved at: preprocess.pkl

🔄 Applying SMOTE to balance classes...

✅ Class distribution after SMOTE:
converted
0    51
1    51
Name: count, dtype: int64
```

Transform_data pipeline:

1. Preprocessing the Data

Code:

processed = preprocessor.fit_transform(X)

- Transforms X using the provided, fitted scikit-learn pipeline (could also be transform, not just fit_transform, if the pipeline is already fitted).

2. Extracting One-Hot Encoded Feature Names

Code

ohe = preprocessor.named_transformers_['cat'].named_steps['onehot']

cat_feature_names = ohe.get_feature_names_out(categorical_features)

- Navigates the pipeline to find the OneHotEncoder and pulls out the names of each new dummy variable column.

- Example: If "source" has categories "A", "B", you get columns ["source_A", "source_B"].

3. Handling Ordinal Feature Names

Code

```
if ordinal_features:
  ord_feature_names = ordinal_features
else:
  ord_feature_names = []
```

- OrdinalEncoder doesn't change feature names; they stay the same.

4. Combining All Feature Names

Code

```
all_columns = numeric_features + list(cat_feature_names) + ord_feature_names
```

- Assemble the final column name list, preserving order: numeric, then expanded categoricals, then ordinals.

5. Returning a Well-Labeled DataFrame

Code

```
return pd.DataFrame(processed, columns=all_columns)
```

- Wraps the array in a DataFrame for interpretability, feature importance analysis, inspection, etc.

Evaluate_classification_model

Input:

- y_true : True class labels (ground truth, e.g., from y_test)
- y_pred : Predicted class labels (e.g., from model.predict())
- y_proba: Predicted class probability scores (e.g., from model.predict_proba()[:, 1])

Metrics Calculated:

- accuracy: overall correct predictions/proportion.
- precision: correct positives / all predicted positives (helpful if false positives are costly).
- recall: correct positives / all actual positives (helpful if missing positives is costly).

- f1: the harmonic mean of precision & recall.
- roc_auc: the area under the ROC curve, measuring the model's ranking ability

5.2.5. Model Training, Logging, and Explanation with SHAP

1. Setup Directories and MLflow

- Creates directories for saving models and SHAP outputs if they don't exist.
- Sets MLflow to local tracking server and experiment name ("Lead Conversion Prediction").

2. Track Results

- **results** list will collect the evaluation metrics for all models.
- **best_models** dictionary will save the best estimator for each model for later use.

3. Model Training Loop

For each model in the dictionary:

- **Grid Search**: Optimizes hyperparameters with 3-fold cross-validation, scoring by F1.

Code

grid = GridSearchCV(model_info['model'], model_info['params'], cv=3, scoring='f1', n_jobs=-1)

grid.fit(X_train, y_train)

- **Evaluate on Validation Data**:
    - Predicts labels (y_pred) and, if possible, probabilities (y_proba).
    - Runs your earlier evaluate_classification_model() to get accuracy, precision, recall, f1, roc_auc.
    - Saves these results, and the best estimator and its params.
- **Save Model**: Dumps the best estimator to disk.

4. MLflow Logging

- Within an MLflow run:
    - **Logs best hyperparameters** found by GridSearch.
    - **Logs computed metrics**.
    - **Logs the best model** using mlflow.sklearn.log_model.

5. SHAP Explanations and Logging

- **Try-catch** for SHAP (since not all models or setups are supported).
- **SHAP explainer** is chosen based on model type:
    - Tree-based (DecisionTree, RandomForest, XGBClassifier): uses TreeExplainer.
    - LogisticRegression: uses general shap.Explainer.
    - Other types: skips SHAP with a warning.
- Generates a **SHAP summary plot** (feature importance) for the validation set, saves to disk, and logs to MLflow as an artifact.

6. Summarize Metrics

- Collects all results in a DataFrame.
- Prints a summary table showing each model and key metrics (accuracy, precision, recall, F1, ROC-AUC).
- **Returns** both the summary DataFrame and dictionary of best models.

5.2.6. Model Selection and Registration

1. Select the Best Model

- Sorts results_df by ROC AUC (descending).
- Picks the top-performing model name and object from best_models.

2: Retrain Best Model

- Fits the selected model on the full training + validation dataset to improve generalization.

3: Build Full Pipeline

- Combines preprocessor and model using sklearn.pipeline.Pipeline.

4: Save Locally

- Saves the complete pipeline (preprocessing + model) as a .pkl file for local use or backup.

5: Log & Register in MLflow

- Sets MLflow tracking server and experiment.
    - Logs the full pipeline with mlflow.sklearn.log_model.

- Registers the model in the **MLflow Model Registry**.
- Waits to ensure the model is registered.
- Transitions the model to the **"Production"** stage.
- Optionally adds an alias "champion" to this production version.
- Prints a link to view the MLflow run in the UI.

Returns

- full_pipeline: The complete retrained pipeline.
- best_model_name: Name of the top model.
- model_path: Local path where the model .pkl was saved.

retrain_loaded function:

1. Load a previously saved scikit-learn pipeline

Loads both the preprocessor and estimator from disk with joblib.

2. Retrain on the entire input data

Fits the loaded pipeline on the full provided data (processed, y).

3. Evaluate performance

Calculates and prints Accuracy, Precision, Recall, F1, and ROC AUC (if possible).

4. Save the retrained pipeline

Writes the updated, retrained model back to disk.

5. Returns metrics for further tracking/logging.

run pipeline function:

1. Building the Preprocessing Pipeline

Functionality: Constructs a Column Transformer to handle numerical, categorical, and ordinal columns, including all necessary preprocessing (imputation, scaling, encoding).

Inputs: Requires lists for numeric, categorical, and ordinal columns along with their respective encoding order.

Benefit: Guarantees that all downstream modeling and inference use a consistent transformation and feature format.

2. Transforming the Data

Purpose: Applies the preprocessor to the input features so that all machine learning models operate on a uniformly cleaned and encoded feature space.

Feature Names: Extracts fitted feature names, which are essential for interpretability tools like SHAP.

3. Data Splitting: Train/Validation/Test

Splits: 60% training, 20% validation, 20% test

Method: Uses stratified splits to retain the original class distribution, improving model reliability and fairness.

Practice: Ensures models are selected/tuned on validation data, while test data remains "locked" until final evaluation.

4. Model Training, Evaluation, Logging, and Explanation

Operation: Iterates through each model and its hyperparameter grid.

Processes:

- Hyperparameter tuning (via cross-validation)
- Metrics calculation (accuracy, precision, recall, F1, ROC-AUC)
- Automated logging (parameters, metrics, models, SHAP plots) to MLflow
- SHAP summary plots for interpretability of results
- Outcome: Compiles a summary DataFrame of all key metrics and saves best model artifacts.

5. **Best Model Retraining, Pipeline Creation, and Registration**

- Retraining: The top-performing model (by ROC-AUC) is retrained on both training and validation data.

Model Dictionary (models)

This dictionary defines **four classification models** along with their **hyperparameter grids** for tuning using techniques like GridSearchCV.

1. Logistic Regression

- Model: LogisticRegression with L2 regularization.
- Hyperparameter:
    - C: Regularization strength (smaller → stronger regularization).

2. Decision Tree

- Model: DecisionTreeClassifier.

- Hyperparameters:
    - max_depth: How deep the tree can grow.
    - min_samples_split: Min samples required to split an internal node.

3. **Random Forest**
    - Model: RandomForestClassifier.
    - Hyperparameters:
        - n_estimators: Number of trees.
        - max_depth: Max depth of each tree.
4. **XGBoost**

- Model: XGBClassifier.
- Hyperparameters:
    - n_estimators: Number of boosting rounds.
    - max_depth: Depth of each tree.

run_pipeline(X, y, models)

This function call likely:

1. **Preprocesses** X and y
2. Performs **hyperparameter tuning** for each model
3. **Trains** all models using the defined grids
4. Logs metrics to **MLflow**
5. Possibly generates **SHAP explainability plots**
6. Selects and registers the **best-performing model**

```
Training: LogisticRegression
2025/07/19 12:50:21 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/07/19 12:50:24 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input_example` parameter when logging the model to auto infer the model signature.
SHAP saved & logged: shap_outputs/LogisticRegression_shap_summary.png
View run LogisticRegression at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34/runs/6ea47081f4a948279239d5ef51188a2c
View experiment at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34

Training: DecisionTree
2025/07/19 12:50:26 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/07/19 12:50:28 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input_example` parameter when logging the model to auto infer the model signature.
SHAP saved & logged: shap_outputs/DecisionTree_shap_summary.png
View run DecisionTree at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34/runs/1ebaac0c1be847e6945aed946638f220
View experiment at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34

Training: RandomForest
2025/07/19 12:50:31 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/07/19 12:50:34 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input_example` parameter when logging the model to auto infer the model signature.
SHAP saved & logged: shap_outputs/RandomForest_shap_summary.png
View run RandomForest at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34/runs/af778e1d194c403490352ffa509088b8
View experiment at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34

Training: XGBoost
2025/07/19 12:50:36 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/07/19 12:50:39 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input_example` parameter when logging the model to auto infer the model signature.
SHAP saved & logged: shap_outputs/XGBoost_shap_summary.png
```

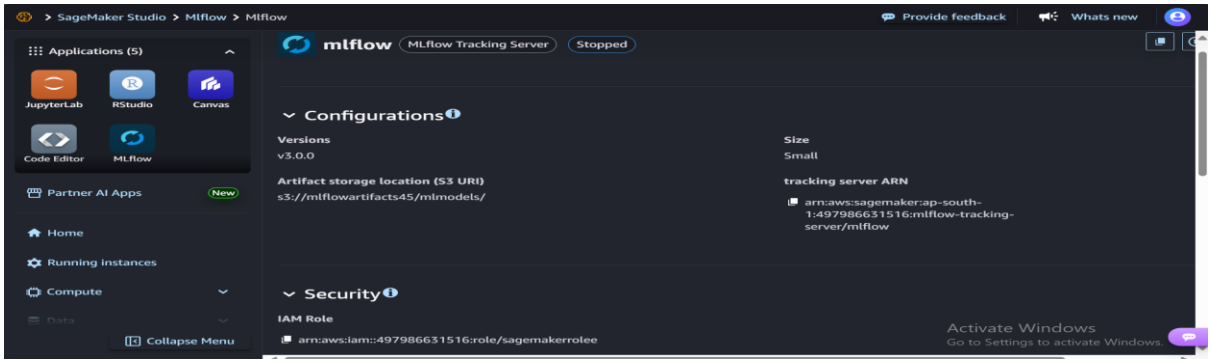
```
All Model Validation Metrics:
              model  accuracy  precision    recall        f1  roc_auc
LogisticRegression      0.85   0.833333 0.714286 0.769231 0.945055
      DecisionTree      0.60   0.400000 0.285714 0.333333 0.527473
      RandomForest      0.85   0.833333 0.714286 0.769231 0.945055
           XGBoost      0.85   0.750000 0.857143 0.800000 0.857143

Best model selected: LogisticRegression
Final pipeline saved at: saved_models/final_LogisticRegression_pipeline.pkl
2025/07/19 12:50:40 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/07/19 12:50:42 WARNING mlflow.models.model: Model logged without a signature and input example. Please set `input_example` parameter when logging the model to auto infer the model signature.
Registering model to MLflow Model Registry: LogisticRegression
Registered model 'LogisticRegression' already exists. Creating a new version of this model...
2025/07/19 12:50:43 WARNING mlflow.tracking._model_registry.fluent: Run with id 07d222d4c16b470481e37ae9bf09fefe has no artifacts at artifact path 'model', registering model based on models:/m-a381f6510ffd49b49c2a12cf187cdf4e instead
2025/07/19 12:50:44 INFO mlflow.store.model_registry.abstract_store: Waiting up to 300 seconds for model version to finish creation. Model name: LogisticRegression, version 13
Created version '13' of model 'LogisticRegression'.
Model 'LogisticRegression' version 13 moved to 'Production'.
Unable to set alias 'champion': 'MlflowClient' object has no attribute 'set_model_version_alias'
View run: http://localhost:5000/#/experiments/34/runs/07d222d4c16b470481e37ae9bf09fefe
View run Final_LogisticRegression at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34/runs/07d222d4c16b470481e37ae9bf09fefe
View experiment at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/34

Final model pipeline: LogisticRegression
Saved pipeline at: saved_models/final_LogisticRegression_pipeline.pkl
(Pipeline(steps=[('preprocessing',
```

6.MLOps, Deployment & Monitoring

6.1 Experiment Tracking with MLflow

6.1.1 Setting Up the MLflow Tracking Server

To centralize experiment tracking, we deploy an MLflow tracking server on a small EC2 instance within our VPC. This provides a persistent, shared location for all developers to log and view experiment results. The server is exposed via its private IP within the VPC, and the security group is configured to allow traffic on port 5000.



6.1.2 Integrating MLflow into the Training Pipeline

MLflow is seamlessly woven into our `train_pipeline` function.

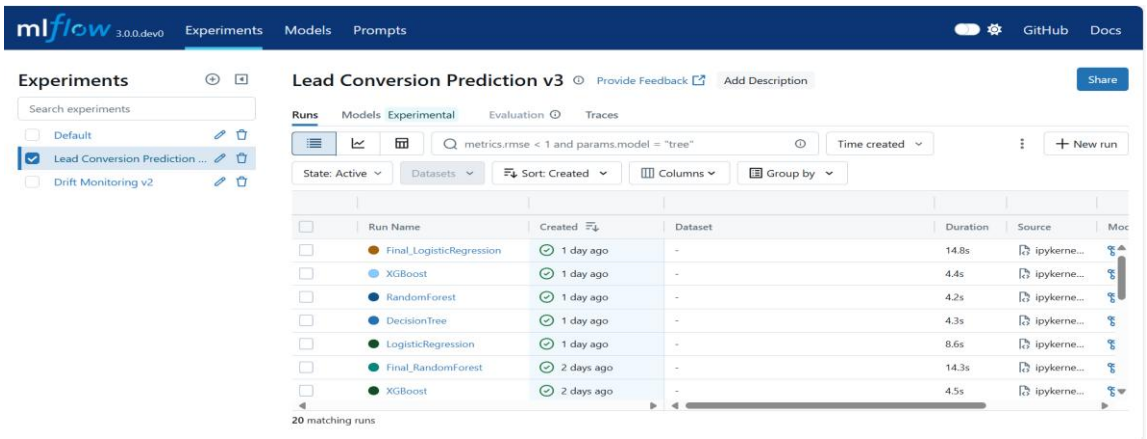
- `mlflow.set_tracking_uri()`: Points the logging client to our EC2-hosted server.
- `mlflow.set_experiment()`: Organizes runs under a specific experiment name, "Lead Conversion Prediction".
- `with mlflow.start_run()`:: Creates a new run context for each model, automatically capturing metadata.
- `mlflow.log_param()` & `mlflow.log_metric()`: Logs hyperparameters and evaluation scores.
- `mlflow.sklearn.log_model()`: Logs the trained model object, its dependencies, and a `conda.yaml` file, making it fully reproducible.
- `mlflow.log_artifact()`: Logs supplementary files like the SHAP plot.

6.2 Model Registration and Lifecycle Management

Saving and Registering the Champion Model

After evaluating all models, our pipeline identifies the "champion" model (typically the one with the highest ROC-AUC or F1-score). This model is then prepared for production.

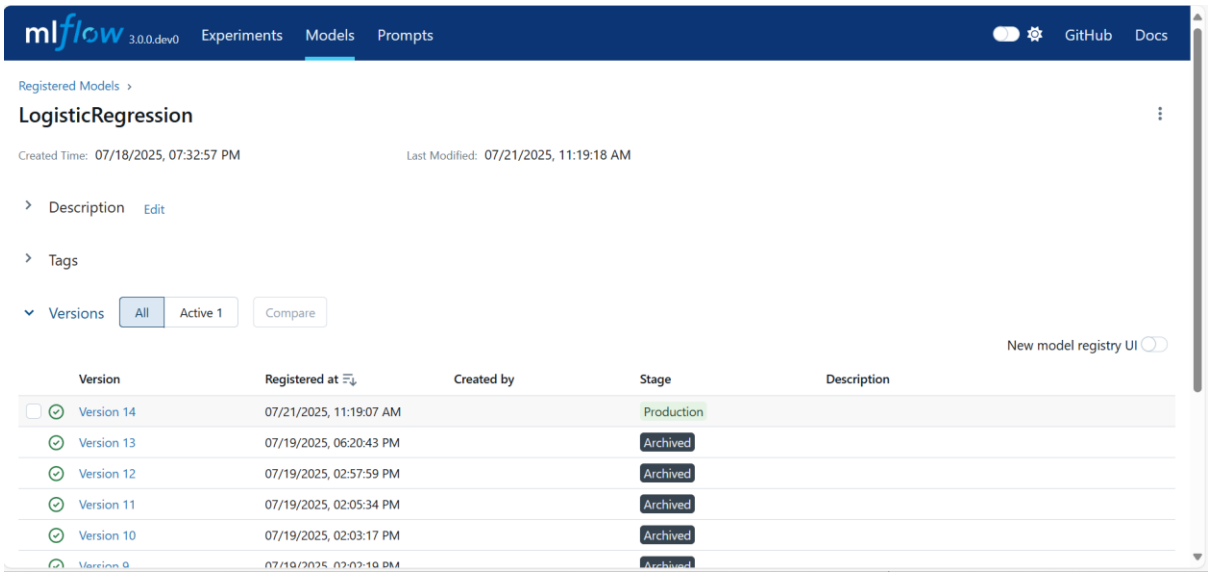
1. **Retrain:** The champion model is retrained on the *entire* dataset (training + validation) to learn from all available data.
2. **Build Full Pipeline:** The retrained model is combined with the fitted `preprocessor` into a single, end-to-end `scikit-learn` pipeline. This ensures that raw data can be fed directly into the final artifact for prediction.
3. **Log and Register:** This full pipeline is logged to a final MLflow run and then **registered** in the MLflow Model Registry with a specific name (e.g., `lead-conversion-classifier`).



6.2.1. MLflow Model Registry for Versioning and Staging

The Model Registry is a central hub for managing the lifecycle of our models. It allows us to:

- **Version Models:** Every time we register a model with the same name, a new version is created.
- **Manage Stages:** We can assign stages to model versions, such as **Staging**, **Production**, or **Archived**.
- **Transition Models:** We can programmatically transition our newly registered model to the "Production" stage, signaling that it is the official version to be used for inference. This is a key step in controlled, automated deployments.



6.3. Model Deployment and Serving

6.3.1. Loading the Production Model from the Registry

Load_and_predict_from_registry

Automatically loading and predicting with the latest MLflow Model Registry model:

1. Automatic Model Discovery

get_latest_production_model_name:

- Connects to the MLflow Model Registry.
- Searches for all registered models and their versions.
- Finds the most recently updated version in a given stage (default: "Production") or with a given alias (e.g., "champion").
- Returns the model's name to be loaded for prediction.

2. Model Loading and Prediction

load_and_predict_from_registry_auto:

- Uses the discovered model name and stage/alias to build the MLflow model URI.
- Loads the registered pipeline with mlflow.sklearn.load_model.
- Predicts on your raw test data (X_test_raw).
- Prints a sample of predictions for validation.

3. Sample Usage

- To predict using the latest "Production" model:

y_pred = load_and_predict_from_registry_auto(X_test_raw, stage="Production")

- Or, if you use model aliases (MLflow 2.3+):(Optional)

y_pred = load_and_predict_from_registry_auto(X_test_raw, alias="champion"

```
✅ Will load LogisticRegression version 12 (stage/alias: 'Production')
📦 Loading from models:/LogisticRegression/Production
Error displaying widget: model not found
✅ Predictions complete. Example: [0 0 0 0 0]
```

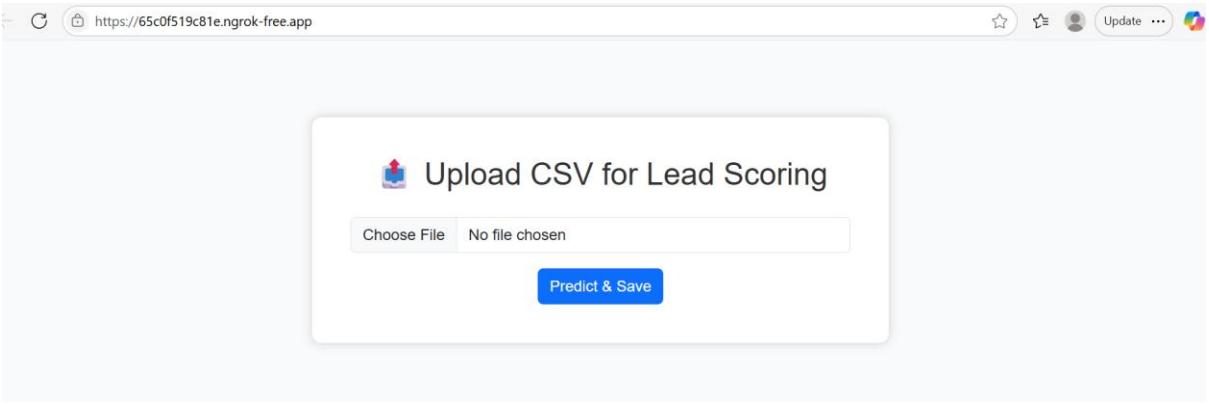
6.3.2. Real-time Prediction with Flask and Ngrok

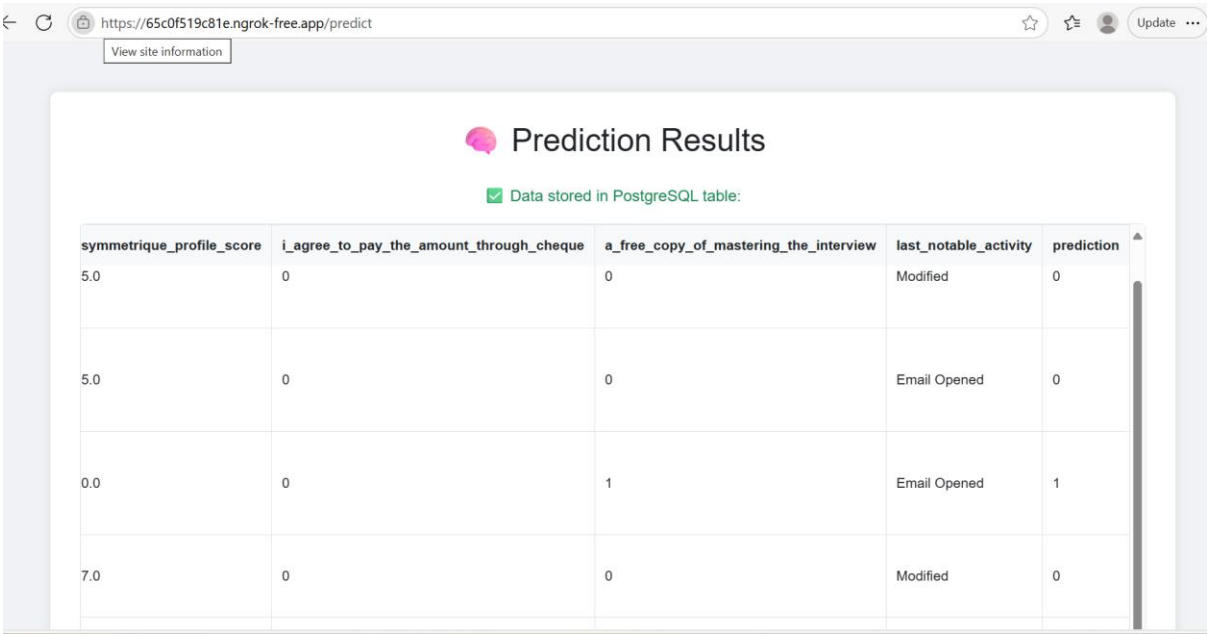
Flask with Ngrok:

1. **Flask App Setup**
   - Defines UPLOAD_FOLDER for saving uploaded files.
   - Sets logging and MLflow tracking URI (SageMaker-hosted).
2. **Model Handling**
   - get_latest_production_model_name(): Finds the latest model in "Production".
   - load_model_from_registry(): Loads the model from MLflow Model Registry.
3. **Routes**
   - / → Shows file upload form (upload.html)
   - /predict → Accepts CSV, cleans it, maps binary values, loads model, makes predictions, and shows results (results.html)
4. **Prediction Logic**

- Cleans column names
- Replaces "Select", "", "None" with NaN
- Fills missing required columns
- Maps Yes/No to 1/0 in binary columns
- Loads the latest production model and predicts

5. **Ngrok Tunnel**
    - Starts an ngrok tunnel for public access to the app





6.4. Drift Analysis with Evidently & MLflow

1. Preprocessing

build and fit a custom preprocessing pipeline on the *entire* dataset:

Code:

```
preprocessor, numeric_features, categorical_features = build_preprocessing_pipeline(X, ordinal_features, ordinal_mapping)

processed = preprocessor.fit_transform(X)
```

- **Why?** Full preprocessing ensures all categorical/ordinal/continuous variables are handled (imputed/encoded/scaled) the same way as in modeling.
- **Best Practice Note:** For **evidently drift** and data diagnostics, it's common to preprocess the *whole* set so that feature columns match and statistics are compatible.

2. Feature Names Extraction

Code:

```
feature_names = preprocessor.get_feature_names_out()
```

- **Why?** After OneHotEncoder or similar expanders, you get the true, final column list to use downstream.

3. Processed DataFrame Construction

Code:

```
df_all = pd.DataFrame(processed, columns=feature_names)
```

- **Why?** Ensures all downstream checks, splits, metrics, and drift analysis work on a DataFrame with correct, descriptive columns.

4. Train/Validation/Test Split (Stratified)

Code:

```
X_temp, X_test, y_temp, y_test = train_test_split(df_all, y, test_size=0.2, stratify=y, random_state=42)

X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp, test_size=0.25, stratify=y_temp, random_state=42)
```

- **Result:** 60% train, 20% val, 20% test with preserved class balance.
- **Why?** Stratification is critical if you have class imbalance.

5. Column Consistency Across Splits

Code:

```
def ensure_same_columns(df, reference_columns):
```

return df.reindex(columns=reference_columns, fill_value=0)

...

X_train = ensure_same_columns(X_train, feature_names)

X_val = ensure_same_columns(X_val, feature_names)

X_test = ensure_same_columns(X_test, feature_names)

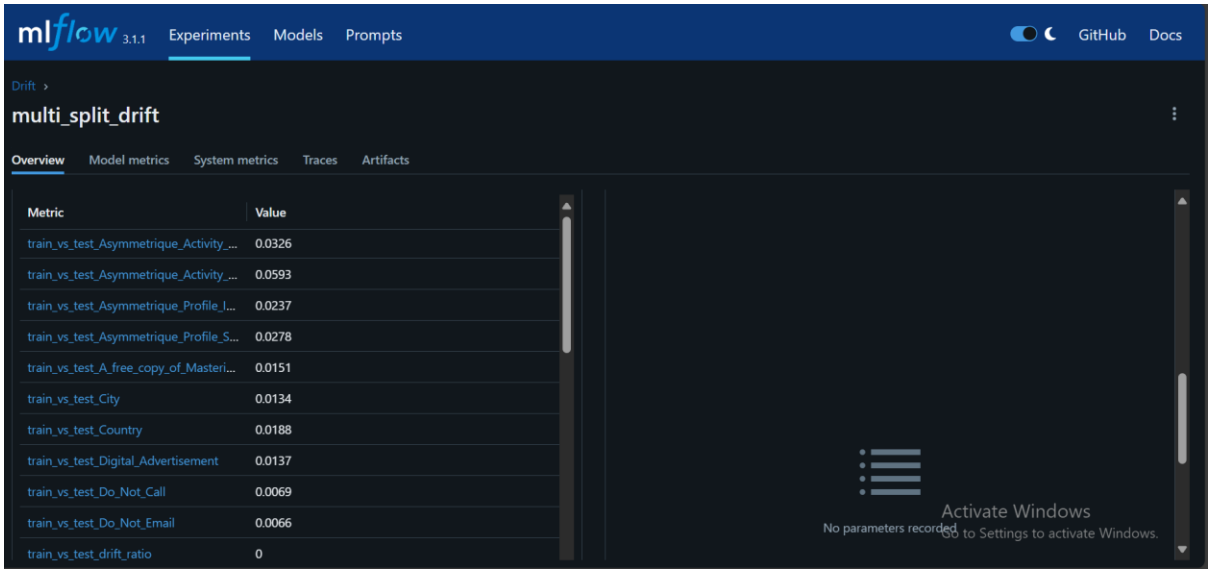
- **Why?** Occasionally, splits may *lose* columns (like a certain category in a small split). You guarantee all dataframes have identical columns, with missing features zero-filled.
- **This is essential** for compatibility with monitoring, modeling, and explainability tools.

6. Drift Report Function

function does all the heavy lifting:

- Builds an **Evidently Report** for each pair (train/val, train/test, val/test).
- Saves HTML reports locally.
- Logs both the HTMLs (as artifacts) and numeric drift scores (as metrics) to MLflow for fully traceable drift monitoring.

```
2025/07/19 09:30:49 INFO mlflow.tracking.fluent: Experiment with name 'Drift Monitoring v2' does not exist. Creating a new experi
ment.
✅ Preprocessing pipeline saved at: preprocess.pkl
🚀 Running drift check: train_vs_val
✅ Logged drift metrics for train_vs_val to MLflow.

🚀 Running drift check: train_vs_test
✅ Logged drift metrics for train_vs_test to MLflow.

🚀 Running drift check: val_vs_test
✅ Logged drift metrics for val_vs_test to MLflow.

📌 Drift reports logged under run ID: fb0b401508844db3a6f487fb8473b989
🏃 View run at: http://127.0.0.1:5000/#/experiments/35/runs/fb0b401508844db3a6f487fb8473b989
🏃 View run drift_report_multi_split at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/35/runs/fb0b401508844db3a6f48
7fb8473b989
🧪 View experiment at: https://ap-south-1.experiments.sagemaker.aws/#/experiments/35
```



7.Full Automation with MWAA

7.1. Orchestrating the MLOps Lifecycle with Airflow

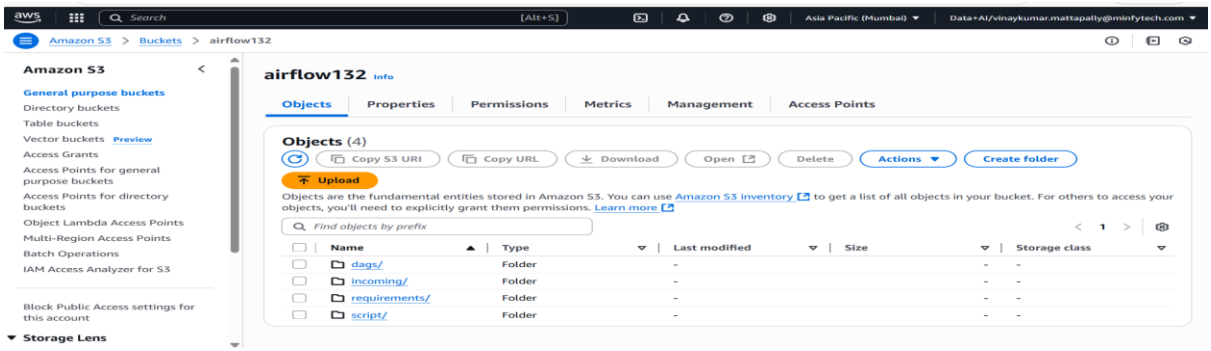
To achieve full, hands-off automation, the entire MLOps workflow is orchestrated using **Amazon Managed Workflows for Apache Airflow (MWAA)**. MWAA is a managed service that makes it easy to run Apache Airflow on AWS. The entire automated process—from data drift detection to model retraining and deployment—is defined as a **Directed Acyclic Graph (DAG)**, which is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

7.2 Setting up the MWAA Environment

Before deploying the DAG, you must set up the MWAA environment and its required resources.

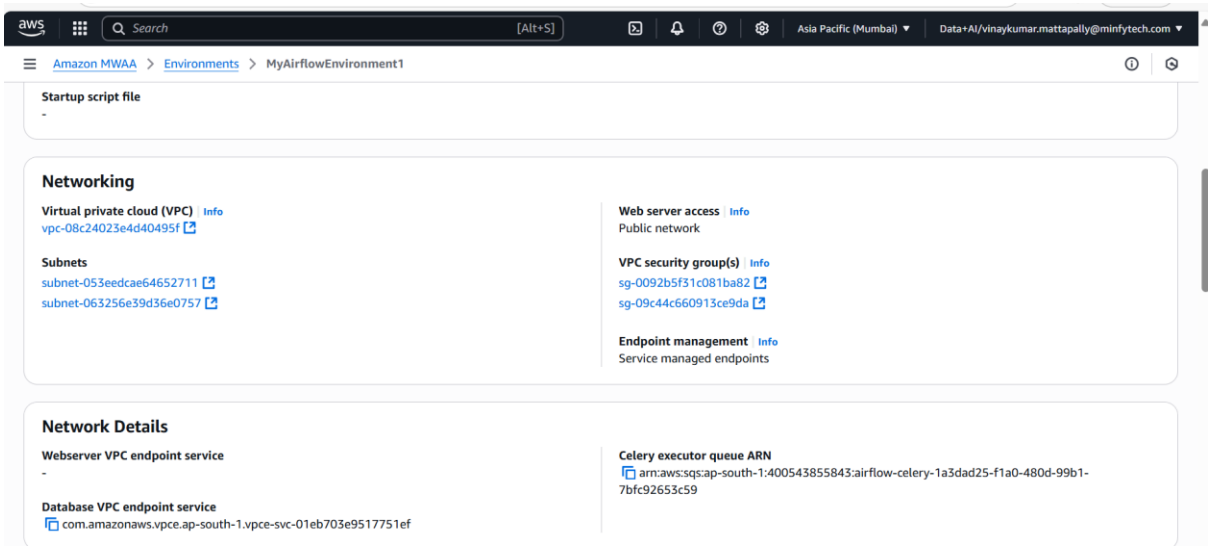
Step 1: Create an S3 Bucket for MWAA Assets MWAA uses an S3 bucket to store your DAGs, custom plugins, and Python dependencies (`requirements.txt`).

1. Create a new S3 bucket with a unique name (e.g., `mwaa-lead-conversion-dags`).
2. Enable **versioning** on this bucket to keep a history of your DAGs and configuration files.



Step 2: Configure the VPC and Networking MWAA requires a specific network setup to run securely and access other resources.

1. **Use the same VPC:** The MWAA environment must be launched in the same VPC as your Redshift, SageMaker, and MLflow resources to communicate with them privately via the VPC endpoints.
2. **Subnets:** You must provide at least two **private subnets** in different Availability Zones for high availability.
3. **Internet Access:** The private subnets must have a route to a **NAT Gateway**. This is required for the Airflow workers to install Python dependencies from public repositories like PyPI.
4. **Security Group:** The MWAA environment should use the same `mlops-sg` security group created in Part 2. This ensures it has the necessary inbound and outbound rules to communicate with the MLflow server, Redshift, S3, and other services.



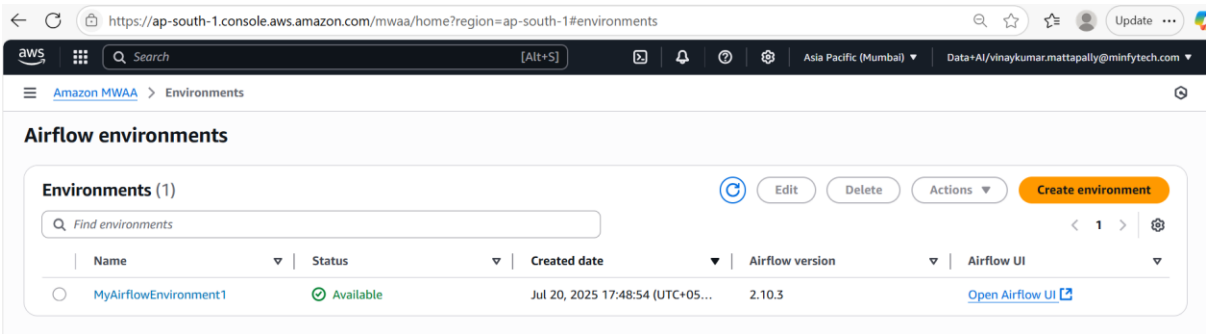
Step 3: Create the MWAA Environment

1. Navigate to the **Managed Workflows for Apache Airflow** service in the AWS Console.
2. Click **Create environment**.
3. **Details:**
    - **Name:** `LeadConversion-MLOps-Prod`
    - **Airflow version:** Choose a recent, stable version
    - **S3 bucket:** Browse and select the S3 bucket you created in Step 1.

- **DAGs folder path:** `dags`
- **Requirements file:** `requirements.txt`

4. **Networking:** Select your VPC, the private subnets, and the `mlops-sg` security group.
5. **Execution role:** Create a new IAM role that grants MWAA permissions to access its S3 bucket and CloudWatch Logs. Attach policies that also allow it to interact with SageMaker, Redshift, and any other services your DAG will orchestrate.

Click **Create environment**. Provisioning can take up to 30 minutes.



7.3. The Automated Retraining DAG (Directed Acyclic Graph)

7.3.2.Preparing and Uploading the DAG File

1. **Structure the Project Folder:** On your local machine, create the following folder structure:

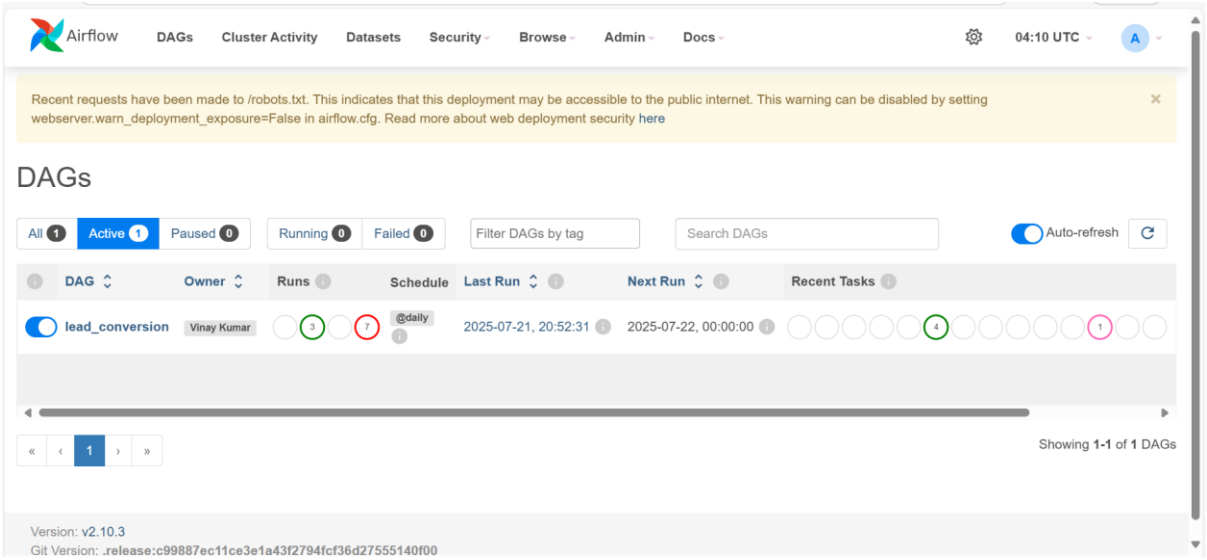
```
mwaa_Capstone/
├── dags/
│   └── dag.py
├──requirements
│     └──requirements.txt
├──incoming
│     └──new_data.csv
├──script
   └──lead_prediction_cloud.py
```

11. **Create `requirements.txt`:** This file lists all Python packages needed by your DAG.

```
apache-airflow-providers-amazon
mlflow-skinny
scikit-learn
pandas
numpy
boto3
evidently
```

19. **Upload to S3:** Upload the `dags` folder and the `requirements.txt` file to the root of your MWAA S3 bucket. MWAA will automatically detect these files and install the requirements on its workers.

The Automated Retraining DAG



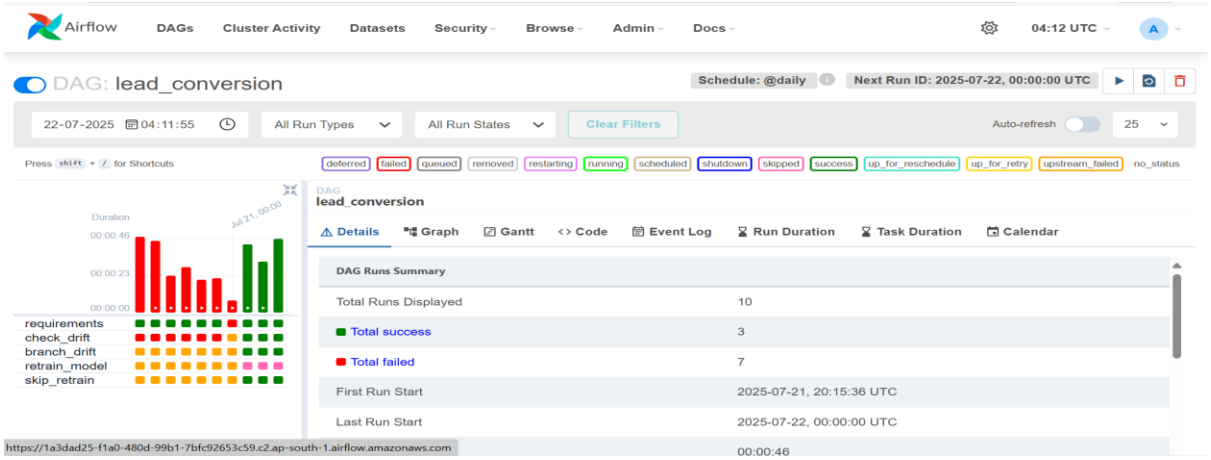
The `dag.py` file contains the logic for the automated workflow.

7.3.2. Workflow Logic and Conditional Retraining

1. **Scheduled Trigger:** The DAG is scheduled to run at a regular interval (e.g., weekly).
2. **Automated Drift Analysis:** The first task executes the Evidently drift detection script. The results are logged to MLflow.
3. **Conditional Retraining:** The next task is a conditional branch. It checks the drift metrics from the previous step against a predefined threshold (e.g., `Share of Drifted Columns > 0.5`).
    - If **Drift Detected**: The DAG proceeds down the retraining path.
    - If **No Drift**: The DAG concludes, logging that no action was needed.
4. **Automated Model Retraining:** If triggered, a new task runs the full `train_pipeline`, using the latest data to train, evaluate, and find a new champion model.
5. **Automated Registration:** If the new model outperforms the old one, it is automatically registered in the MLflow Model Registry.
6. **Automated Deployment:** The final task transitions the new model version to the "Production" stage, making it the active model for the prediction service.
7. **Alerting:** Throughout the process, the DAG sends notifications (e.g., via email or Slack) to the MLOps team about key events like drift detection, retraining success, or any failures.

Apache Airflow web interface, specifically displaying the status of the **"lead_conversion" DAG** (Directed Acyclic Graph), which automates and schedules the steps in your lead conversion prediction pipeline.

7.3.3. Monitoring DAG Runs in the Airflow UI



Key Sections Explained:

1. DAG Overview

- **DAG Name:** lead_conversion
- **Schedule:** Runs daily (@daily). The next run is scheduled for July 22, 2025, at midnight UTC.

2. Run History Visualization

- **Bar Chart and Grid:**
  - The **bar chart** on the left (vertical colored bars) shows the duration of each DAG run. Green bars indicate successful runs while red bars show failures.
  - The **square matrix** (below the chart) displays the status of individual tasks within each run. Colored dots represent task outcomes (green for success, red for failure, etc.) for tasks such
    as requirements, check_drift, branch_drift, retrain_model, and skip_retrain.

3. Run Summary

- **Total Runs Displayed:** 10
- **Total Success:** 3 runs completed successfully.
- **Total Failed:** 7 runs failed.
- **First Run Start:** July 21, 2025, 20:15:36 UTC
- **Last Run Start:** July 22, 2025, 00:00:00 UTC
- **Last Run Duration:** 46 seconds

4. Task Status Legend

- Labels at the top (blue, red, green, yellow, etc.) categorize task states—**success**, **failed**, **skipped**, **running**, and more—making it easy to filter and analyze execution history.

5. Control Panel

- Allows you to trigger, pause, or refresh DAG runs and toggle auto-refresh.

Interpretation

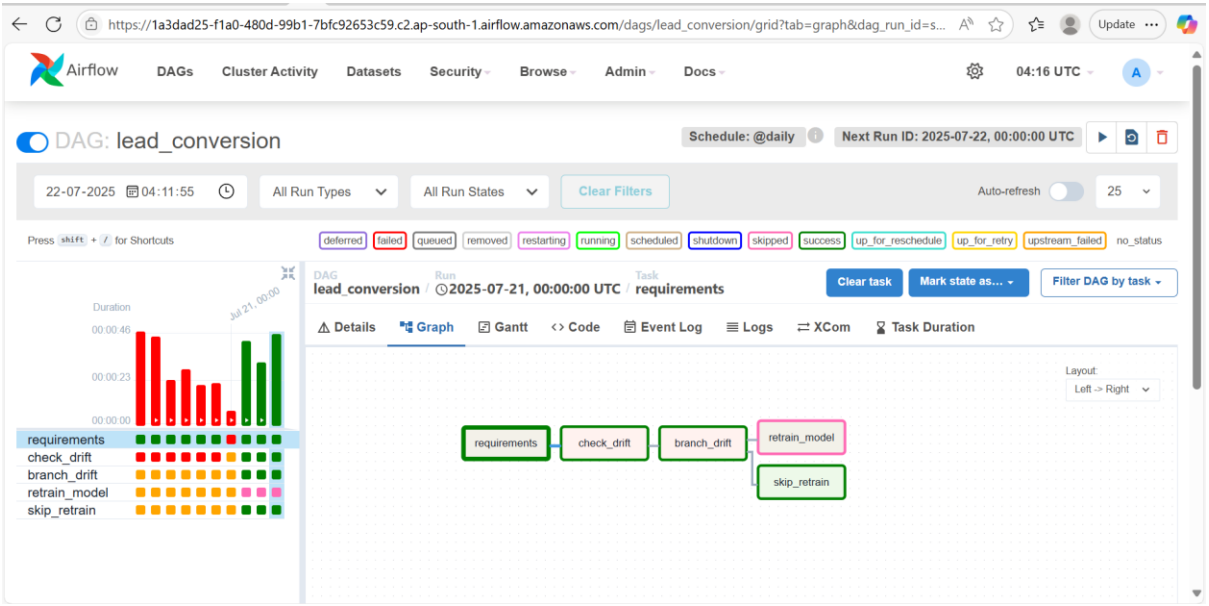
- The lead_conversion DAG is set to run daily, automating different ML pipeline steps (data requirements check, drift detection, retraining, etc.).
- Of the last 10 runs, **only 3 succeeded and 7 failed**, suggesting there are issues in the workflow that need attention.
- Task-level status helps quickly identify recurring failure points (for example, if check_drift or retrain_model consistently fails).
- This dashboard helps users monitor workflow health, debug failures, and ensure reliable automation of the machine learning pipeline.

Let me know if you need specifics about any task or further troubleshooting guidance!

Related

What specific aspect of deployment or model explanation do I need clarity on

Graph:



Appendix

8 .Technology & Libraries Stack

- **Cloud Platform:** Amazon Web Services (AWS)
- **Data Storage:** Amazon S3, Amazon Redshift
- **Data Integration (ETL):** AWS Glue (Studio, Crawlers, Data Catalog)

- **Security & Networking:** AWS IAM, AWS VPC, AWS Secrets Manager, AWS KMS
- **ML Development:** Amazon SageMaker Studio Lab
- **MLOps & Tracking:** MLflow
- **Orchestration (Future):** Managed Workflows for Apache Airflow (MWAA)
- **Serving:** Flask, Ngrok
- **Core Python Libraries:**
    - **Data Handling:** pandas, numpy
    - **Visualization:** matplotlib, seaborn
    - **ML & Preprocessing:** scikit-learn
    - **Advanced Models:** xgboost, lightgbm
    - **Model Explainability:** shap
    - **Data Drift:** evidently
    - **AWS Interaction:** boto3
    - **Serialization:** joblib

9 . Code Repository & Resources

- **Code Repository:** VinaykumarMinfy/Lead_Prediction_Capstone
- **Official Documentations:**
    - AWS Glue
    - Amazon SageMaker
    - MLflow
    - Evidently AI