Bootcamp Project 2 — Transactions and Loan Data for a Customer

Project Objective

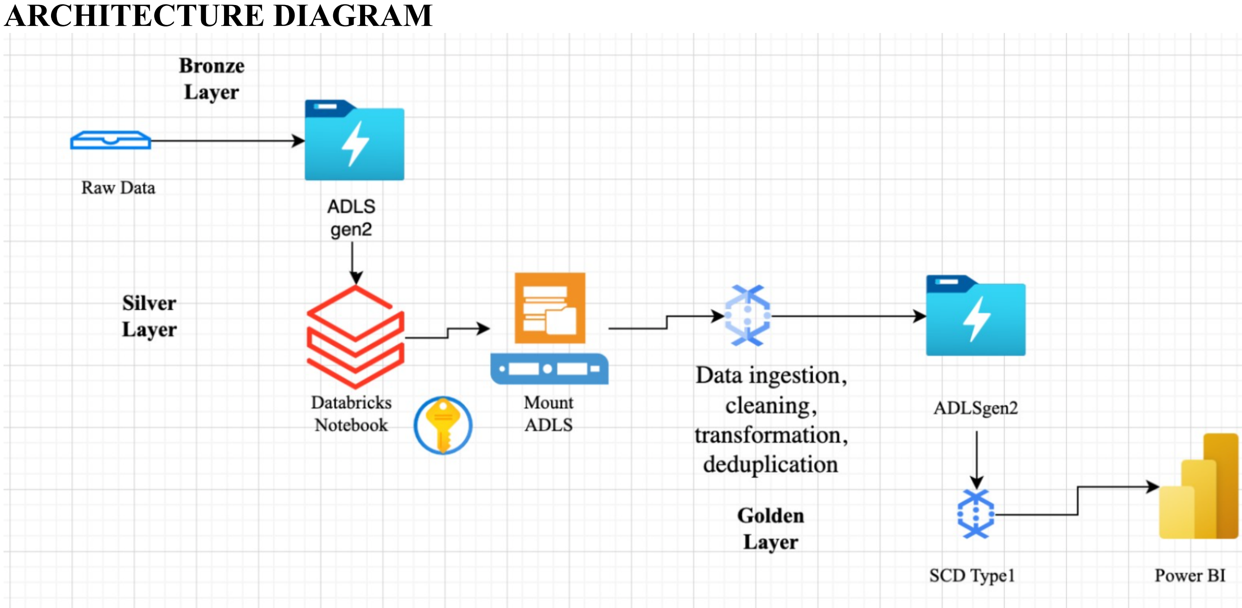
Design and implement a robust, scalable data pipeline for processing customer account and loan data using modern Azure services:

- Ingest data from **ADLS Gen2 (Bronze Layer)**
- Mount storage using **Service Principal Authentication**
- Clean and transform using **Databricks Notebooks (Silver Layer)**
- Create **Delta Tables with SCD Type 1 (Gold Layer)**
- Enable downstream analytics and **visualization in Power BI**

Tools and Technologies Used

- **Azure Data Lake Storage Gen2 (ADLS Gen2)**
- **Service Principal** for secure authentication.
- **Azure Key Vault**
- **Azure Databricks (PySpark and SQL)**
- **Delta Lake**
- **Power BI**

ARCHITECTURE DIAGRAM



Project Steps

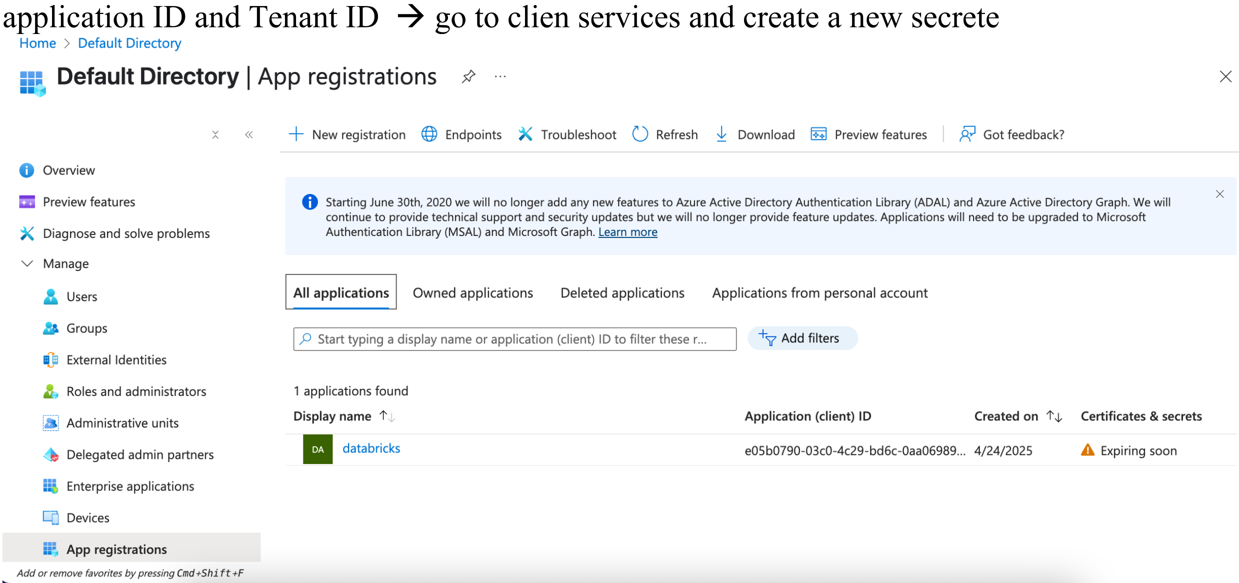
Step 1: Data Ingestion (Bronze Layer)

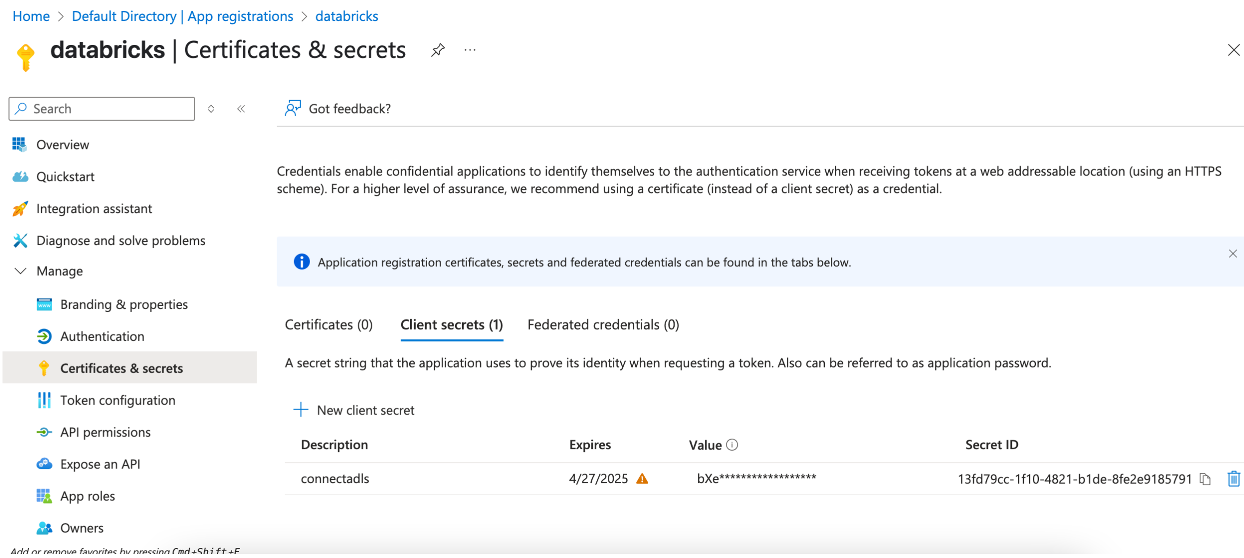
- **Source Files:**
  - `accounts.csv`
  - `customers.csv`
  - `loan_payments.csv`
  - `loans.csv`
  - `transactions.csv`
- **Sink:**
  - ADLS Gen2 Raw (Bronze) Container
- **Reference:**
  - [Kaggle AI Bank Dataset]

Step 2: Mounting Storage & Data Cleaning (Silver Layer)

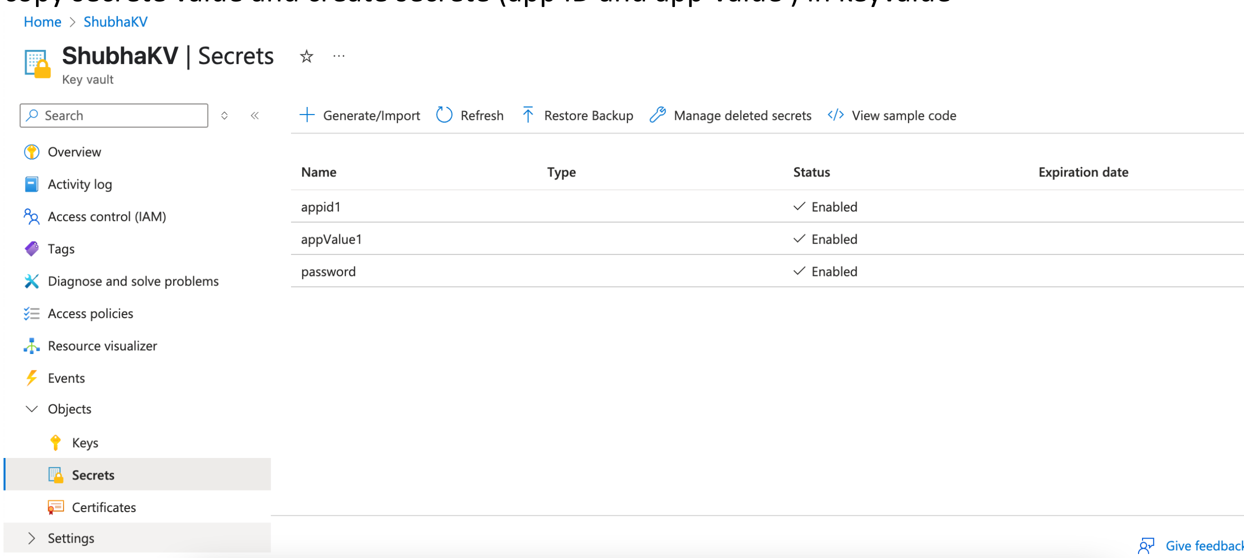
- **Mount ADLS Gen2 Storage** to Databricks using Service Principal credentials stored in Azure Key Vault

Microsoft Entra ID →Manage → App Registration →Create a service Principle → copy application ID and Tenant ID → go to clien services and create a new secrete

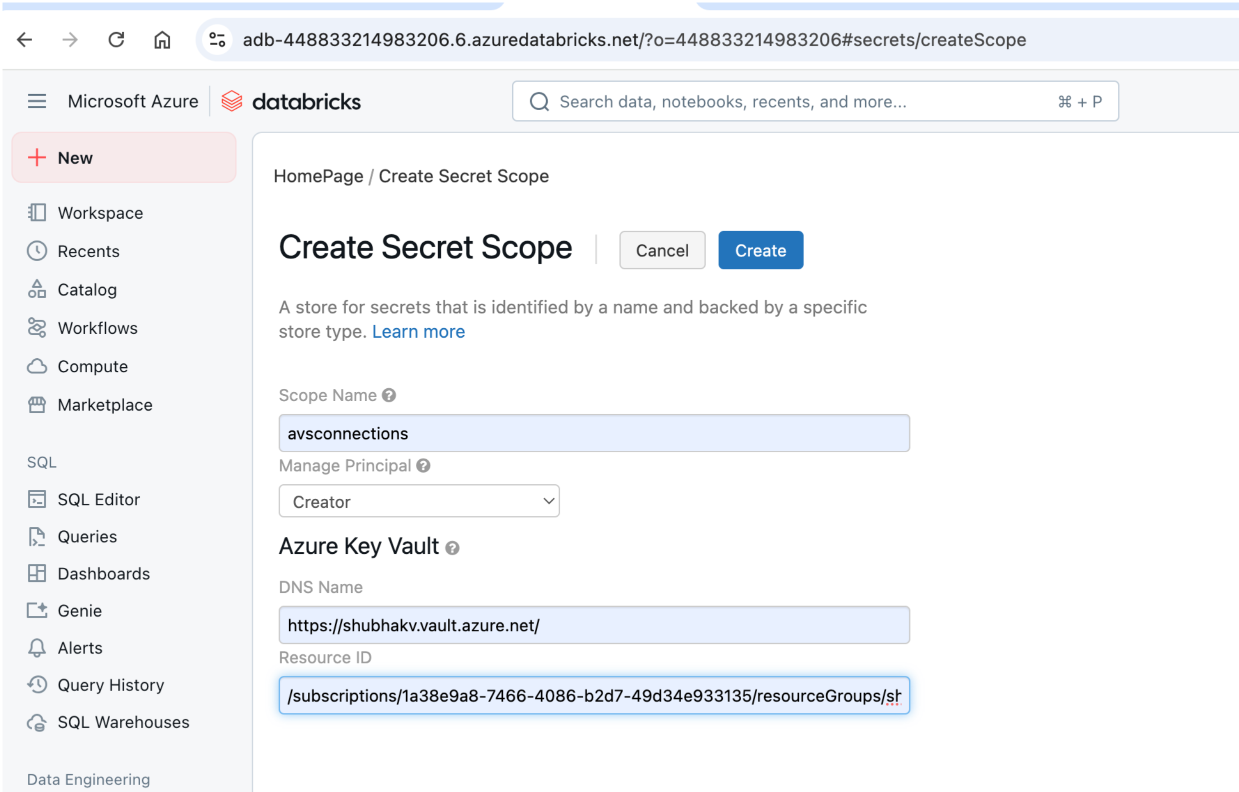




copy secrete value and create secrete (app ID and app Value) in keyvalue



Create a scope in databricks



Now create a mount point in adls gen2

```
configs = {
    "fs.azure.account.auth.type": "OAuth",
    "fs.azure.account.oauth.provider.type": "org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
    "fs.azure.account.oauth2.client.id": dbutils.secrets.get(scope="avsconnections", key="appid1"),
    "fs.azure.account.oauth2.client.secret": dbutils.secrets.get(scope="avsconnections", key="appValue1"),
    "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/f9087d13-42f9-4a25-8d93-312ee56664c2/oauth2/token"
}

dbutils.fs.mount(
    source="abfss://edw@shubhaadlsgen2.dfs.core.windows.net/",
    mount_point="/mnt/edw",
    extra_configs=configs
)
```

Read the raw CSV files into Spark DataFrames.

Remove poor quality data and prepare consistent Parquet files.

Actions Taken:

- Dropped all **null values** (.na.drop()) to avoid incomplete records.
- Removed **duplicate records** (.dropDuplicates()).
- Saved each cleaned dataset as **Parquet** into the **Silver layer** (/mnt/edw/silver/...).

21 hours ago (23s)

```
# Define function to clean and write a DataFrame
def clean_and_write_csv(input_path, output_path):
    df = spark.read.format("csv").option("header", "true").load(input_path)
    df = df.na.drop().dropDuplicates()
    df.write.mode("overwrite").format("parquet").save(output_path)

# Process each dataset from Bronze → Silver
clean_and_write_csv("/mnt/edw/bronze_folder/accounts.csv", "/mnt/edw/silver/Accounts")
clean_and_write_csv("/mnt/edw/bronze_folder/customers.csv", "/mnt/edw/silver/Customers")
clean_and_write_csv("/mnt/edw/bronze_folder/transactions.csv", "/mnt/edw/silver/Transactions")
clean_and_write_csv("/mnt/edw/bronze_folder/loans.csv", "/mnt/edw/silver/Loans")
clean_and_write_csv("/mnt/edw/bronze_folder/loan_payments.csv", "/mnt/edw/silver/Loan_Payments")
```

(15) Spark Jobs

Joining Datasets and Creating a Master Data Frame

Create a combined view containing necessary customer + account + transaction + loan + payment information.

Actions Taken:

- Performed multiple joins:
  - Joined Accounts and Customers on customer_id
  - Joined the result with Transactions on account_id
  - Joined further with Loans on customer_id
  - Finally joined with Loan Payments on loan_id
- Selected only required columns:
  - account_id, transaction_id, customer_id, loan_id, payment_id, amount, date
- Removed duplicates on the combination of selected fields.

```python
from pyspark.sql.functions import col
# Step 1: Read cleaned data from Silver (Parquet)
df_account = spark.read.parquet("/mnt/edw/silver/Accounts")
df_customer = spark.read.parquet("/mnt/edw/silver/Customers")
df_transactions = spark.read.parquet("/mnt/edw/silver/Transactions")
df_loans = spark.read.parquet("/mnt/edw/silver/Loans")
df_loan_payments = spark.read.parquet("/mnt/edw/silver/Loan_Payments")

# Step 2: Join DataFrames (use proper keys based on schema)
df_join1 = df_account.join(df_customer, on="customer_id", how="left")
df_join2 = df_join1.join(df_transactions, on="account_id", how="left")
df_join3 = df_join2.join(df_loans, on="customer_id", how="left")
final_df = df_join3.join(df_loan_payments, on="loan_id", how="left")

# Step 3: Select required columns and remove duplicates
final_selected = final_df.select(
    col("account_id").cast("int"),
    col("transaction_id").cast("int"),
    col("customer_id").cast("int"),
    col("loan_id").cast("int"),
    col("payment_id").cast("int"),
    col("balance").cast("float"),
    col("transaction_date").cast("timestamp"),
    col("transaction_amount").cast("float"),
    col("loan_amount").cast("float"),
    col("payment_amount").cast("float"),
    col("payment_date").cast("timestamp")
).dropDuplicates()
```

Store the enriched, trusted master dataset into the **Gold Layer** in **Delta** format.

```python
# Step 4: Write final DataFrame to Gold layer in Delta format
final_selected.write.mode("overwrite").format("delta").save("/mnt/edw/silver/Customer_Account_Loan_Data")
```

SCD Type 1 Implementation (Slowly Changing Dimension)

Goal:

Maintain the Gold Delta Tables by handling changes (update old records, insert new ones).

Actions Taken:

- Created a hash_key (using CRC32) on all columns to detect changes easily.
- Used **Delta Lake MERGE** operation:
    - **When Matched** → Update the record if changed.
    - **When Not Matched** → Insert as new record.
- Managed audit columns:
    - created_date, created_by

- updated_date, updated_by

Create a Delta Table (if not exists)

- Creates a **Delta table** for customer data.

Stores data in **Delta format**, enabling ACID transactions & time travel.

- 21 hours ago (3s) 7

```
%sql

create table if not exists delta.`/mnt/edw/gold/Accounts/`(
  account_id int,
  customer_id int,
  account_type string,
  balance float,
  hash_key bigint,
  created_by string,
  created_date timestamp,
  updated_by string,
  updated_date timestamp
)
```

- Define Source and Target path

21 hours ago (<1s) 9

```
from pyspark.sql.functions import *
src_path="/mnt/edw/silver/Accounts"
print(src_path)
tgt_path="/mnt/edw/gold/Accounts/"
print(tgt_path)
```

```
/mnt/edw/silver/Accounts
/mnt/edw/gold/Accounts/
```

read the data

21 hours ago (1s) 10

```
df_src=spark.read.format("parquet").option("header", "true").option("inferSchema", "true").load(src_path)
display(df_src)
```

(2) Spark Jobs

Generate hashkey

21 hours ago (<1s) 11

```
df_src1=df_src.withColumn("hash_key",crc32(concat(*df_src.columns)))
display(df_src1)
```

(1) Spark Jobs

df_src1: pyspark.sql.dataframe.DataFrame = [account_id: string, customer_id: string ... 3 more fields]

Table

	account_id	customer_id	account_type	balance	hash_key
1	1	45	Savings	1000.50	4261402674
2	48	6	Checking	4900.00	3216528439
3	21	53	Savings	300.25	2800704470
4	29	58	Savings	75.25	3637748452
5	34	41	Checking	3500.50	868746038
6	35	62	Savings	175.75	3427479153
7	5	56	Savings	500.00	2866974084
8	78	4	Checking	7900.50	2650882798

load data and perform anti join

21 hours ago (1s) 13 Python

```
df_src1=df_src1.alias("src").join(dbtable.toDF().alias("tgt"), ((col("src.account_id") == col("tgt.account_id")) & (col("src.hash_key") == col("tgt.hash_key"))), "anti").select(col("src.*"))
df_src1.show()
```

(1) Spark Jobs

Merge Changes into the Delta Table

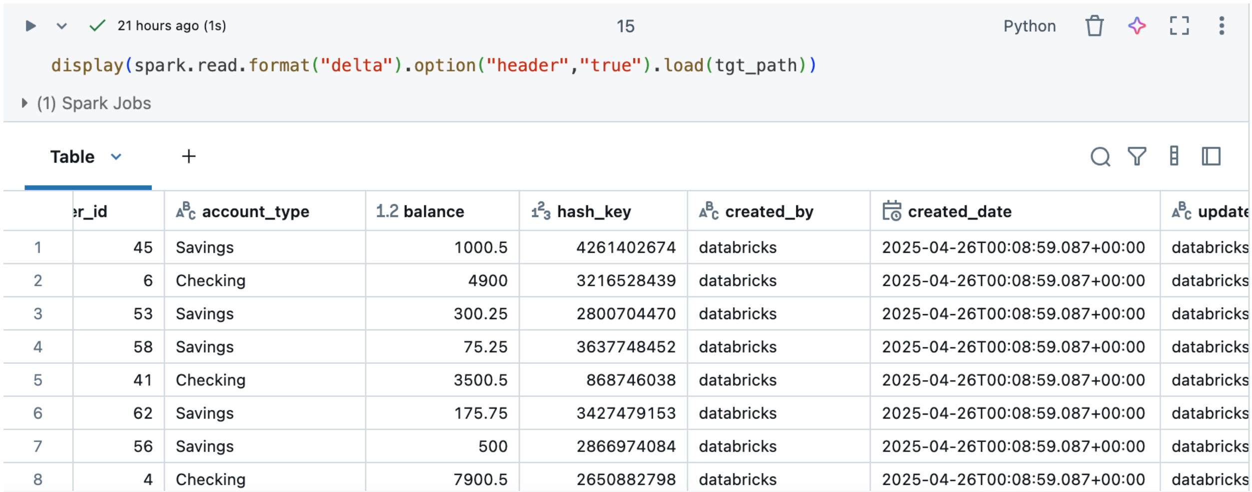
- **Merges new and updated records into Delta table.**
- **Updates existing records** where `ID` matches.
- **Inserts new records** where no match is found.

21 hours ago (6s) 14 Python

```
dbtable.alias("tgt").merge(df_src1.alias("src"),"tgt.account_id = src.account_id")\
    .whenMatchedUpdate(set={"tgt.account_id":"src.account_id","tgt.customer_id":"src.customer_id","tgt.account_type":"src.account_type","tgt.balance":"src.balance","tgt.hash_key":"src.hash_key","tgt.updated_date":current_timestamp(),"tgt.updated_by":lit("databricks_Updated")})\
        .whenNotMatchedInsert(values={"tgt.account_id":"src.account_id","tgt.customer_id":"src.customer_id","tgt.account_type":"src.account_type","tgt.balance":"src.balance","tgt.hash_key":"src.hash_key","tgt.created_date":current_timestamp(),"tgt.created_by":lit("databricks"),"tgt.updated_date":current_timestamp(),"tgt.updated_by":lit("databricks")}).execute()
```

(7) Spark Jobs

display the updated record



	er_id	account_type	balance	hash_key	created_by	created_date	update
1	45	Savings	1000.5	4261402674	databricks	2025-04-26T00:08:59.087+00:00	databricks
2	6	Checking	4900	3216528439	databricks	2025-04-26T00:08:59.087+00:00	databricks
3	53	Savings	300.25	2800704470	databricks	2025-04-26T00:08:59.087+00:00	databricks
4	58	Savings	75.25	3637748452	databricks	2025-04-26T00:08:59.087+00:00	databricks
5	41	Checking	3500.5	868746038	databricks	2025-04-26T00:08:59.087+00:00	databricks
6	62	Savings	175.75	3427479153	databricks	2025-04-26T00:08:59.087+00:00	databricks
7	56	Savings	500	2866974084	databricks	2025-04-26T00:08:59.087+00:00	databricks
8	4	Checking	7900.5	2650882798	databricks	2025-04-26T00:08:59.087+00:00	databricks