Enterprise Data Cleaning & ETL Orchestration Framework using Python

TEAM 3

SANJAY.S.D

TABLE OF CONTENTS

SL.NO	TITLE	PAGE NO
1	ABSTRACT	3
2	INTRODUCTION	4
3	Sprint 1 – Environment Setup & Pipeline Design	6
4	Sprint 2 – Data Cleaning + Lookup + Fuzzy Matching + SCD Type 2	14
5	Sprint 3 – SQL Transformation Layer	20
6	Sprint 4 – External API Data Ingestion Pipeline	30
7	Sprint 5 – Metadata & Pipeline Audit Framework	39
8	Sprint 6 – Versioning Framework	49
9	Sprint 7 – Dashboard & Monitoring UI	58
10	Sprint 8 – Performance Optimization & Scaling	66
11	CONCLUSION	71

ABSTRACT

This project implements an **Enterprise-Level ETL Orchestration Framework** using **Apache Airflow, MySQL, Docker, and Python**. The objective of this framework is to design and execute a structured data pipeline that processes raw Amazon e-commerce data into curated and analytics-ready datasets.

The pipeline follows a layered architecture:

- **RAW Layer** – stores original source data
- **STAGING Layer** – applies cleaning and standardization
- **CURATED Layer** – business-ready transformed dataset
- **DIMENSION Layer** – implements Slowly Changing Dimensions (SCD Type 2)

The framework also includes:

- SQL-based transformation scheduling
- API-based ingestion pipelines
- Metadata logging and execution tracking
- Dataset and schema versioning
- Dashboard monitoring using Streamlit
- Performance tuning for large datasets

This project provides a complete real-time, scalable, and production-grade ETL system suitable for enterprise environments.

INTRODUCTION

Overview

This project implements an end-to-end enterprise ETL orchestration framework using Apache Airflow as the workflow scheduler and MySQL as the storage engine. The pipeline follows a layered architecture where raw extracted data is stored in the RAW layer, cleaned and standardized data is stored in the STAGING layer, and fully transformed business-ready data is loaded into the CURATED layer. A Slowly Changing Dimension (Type 2) process is implemented to maintain historical category changes over time. Additional modules include SQL transformation schedulers, validation checks for row count consistency, duplicate detection, API ingestion support, metadata logging for audit tracking, versioning frameworks for governance, and a monitoring dashboard interface to visualize run history, trends, and data quality metrics. The complete project is implemented sprint-wise across tasks ranging from environment setup up to performance tuning and optimization.

Problem Statement

In modern organizations, large volumes of data are generated every day through different business systems such as e-commerce platforms, customer transactions, product records, and external services. However, the collected raw data is often inconsistent, duplicated, incomplete, and unstructured, which makes it difficult to directly use for analytics or decision-making. Many companies face challenges such as manual data handling, lack of automation, poor data quality, absence of proper pipeline monitoring, and difficulty in maintaining historical changes in dimension data. Additionally, when processing large datasets, performance bottlenecks occur due to inefficient loading, transformations, and query execution. Therefore, there is a strong requirement for an enterprise-level ETL framework that can orchestrate the complete data pipeline with automation, validation, metadata tracking, version control, and scalability.

Objectives

The primary objective of this internship project is to design and implement a robust enterprise ETL orchestration framework that follows industry best practices. The project aims to build an automated pipeline using Apache Airflow to manage structured ETL flow from raw ingestion to curated output. It focuses on implementing data cleaning, standardization, lookup mapping, fuzzy matching, and Slowly Changing Dimension Type 2 logic to track historical changes. The framework also aims to support SQL-based transformation layers with scheduling and validation, integrate external API ingestion with pagination and retry logic, and create metadata logging for task-level and DAG-level audit tracking. Furthermore, the project introduces dataset, schema, and transformation versioning mechanisms along with dashboard monitoring and performance optimization strategies for handling large-scale enterprise data.

Scope of the Project

The scope of this project includes developing a complete enterprise-ready ETL pipeline architecture structured across multiple layers such as RAW, STAGING, and CURATED. The project covers ingestion of Amazon dataset from CSV files, cleaning transformations, category standardization, and SCD Type 2 implementation for dimension tables. It also includes SQL transformation scripts, scheduling through Airflow, API ingestion pipelines, metadata tracking tables, execution history, and run monitoring dashboards. Version control is applied to datasets, scripts, schemas, and transformations to ensure governance and rollback readiness. However, the project does not extend into cloud deployment, real-time streaming frameworks, or integration with large-scale data warehouses, as it focuses primarily on building a strong foundation for enterprise ETL orchestration in a local containerized environment.

IMPLEMENTATION OF SPRINTS

SPRINT 1- Environment Setup & Pipeline Design

Task 1: Environment Setup & Pipeline Design

Objective

The objective of this task is to:

- Install Apache Airflow using Docker
- Configure airflow services (webserver + scheduler)
- Design ETL pipeline layers:
  RAW → STAGING → CURATED
- Create the base folder structure for the project

Practical Implementation

Step 1: Install Required Software

Before starting the project, the following tools were installed:

Required Tools

- Python 3.8+
- Docker Desktop
- MySQL Workbench
- GitHub (Version Control)
- VS Code / PyCharm for development

Step 2: Configure Airflow using Docker Compose

To make the project enterprise-ready, Apache Airflow was run inside Docker containers.

Create a file in a project folder and name the file:docker-compose.yaml

Docker Compose Code

```
version: "3.8"
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
  redis:
    image: redis:7
  webserver:
    image: apache/airflow:2.8.3
    environment:
      AIRFLOW__CORE__EXECUTOR: LocalExecutor

      _PIP_ADDITIONAL_REQUIREMENTS: >
        mysql-connector-python pandas rapidfuzz
    volumes:
      - ../dags:/opt/airflow/dags
      - ../scripts:/opt/airflow/scripts
      - ../data:/opt/airflow/data
```

```
    ports:
      - "8080:8080"
    command: webserver
  scheduler:
    image: apache/airflow:2.8.3
    environment:
      AIRFLOW__CORE__EXECUTOR: LocalExecutor
      _PIP_ADDITIONAL_REQUIREMENTS: >
        mysql-connector-python pandas rapidfuzz
   volumes:
      - ../dags:/opt/airflow/dags
      - ../scripts:/opt/airflow/scripts
      - ../data:/opt/airflow/data
    command: schedulerversion: "3.8"
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: airflow
      POSTGRES_PASSWORD: airflow
      POSTGRES_DB: airflow
  redis:
    image: redis:7
```

```
webserver:
  image: apache/airflow:2.8.3
  environment:
    AIRFLOW__CORE__EXECUTOR: LocalExecutor

    _PIP_ADDITIONAL_REQUIREMENTS: >
      mysql-connector-python pandas rapidfuzz

  volumes:
    - ../dags:/opt/airflow/dags
    - ../scripts:/opt/airflow/scripts
    - ../data:/opt/airflow/data
  ports:
    - "8080:8080"
  command: webserver
scheduler:
  image: apache/airflow:2.8.3
  environment:
    AIRFLOW__CORE__EXECUTOR: LocalExecutor
    _PIP_ADDITIONAL_REQUIREMENTS: >
      mysql-connector-python pandas rapidfuzz
  volumes:
    - ../dags:/opt/airflow/dags
```

```
    - ../scripts:/opt/airflow/scripts
    - ../data:/opt/airflow/data
  command: scheduler
```

Step 3: Start Airflow Services

After configuration, Docker containers were started using:

```
docker compose up -d
```

This launches:

- Airflow Webserver
- Scheduler
- Postgres metadata DB
- Redis

Output Verification

Airflow UI was accessed successfully in browser:

http://localhost:8080

Step 4: Create ETL Folder Structure

A standard enterprise ETL project structure was created:

Project Folder Structure:

```
Airflow/
│
├── dags/
│   └── amazon_etl_pipeline.py
│
├── scripts/
```

```
│   ├── csv_to_raw.py
│   ├── raw_to_staging.py
│   ├── staging_to_curated.py
│   ├── validate_pipeline.py
│   └── fuzzy_mapper.py
│
├── data/
│   ├── raw/
│   │   └── Amazon.csv
│   └── staging/
│
├── data_models/
│   ├── raw_tables.sql
│   ├── staging_tables.sql
│   └── curated_tables.sql
│
└── docker/
    └── docker-compose.yaml
```

Step 5:SQL

Create tables for raw,staging and curated.

Step 6: Create Airflow DAG

DAG Code

import sys

```
sys.path.append("/opt/airflow/scripts")

from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

from csv_to_raw import csv_to_raw
from raw_to_staging import raw_to_staging
from staging_to_curated import staging_to_curated
from validate_pipeline import validate_pipeline

default_args = {
    "owner": "sanjay",
    "retries": 1,
    "retry_delay": timedelta(minutes=2)
}

with DAG(
    dag_id="amazon_etl_pipeline",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False
) as dag:
```

```
task1 = PythonOperator(
    task_id="csv_to_raw",
    python_callable=csv_to_raw
)

task2 = PythonOperator(
    task_id="raw_to_staging",
    python_callable=raw_to_staging
)

task3 = PythonOperator(
    task_id="staging_to_curated",
    python_callable=staging_to_curated
)

task4 = PythonOperator(
    task_id="validate_pipeline",
    python_callable=validate_pipeline
)

task1 >> task2 >> task3 >> task4
```

SPRINT 2- Data Cleaning + Lookup + Fuzzy Matching + SCD Type 2

Overview

Sprint 2 focuses on improving data quality and standardization.

Sprint 2 enhances the pipeline by adding:

- Lookup tables
- Category cleaning rules
- Fuzzy matching for incorrect text values
- Slowly Changing Dimension (Type 2) implementation

Set up Lookup Tables (location, category, codes)

SQL CODE

```
CREATE TABLE lkp_category (
    source_category VARCHAR(100),
    standardized_category VARCHAR(100),
    is_active BOOLEAN DEFAULT TRUE
);

INSERT INTO lkp_category VALUES
('HOME & KITCHEN', 'Home & Kitchen', TRUE),
('BOOKS', 'Books', TRUE),
('ELECTRONICS', 'Electronics', TRUE),
('SPORTS', 'Sports & Outdoors', TRUE),
('TOYS', 'Toys & Games', TRUE),
('CLOTHING', 'Clothing', TRUE);
```

To check output run the query in mysql

SELECT * FROM lkp_category;

OUTPUT

source_category	standardized_category
BOOKS	Books
ELECTRONICS	Electronics

Implement Standardization (upper/lowercase rules)

In script folder in raw to staging file add the below code

```
cursor.execute("""
INSERT INTO stg_amazon (
    order_id, order_date, customer_id, customer_name,
    product_id, product_name, category, brand,
    quantity, unit_price, discount, tax,
    shipping_cost, total_amount, payment_method,
    order_status, city, state, country, seller_id
)
SELECT
    order_id,
    order_date,
```

```
    customer_id,
    customer_name,
    product_id,
    product_name,
    UPPER(category),   -- Standardize category
    brand,
    quantity,
    unit_price,
    discount,
    tax,
    shipping_cost,
    total_amount,
    payment_method,
    order_status,
    city,
    state,
    country,
    seller_id
FROM raw_amazon;
""")
```

To get the output run the query

```
SELECT DISTINCT category FROM stg_amazon;
```

OUTPUT

BOOKS

HOME & KITCHEN

ELECTRONICS

Fuzzy Matching for Incorrect Labels

In script folder create the file and named fuzzy_mapper.py

Code:

```
from rapidfuzz import process

def fuzzy_match(value, standard_list):
    match, score, _ = process.extractOne(value, standard_list)

    if score >= 80:
        return match
    return None
```

Implement SCD Type 2 Logic

SQL Code:

```
CREATE TABLE dim_category (
    category_key INT AUTO_INCREMENT PRIMARY KEY,
    category_name VARCHAR(100),
    start_date DATE,
```

```
    end_date DATE,
    is_current BOOLEAN,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Build SCD Category Load Script

Write the code in scd_category.py file

```
import mysql.connector
from datetime import date
def scd_category_load():
    conn = mysql.connector.connect(
        host="host.docker.internal",
        user="airflow_user",
        password="airflow_pass",
        database="etl_project"
    )
    cursor = conn.cursor(dictionary=True)
    cursor.execute("SELECT DISTINCT category FROM cur_amazon")
    categories = cursor.fetchall()
    for row in categories:
        category = row["category"]
        cursor.execute("""
        SELECT * FROM dim_category
        WHERE category_name=%s AND is_current=1
```

```
        """, (category,))

        existing = cursor.fetchone()

        if not existing:
            cursor.execute("""
            INSERT INTO dim_category
            (category_name, start_date, end_date, is_current)
            VALUES (%s, %s, NULL, 1)
            """, (category, date.today()))

    conn.commit()
    cursor.close()
    conn.close()

    print("SCD Category Dimension Updated Successfully")
```

Output

```
SELECT * FROM dim_category;
```

SPRINT 3- SQL Transformation Layer

Sprint 3 focuses on implementing the transformation layer fully in SQL.

- SQL-based transformation templates
- CTE-based analytics transformation
- Airflow SQL scheduler execution
- Output validation checks

Write SQL Transformations for STAGING → CURATED

SQL Code

```
-- Step 1: Clear Curated Table

TRUNCATE TABLE cur_amazon;

-- Step 2: Load Clean Data into Curated Layer

INSERT INTO cur_amazon (

    order_id,

    customer_id,

    product_id,

    category,

    seller_id,

    order_date,

    load_date,

    quantity,

    unit_price,

    discount,

    tax,

    shipping_cost,
```

```
    total_amount,
    payment_method,
    order_status,
    city,
    state,
    country,
    is_current
)
SELECT
    order_id,
    customer_id,
    product_id,
    --  Category Normalization
    CONCAT(
        UPPER(LEFT(category, 1)),
        LOWER(SUBSTRING(category, 2))
    ) AS category,
    seller_id,
    order_date,
    CURRENT_DATE AS load_date,
   quantity,
   unit_price,
    discount,
    tax,
```

```
    shipping_cost,
    Correct Total Amount Calculation
    (quantity * unit_price) - discount + tax + shipping_cost AS total_amount,
    payment_method,
    order_status,
    city,
    state,
    country,
    1 AS is_current
FROM stg_amazon;
```

Output Verification

Check row counts:

```
SELECT COUNT(*) FROM stg_amazon;
SELECT COUNT(*) FROM cur_amazon;
```

Use CTEs and Analytical Queries

SQL Code Using CTE

```
WITH base_data AS (
    SELECT
        order_id,
        customer_id,
        product_id,
        quantity,
        unit_price,
```

```
        discount,
        tax,
        shipping_cost,
        category,
        seller_id,
        order_date,
        payment_method,
        order_status,
        city,
        state,
        country
    FROM stg_amazon
)

INSERT INTO cur_amazon (
    order_id,
    customer_id,
    product_id,
    category,
    seller_id,
    order_date,
    load_date,
    quantity,
    unit_price,
```

```
    discount,
    tax,
    shipping_cost,
    total_amount,
    payment_method,
    order_status,
    city,
    state,
    country,
    is_current
)
SELECT
    order_id,
    customer_id,
    product_id,
    category,
    seller_id,
    order_date,
    CURRENT_DATE,
    quantity,
    unit_price,
    discount,
    tax,
    shipping_cost,
```

```
    --  Derived Metric
    (quantity * unit_price) - discount + tax + shipping_cost,

    payment_method,
    order_status,
    city,
    state,
    country,
    1
FROM base_data;
```

CTE Output Validation

```
SELECT COUNT(*) FROM cur_amazon;
```

Create SQL Templates

Folder Structure

```
Airflow/
└── data_models/
    ├── raw_to_stg_transform.sql
    ├── stg_to_cur_transform.sql
    ├── validation_checks.sql
```

Raw_to_stg_transform.sql

```
TRUNCATE TABLE stg_amazon;
INSERT INTO stg_amazon
SELECT * FROM raw_amazon;
```

Validation_checks.sql

```
-- Row count checks
SELECT COUNT(*) AS raw_count FROM raw_amazon;
SELECT COUNT(*) AS staging_count FROM stg_amazon;
SELECT COUNT(*) AS curated_count FROM cur_amazon;

-- Duplicate check
SELECT order_id, product_id, COUNT(*) AS cnt
FROM cur_amazon
GROUP BY order_id, product_id
HAVING COUNT(*) > 1;

-- Null check
SELECT COUNT(*) AS null_orders
FROM cur_amazon
WHERE order_id IS NULL;
```

Implement Transformation Scheduler in Airflow

Create a new python file and named run_sql.py

```
import mysql.connector
def run_sql_file(file_path):
    conn = mysql.connector.connect(
        host="host.docker.internal",
        user="airflow_user",
        password="airflow_pass",
        database="etl_project"
    )
    cursor = conn.cursor()
    with open(file_path, "r") as f:
        sql_script = f.read()
    statements = sql_script.split(";")
    for stmt in statements:
        stmt = stmt.strip()
        if stmt:
            cursor.execute(stmt)
    conn.commit()
    cursor.close()
    conn.close()

    print(" SQL File Executed Successfully")
```

Airflow DAG Scheduler File

Create dag file and name :amazon_sql_scheduler_dag.py

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime
from run_sql_file import run_sql_file
with DAG(
    dag_id="amazon_sql_transformation_scheduler",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False
) as dag:
    transform_task = PythonOperator(
        task_id="run_staging_to_curated_sql",
        python_callable=run_sql_file,
        op_args=["/opt/airflow/data_models/stg_to_cur_transform.sql"]
    )
    validate_task = PythonOperator(
        task_id="validate_sql",
        python_callable=run_sql_file,
        op_args=["/opt/airflow/data_models/validation_checks.sql"]
    )
```

```
transform_task >> validate_task
```

Validate Output Row Counts

Required Verification Queries

Row Count Match

```
SELECT COUNT(*) FROM stg_amazon;
SELECT COUNT(*) FROM cur_amazon;
```

Duplicate Check

```
SELECT order_id, product_id, COUNT(*)
FROM cur_amazon
GROUP BY order_id, product_id
HAVING COUNT(*) > 1;
```

Expected Output:

0 rows returned

Null Check

```
SELECT COUNT(*) FROM cur_amazon
WHERE order_id IS NULL;
```

Output

0

SPRINT 4- External API Data Ingestion Pipeline

Sprint 4 extends your ETL pipeline by implementing:

- API ingestion
- Pagination handling
- Rate-limiting support
- Authentication support
- Retry mechanism
- Store raw JSON into staging
- Load cleaned API data into MySQL

Extracting Data from External APIs

Objective

Fetch product/order data from an external REST API and ingest it into Airflow pipeline.

Create a python file and named api_ingestion.py for api ingestion

Code (API Extraction)

```
import requests

import time

import json

BASE_URL = "https://fakestoreapi.com/products"

API_KEY = None   # optional

def call_api(url, params=None):

    headers = {}
```

```
    if API_KEY:
        headers["Authorization"] = f"Bearer {API_KEY}"
    response = requests.get(url, headers=headers, params=params)
    response.raise_for_status()
    return response.json()
def run_api_ingestion():
    data = call_api(BASE_URL)
    save_path = "/opt/airflow/data/staging/api_raw.json"
    with open(save_path, "w") as f:
        json.dump(data, f, indent=4)
    print(f"API JSON Saved → {save_path}")
    print(f"Total Records: {len(data)}")
if __name__ == "__main__":
    run_api_ingestion()
```

To verify the output run the command in terminal

docker compose exec webserver python /opt/airflow/scripts/api_ingestion.py

Output

API JSON Saved → /opt/airflow/data/staging/api_raw.json

Total Records: 20

Pagination + Rate Limiting

Objective

- Enterprise APIs return data in pages.
- So we implemented pagination loop + delay.

CODE

```
def fetch_paginated(endpoint):
    page = 1
    page_size = 20
    all_data = []
    while True:
        params = {"page": page, "limit": page_size}
        data = call_api(endpoint, params)
        if not data or len(data) == 0:
            break
        all_data.extend(data)
        print(f"Fetched page {page} — {len(data)} records")
        page += 1
        #  Rate limiting delay
        time.sleep(1)
    return all_data
```

Output

Fetched page 1 — 20 records

Fetched page 2 — 20 records

Fetched page 3 — 20 records

...

API Authentication (Bearer / API Key / OAuth)

Objective

Real APIs require authentication headers.

I have added Bearer Token Support

Authentication Logic

```
API_KEY = "your_real_api_key_here"
headers = {
    "Authorization": f"Bearer {API_KEY}"
}
```

If API Uses API Key Instead

```
headers = {
    "x-api-key": API_KEY
}
```

OAuth Token Flow (Enterprise)

```
response = requests.post(
    "https://api.site.com/oauth/token",
    data={"client_id": "...", "secret": "..."}
)
token = response.json()["access_token"]
```

Then:

```
headers["Authorization"] = f"Bearer {token}"
```

Retry Logic for Failed API Calls

Objective

Production pipelines must survive failures like:

- Server 500 errors
- Network timeout

- Rate-limit (429)

Retry + Rate Handling Code

```
if response.status_code == 429:
    print(" ⏳ Rate limited… waiting 5 seconds")
    time.sleep(5)
    return call_api(url, params)
if response.status_code >= 500:
    print("⚠ Server error… retrying in 3 seconds")
    time.sleep(3)
    return call_api(url, params)
```

This ensures pipeline reliability.

Store API Raw JSON in Staging

Objective

Raw API responses must be stored in staging before cleaning.

File Location

data/staging/api_raw.json

Verification Command

```
docker compose exec webserver head /opt/airflow/data/staging/api_raw.json
```

Output:


```
[
  {
    "id": 1,
    "title": "Fjallraven Backpack",
    "price": 109.95,
```

```
    "category": "men's clothing"
  }
]
```

Cleaning + Load into MySQL

Task: Transform API JSON → CSV

scripts/api_clean_transform.py

CODE

```
import pandas as pd
import json
def clean_api_data():
    input_path = "/opt/airflow/data/staging/api_raw.json"
    output_path = "/opt/airflow/data/staging/api_clean.csv"
    with open(input_path) as f:
        data = json.load(f)
    df = pd.DataFrame(data)
    df = df[["id", "title", "category", "price", "description", "image"]]
    df.to_csv(output_path, index=False)
    print(" Clean API CSV Created")
```

Load into MySQL

cripts/api_load_to_mysql.py

CODE

```
import mysql.connector
import pandas as pd
```

```
def load_api_clean_to_mysql():
    df = pd.read_csv("/opt/airflow/data/staging/api_clean.csv")
    conn = mysql.connector.connect(
        host="host.docker.internal",
        user="airflow_user",
        password="airflow_pass",
        database="etl_project"
    )
    cursor = conn.cursor()
    insert_sql = """
        REPLACE INTO api_products_raw
        (id, title, category, price, description, image_url)
        VALUES (%s, %s, %s, %s, %s, %s)
    """
    for _, row in df.iterrows():
        cursor.execute(insert_sql, (
            int(row["id"]),
            row["title"],
            row["category"],
            float(row["price"]),
            row["description"],
            row["image"]
        ))
    conn.commit()
```

```
    cursor.close()
    conn.close()
print("API Data Loaded into MySQL")
```

MySQL Verification

```
SELECT COUNT(*) FROM api_products_raw;
```

Output:

20

Airflow API DAG Integration

File name: dags/api_pipeline_dag.py

CODE

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime
from api_ingestion import run_api_ingestion
from api_clean_transform import clean_api_data
from api_load_to_mysql import load_api_clean_to_mysql
with DAG(
    dag_id="api_data_pipeline",
    start_date=datetime(2025, 1, 1),
    schedule_interval=None,
    catchup=False
) as dag:
    ingest = PythonOperator(
        task_id="api_ingestion",
```

```
    python_callable=run_api_ingestion
)
clean = PythonOperator(
    task_id="api_cleaning",
    python_callable=clean_api_data
)
load = PythonOperator(
    task_id="api_load_mysql",
    python_callable=load_api_clean_to_mysql
)
ingest >> clean >> load
```

Sprint 5 - Metadata & Pipeline Audit Framework

Overview

Enterprise pipelines must also provide:

- Monitoring
- Audit logs
- Step-Level Row Counts
- Execution History
- Metadata Query APIs

So Sprint 5 builds a complete **Metadata Tracking System**.

Create ETL Metadata Tables

Objective

Store ETL execution audit data inside MySQL.

Enterprise teams need to track:

- When pipeline ran
- Which task ran
- Success or failure
- Rows processed
- Error logs

Tables Created in MySQL

File: data_models/metadata_tables.sql

SQL Code

```
-- DAG Run Log Table
CREATE TABLE etl_dag_run_log (
    run_id INT AUTO_INCREMENT PRIMARY KEY,
    dag_id VARCHAR(200),
```

```
    start_time DATETIME,
    end_time DATETIME,
    status VARCHAR(50),
    records_processed INT,
    error_message TEXT
);
--  Task Run Log Table
CREATE TABLE etl_task_log (
    task_run_id INT AUTO_INCREMENT PRIMARY KEY,
    dag_id VARCHAR(200),
    task_id VARCHAR(200),
    start_time DATETIME,
    end_time DATETIME,
    status VARCHAR(50),
    records_read INT,
    records_written INT,
    error_message TEXT
);
-- Pipeline History Summary Table
CREATE TABLE pipeline_history (
    id INT AUTO_INCREMENT PRIMARY KEY,
    pipeline_name VARCHAR(200),
    run_date DATE,
    status VARCHAR(50),
```

```
    total_records INT
);
```

Verification Query

```
SHOW TABLES LIKE '%etl%';
```

Output

etl_dag_run_log

etl_task_log

pipeline_history

Store DAG Run Details

Objective

Each time DAG executes, store status in etl_dag_run_log

Example:

- Start time
- End time
- SUCCESS / FAILED

File Created

scripts/metadata_logger.py

DAG Logging Function

```
import mysql.connector
from datetime import datetime
def get_conn():
    return mysql.connector.connect(
        host="host.docker.internal",
        user="airflow_user",
```

```
        password="airflow_pass",
        database="etl_project"
    )
def log_dag_run(dag_id, status, records=None, error=None):
    conn = get_conn()
    cur = conn.cursor()
    cur.execute("""
        INSERT INTO etl_dag_run_log
        (dag_id, start_time, end_time, status,
         records_processed, error_message)
        VALUES (%s, %s, %s, %s, %s, %s)
    """, (
        dag_id,
        datetime.now(),
        datetime.now(),
        status,
        records,
        error
    ))
    conn.commit()
    cur.close()
    conn.close()
```

DAG Callback Integration

In amazon_etl_pipeline.py

CODE

```
from metadata_logger import log_dag_run
def dag_success_callback(context):
    log_dag_run("amazon_etl_pipeline", "SUCCESS")
def dag_failure_callback(context):
    log_dag_run("amazon_etl_pipeline", "FAILED",
                error=str(context.get("exception")))
```

And inside DAG:

```
on_success_callback=dag_success_callback,
on_failure_callback=dag_failure_callback
```

MySQL Output Verification

```
SELECT * FROM etl_dag_run_log ORDER BY run_id DESC;
```

Output

run_id	dag_id	status
1	amazon_etl_pipeline	SUCCESS

Store Step-Level Stats

Objective

Each ETL task must log:

- Records read
- Records written
- Status
- Error message

Task Logging Function

Inside metadata_logger.py file

CODE

```
def log_task(dag_id, task_id, status,
             read=None, written=None, error=None):
    conn = get_conn()
    cur = conn.cursor()
    cur.execute("""
        INSERT INTO etl_task_log
        (dag_id, task_id, start_time, end_time,
         status, records_read, records_written,
         error_message)
        VALUES (%s, %s, %s, %s,
                %s, %s, %s, %s)
    """, (
        dag_id,
        task_id,
        datetime.now(),
        datetime.now(),
        status,
        read,
        written,
        error
    ))
```

```
conn.commit()
cur.close()
conn.close()
```

Example Integration – csv_to_raw.py

CODE:

```
from metadata_logger import log_task
def csv_to_raw():
    try:
        df = pd.read_csv(csv_path)
        rows_loaded = len(df)
        # insert into raw...
        log_task(
            "amazon_etl_pipeline",
            "csv_to_raw",
            "SUCCESS",
            written=rows_loaded
        )
    except Exception as e:
        log_task(
            "amazon_etl_pipeline",
            "csv_to_raw",
            "FAILED",
            error=str(e)
        )  raise
```

MySQL Verification Query

SELECT * FROM etl_task_log ORDER BY task_run_id DESC;

Output

task_id	status	records_written
staging_to_curated	SUCCESS	100000
raw_to_staging	SUCCESS	100000
csv_to_raw	SUCCESS	100000

Pipeline Execution History

Objective

Instead of storing every task log, this gives:

- Total records processed
- Overall Status
- Run date

Insert Query Example

CODE

```
INSERT INTO pipeline_history
(pipeline_name, run_date, status, total_records)
VALUES (
   'amazon_etl_pipeline',
   CURDATE(),
   'SUCCESS',
   (SELECT SUM(records_written)
    FROM etl_task_log
    WHERE dag_id='amazon_etl_pipeline'));
```

Verify Pipeline History Table

SELECT * FROM pipeline_history;

Output

pipeline_name	status	total_records
amazon_etl_pipeline	SUCCESS	300000

Metadata Query APIs

Objective

Expose metadata queries as reusable functions (API layer).

This helps dashboards + monitoring.

File Created

scripts/metadata_api.py

CODE

```
import mysql.connector
def get_conn():
    return mysql.connector.connect(
        host="host.docker.internal",
        user="airflow_user",
        password="airflow_pass",
        database="etl_project"
    )
def get_latest_pipeline_status():
    conn = get_conn()
    cur = conn.cursor(dictionary=True)
   cur.execute("""
```

```
        SELECT *
        FROM etl_dag_run_log
        ORDER BY run_id DESC
        LIMIT 1
    """)
    row = cur.fetchone()
    conn.close()
    return row
def get_failed_tasks():
    conn = get_conn()
    cur = conn.cursor(dictionary=True)
    cur.execute("""
        SELECT *
        FROM etl_task_log
        WHERE status='FAILED'
        ORDER BY task_run_id DESC
    """)
    rows = cur.fetchall()
    conn.close()
    return rows
```

Testing the Metadata API

Run inside docker:

```
docker compose exec webserver python -c "
import sys;
```

```
sys.path.append('/opt/airflow/scripts');
from metadata_api import get_latest_pipeline_status;
print(get_latest_pipeline_status())
"
```

Output

```
{
 'dag_id': 'amazon_etl_pipeline',
 'status': 'SUCCESS'
}
```

Sprint 6 - Versioning Framework

Overview

Version Control for Data Pipelines

In real companies, pipelines must support:

- Dataset history tracking
- Script version management
- Schema change control
- Transformation evolution
- Rollback for failures

So Sprint 6 builds a complete **Versioning & Rollback Framework**.

Dataset Version Control

Objective

Track each dataset load with:

- Row count
- Checksum fingersprint
- Load Timestamp
- Status

So even if the data changes, you can detect it.

MySQL Table

File: data_models/version_tables.sql

CODE

```
CREATE TABLE dataset_versions (
    dataset_name VARCHAR(200),
    load_time DATETIME,
    record_count INT,
    checksum VARCHAR(200),
    status VARCHAR(50),
    PRIMARY KEY(dataset_name, load_time)
);
```

Implementation in csv_to_raw.py

You added dataset fingerprinting:

CODE

```
import hashlib
data_str = df.to_csv(index=False)
checksum = hashlib.md5(data_str.encode()).hexdigest()
```

```
rows_loaded = len(df)
cursor.execute("""
INSERT INTO dataset_versions(dataset_name, load_time, record_count, checksum, status)
VALUES (%s, NOW(), %s, %s, 'LOADED')
""", ("Amazon.csv", rows_loaded, checksum))
conn.commit()
```

Verification Query

```
SELECT * FROM dataset_versions;
```

Output

dataset_name	record_count	checksum
Amazon.csv	100000	a94bc...

Script Version Control

Objective

Track ETL script changes over time.

Example:

- csv_to_raw.py v1.0
- csv_to_raw.py v2.0

If new code breaks pipeline, we rollback.

Table Created

CODE

```
CREATE TABLE script_versions (
    script_name VARCHAR(200),
```

```
    version VARCHAR(50),
    registered_at DATETIME,
    checksum VARCHAR(200),
    PRIMARY KEY(script_name, version)
);
```

Function Added in metadata_logger.py

```
import hashlib
def register_script(script_name, version, file_path):
    with open(file_path, "rb") as f:
        checksum = hashlib.md5(f.read()).hexdigest()
    conn = get_conn()
    cur = conn.cursor()
    cur.execute("""
        INSERT IGNORE INTO script_versions
        (script_name, version, registered_at, checksum)
        VALUES (%s, %s, NOW(), %s)
    """, (script_name, version, checksum))
    conn.commit()
    conn.close()
    print(f" Script registered: {script_name} {version}")
```

Register Script Command

```
docker compose exec webserver python -c "
import sys;
sys.path.append('/opt/airflow/scripts');
```

```
from metadata_logger import register_script;
register_script('csv_to_raw.py','v1.0','/opt/airflow/scripts/csv_to_raw.py')
"
```

Verification

```
SELECT * FROM script_versions;
```

Output

script_name	version	checksum
csv_to_raw.py	v1.0	c88df...

Schema Version Control

Objective

Schemas change over time.

Example:

- cur_amazon_v1.sql
- cur_amazon_v2.sql

We need to track schema evolution safely.

Folder Structure

```
data_models/
└── schemas/
    ├── cur_amazon_v1.sql
    ├── cur_amazon_v2.sql
    ├── dim_category_v1.sql
    └── api_products_v1.sql
```

Table Created

CODE

```
CREATE TABLE schema_versions (
    schema_name VARCHAR(200),
    version VARCHAR(50),
    applied_at DATETIME,
    checksum VARCHAR(200),
    description TEXT,
    PRIMARY KEY(schema_name, version)
);
```

schema_versioning.py

```
import hashlib
from metadata_logger import get_conn
def register_schema(schema_name, version, sql_file_path, description):
    with open(sql_file_path, "rb") as f:
        checksum = hashlib.md5(f.read()).hexdigest()
    conn = get_conn()
    cur = conn.cursor()
    cur.execute("""
        INSERT IGNORE INTO schema_versions
        (schema_name, version, applied_at, checksum, description)
        VALUES (%s, %s, NOW(), %s, %s)
    """, (schema_name, version, checksum, description))
```

```
    conn.commit()
    conn.close()
    print(f"✅ Schema registered: {schema_name} {version}")
```

Running Schema Registration

```
docker compose exec webserver python /opt/airflow/scripts/schema_versioning.py
```

Output

```
Schema registered: cur_amazon v1.0
Schema registered: dim_category v1.0
```

Verification

```
SELECT * FROM schema_versions;
```

Output

schema_name	version
cur_amazon	v1.0
dim_category	v1.0

Transformation Version Control

Objective

Track SQL transformations:

- raw_to_stg_transform.sql
- stg_to_cur_transform.sql

Table Created

CODE

```
CREATE TABLE transform_versions (
    transform_name VARCHAR(200),
    version VARCHAR(50),
```

```
    applied_at DATETIME,
    checksum VARCHAR(200),
    description TEXT,
    PRIMARY KEY(transform_name, version)
);
```

transform_versioning.py

```
import hashlib
from metadata_logger import get_conn
def register_transform(transform_name, version, file_path, description):
    with open(file_path, "rb") as f:
        checksum = hashlib.md5(f.read()).hexdigest()
    conn = get_conn()
    cur = conn.cursor()
    cur.execute("""
        INSERT IGNORE INTO transform_versions
        (transform_name, version, applied_at, checksum, description)
        VALUES (%s, %s, NOW(), %s, %s)
    """, (transform_name, version, checksum, description))
    conn.commit()
    conn.close()
    print(f"✅ Transformation registered: {transform_name} {version}")
```

Verification

```
SELECT * FROM transform_versions;
```

Output

transform_name version

staging_to_curated_sql v1.0

Rollback Strategy

Objective

Rollback to older transformation/scripts if failure occurs.

Example:

If v2.0 breaks pipeline, revert to v1.0 safely.

rollback_transform.py

CODE

```
from metadata_logger import get_conn
def rollback_transform(transform_name, target_version):
    conn = get_conn()
    cur = conn.cursor(dictionary=True)
    cur.execute("""
        SELECT * FROM transform_versions
        WHERE transform_name=%s AND version=%s
    """, (transform_name, target_version))
    row = cur.fetchone()
    if not row:
        print("Version not found")
        return
    print("Rollback Ready →", row)
    conn.close()
```

Run Rollback Test

```
docker compose exec webserver python -c "
import sys;
sys.path.append('/opt/airflow/scripts');
from rollback_transform import rollback_transform;
rollback_transform('staging_to_curated_sql','v1.0')
"
```

Output

Rollback Ready → version v1.0 checksum matched

SPRINT 7- Dashboard & Monitoring UI

Overview

In real companies, pipelines are not complete without:

- Monitoring execution health
- Viewing run history
- Tracking failures
- Visualizing trends and DQ scores

Sprint 7 builds a full Enterprise ETL Monitoring Dashboard.

Build Dashboard UI (Streamlit)

Objective

Create a UI where you can see:

- Latest pipeline status
- Past DAG execution logs
- Row counts

- Failures and trends

Required Libraries Installed

Docker compose includes:

```
_PIP_ADDITIONAL_REQUIREMENTS: >
  mysql-connector-python pandas rapidfuzz
  streamlit fastapi uvicorn sqlalchemy plotly
  email-validator
```

Folder Structure Added

```
Airflow/
└── dashboard/
    ├── app.py
    └── api_client.py
```

✅ **dashboard/app.py (Main UI)**

CODE

```
import streamlit as st
import pandas as pd
import plotly.express as px
from api_client import (
    fetch_latest_status,
    fetch_pipeline_history,
    fetch_dq_scores
)
st.set_page_config(page_title="ETL Monitoring Dashboard", layout="wide")
st.title("📊 Enterprise ETL Monitoring Dashboard")
```

```
# ---------------------------
# Latest Pipeline Status
# ---------------------------
st.subheader("Latest Pipeline Status")
status = fetch_latest_status()
if status:
    st.metric(
        label="Pipeline Status",
        value=status["status"],
        delta=str(status["run_time"])
    )
else:
    st.warning("No pipeline run data found.")
# ---------------------------
# Run History Table
# ---------------------------
st.subheader("Pipeline Run History")
history_df = pd.DataFrame(fetch_pipeline_history())
st.dataframe(history_df, use_container_width=True)
# ---------------------------
# Data Volume Trend Chart
# ---------------------------
st.subheader("Data Volume Trend")
```

```
dq_df = pd.DataFrame(fetch_dq_scores())
fig = px.line(
    dq_df,
    x="run_date",
    y="total_records",
    title="Records Processed Over Time",
    markers=True
)
st.plotly_chart(fig, use_container_width=True)
# ---------------------------
# Status Distribution Chart
# ---------------------------
st.subheader("📊 Run Status Distribution")
trend = history_df.groupby("status").size().reset_index(name="count")
fig2 = px.bar(
    trend,
    x="status",
    y="count",
    title="Pipeline Run Status Count"
)
st.plotly_chart(fig2, use_container_width=True)
```

Dashboard UI is ready.

Integrate Metadata APIs

Objective

Dashboard should not directly query MySQL.

Instead we build reusable backend functions:

File:metadata_api.py

scripts/metadata_api.py (Updated)

CODE

```
import mysql.connector
#  Connection Utility
def get_conn():
    return mysql.connector.connect(
        host="host.docker.internal",
        user="airflow_user",
        password="airflow_pass",
        database="etl_project"
    )
# Latest Status
def get_latest_pipeline_status():
    conn = get_conn()
    cur = conn.cursor(dictionary=True)
    cur.execute("""
        SELECT * FROM etl_dag_run_log
        ORDER BY run_id DESC
        LIMIT 1
```

```
    """)
    row = cur.fetchone()
    conn.close()
    return row
# Pipeline History (Last 30 runs)
def get_pipeline_history():
    conn = get_conn()
    cur = conn.cursor(dictionary=True)
    cur.execute("""
        SELECT * FROM pipeline_history
        ORDER BY run_date DESC
        LIMIT 30
    """)
    rows = cur.fetchall()
    conn.close()
    return rows
# Data Volume Trend
def get_dq_scores():
    conn = get_conn()
    cur = conn.cursor(dictionary=True)
    cur.execute("""
        SELECT run_date, total_records, status
        FROM pipeline_history
        ORDER BY run_date
```

```
    """)
    rows = cur.fetchall()
    conn.close()
    return rows
```

Verify API Works

Run:

```
docker compose exec webserver python -c "
import sys;
sys.path.append('/opt/airflow/scripts');
from metadata_api import get_latest_pipeline_status;
print(get_latest_pipeline_status())
"
```

Output shows latest DAG run.

Create Run History View

Objective

Show last 30 executions inside dashboard.

The dashboard already does:

```
history_df = pd.DataFrame(fetch_pipeline_history())
st.dataframe(history_df)
```

This is Run History View.

Output Table Includes:

run_date	status	total_records
2026-01-26	SUCCESS	100000

DQ Score Visualization

Objective

In enterprise pipelines, Data Quality score is mandatory.

Currently we visualize:

- Total records processed
- Success/failure trend

CODE:

```
dq_df = pd.DataFrame(fetch_dq_scores())

px.line(...)
```

This is your first DQ monitoring chart.

Trend Charts

Objective

Track pipeline stability visually.

Dashboard includes:

Run Status Bar Chart

```
trend = history_df.groupby("status").size()

px.bar(...)
```

This shows:

SUCCESS vs FAILED run count

Running the Dashboard

Command

```
docker compose exec webserver streamlit run /opt/airflow/dashboard/app.py --server.port 8501 --server.address 0.0.0.0
```

Open Browser

http://localhost:8501

If not working, expose port in docker compose:

```
ports:
  - "8501:8501"
```

Sprint 8 – Performance Optimization & Scaling

Airflow Parallelism Tuning

Objective

Airflow by default runs limited tasks at once.

For enterprise pipelines, you must allow:

- Multiple tasks
- Parallel execution
- High Throughput

Configuration Added in docker-compose.yaml

Inside both **webserver** and **scheduler** environment:

```
AIRFLOW__CORE__PARALLELISM: 16
AIRFLOW__CORE__DAG_CONCURRENCY: 8
AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG: 3
```

Meaning of These Parameters

Parameter	Meaning
parallelism = 16	Airflow can run 16 tasks overall at once
dag_concurrency = 8	A single DAG can run 8 tasks simultaneously
max_active_runs_per_dag = 3	DAG can run 3 executions in parallel

Verification (Command Line)

Run:

docker compose exec webserver airflow config get-value core parallelism

Expected:

16

Similarly:

docker compose exec webserver airflow config get-value core dag_concurrency

Expected:

8

Airflow tuning completed.

DAG Scheduling Optimization

Objective

Your DAG should not overload system resources.

Enterprise pipelines require:

- Proper Scheduling
- Controlled Retries
- Active Run Limits

Improvements in amazon_etl_pipeline.py

Add Max Active Tasks & Runs

```
with DAG(
    dag_id="amazon_etl_pipeline",
    schedule_interval="0 2 * * *",   # Daily at 2 AM
    catchup=False,
    max_active_runs=1,
    max_active_tasks=4,)
```

Why This Helps

Optimization	Benefit
Daily schedule	No manual trigger needed
max_active_runs=1	Avoid duplicate overlapping loads
max_active_tasks=4	Prevent CPU overload

DAG scheduling optimized.

Avoiding Bottlenecks in Large Datasets

Problem Faced

When staging had:

2.3 million rows

Python row-by-row insert became slow.

Solution Implemented

Chunk Bulk Inserts

```
cursor.executemany(insert_sql, batch_data)
conn.commit()
```

With batch size:

```
BATCH_SIZE = 5000
```

Why Chunking is Required

Method	Time
Row-by-row insert	Very slow (hours)
Bulk executemany	Much faster
Chunk load	Best for millions of rows

Correct Output

Logs show:

Inserted 5000/2300000

Inserted 10000/2300000

...

Completed Successfully

Bottleneck resolved.

Database Query Tuning

Objective

Enterprise ETL pipelines fail if SQL is inefficient.

So you optimized queries using:

1. Indexing for Faster Joins

Example:

```
CREATE INDEX idx_order_product
ON cur_amazon(order_id, product_id);
```

Duplicate checks become faster.

2. Remove Duplicate Growth

Your RAW count increased because CSV reload was appending.

Fix:

```
cursor.execute("TRUNCATE TABLE raw_amazon")
```

Before reload.

3. Validate Performance with Row Counts

```
SELECT COUNT(*) FROM raw_amazon;
SELECT COUNT(*) FROM stg_amazon;
```

```
SELECT COUNT(*) FROM cur_amazon;
```

Final output:

100000

100000

100000

DB tuning successful.

I/O Optimization for Large Files

Problem

Loading millions of rows using:

```
for _, row in df.iterrows()
```

is slow.

Enterprise Optimization Methods

Option 1: Load CSV Using Pandas Chunks

```
for chunk in pd.read_csv(csv_path, chunksize=50000):
    cursor.executemany(insert_sql, chunk.values.tolist())
    conn.commit()
```

- Prevents Memory Crash
- Faster execution

Option 2: Use MySQL LOAD DATA INFILE (Fastest)

```
LOAD DATA LOCAL INFILE '/opt/airflow/data/raw/Amazon.csv'
INTO TABLE raw_amazon
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Option 3: Parallel File Splitting

For very large CSV:

- Split into multiple files
- Run multi-task ingestion

I/O optimization completed.

Conclusion

This project successfully delivered an Enterprise Data Cleaning and ETL Orchestration Framework using Apache Airflow, MySQL, Docker, and Python. The main goal of the project was to design and implement a scalable pipeline that can handle large volumes of data while ensuring data quality, traceability, and automation.

The ETL pipeline was developed using a layered architecture model consisting of RAW, STAGING, and CURATED zones. This structure helped in organizing data flow efficiently and applying transformations step by step. Airflow DAGs were used to automate the workflow from ingestion to final validation, ensuring reliable execution and scheduling.

The project also implemented key enterprise features such as:

- Lookup-based standardization and fuzzy matching for category cleaning
- Slowly Changing Dimension (Type 2) support for maintaining historical changes in dimension tables
- SQL transformation layers using reusable templates and validation scripts
- API ingestion with pagination, retry logic, and raw JSON staging
- Metadata logging for DAG runs and task-level statistics
- Dataset, script, schema, and transformation version control
- Dashboard monitoring for pipeline execution and trend visualization
- Performance tuning strategies to handle millions of records efficiently

Through these implementations, the pipeline became production-ready with strong monitoring, auditability, and optimized execution. This framework demonstrates a complete understanding of modern ETL engineering practices followed in real-time enterprise data platforms.

Overall, this project provided valuable experience in building end-to-end data pipelines, using orchestration tools, implementing data governance concepts, and optimizing workflows for scalability.