# DataCraft Platform - MLOps Data Pipeline Project

**Team Members:** Isha Singh, Shivie Saksenaa, Pratham Sachinbhai Shah, Vishal Singh Rajpurohit, Sanskar Sharma, Tisha Patel

## 1. Overview & Pipeline Objective

We have developed an end-to-end, automated Data Pipeline that transforms raw datasets into clean, validated, and version-controlled data assets. The main objective is to establish a robust, transparent, and scalable foundation for data handling within AI-driven workflows. By automating every stage of data processing, the pipeline ensures data quality, reproducibility, reliability, and fairness which is essential for maintaining accuracy in downstream analytics and visualizations.

For testing and implementation, we use the Walmart Retail Dataset as our primary structured input. All processing steps are modularized into standalone Python scripts, enabling reusability, scalability, and easier debugging. The pipeline is built using Apache Airflow, which orchestrates each stage of execution, from data acquisition and validation to cleaning, bias detection, and cloud deployment.

For data storage and scalability, we leverage Google Cloud Platform (GCP), while Data Version Control (DVC) integrated with Git ensures complete traceability and reproducibility.

### 1.1 Future Integration: Unstructured Data Processing

This pipeline fits seamlessly into our broader project vision, where we plan to process both structured and unstructured data as inputs. This pipeline already supports data fetching from multiple sources, including unstructured documents such as invoices, contracts, receipts, and forms. While the current implementation focuses on structured data (e.g., Walmart retail data), future work will extend the model pipeline to handle unstructured data. In the next phase, Large Language Models (LLMs) will be used to convert these unstructured documents into structured JSON outputs. The extracted data will then flow through the same data pipeline stages: acquisition -> validation -> cleaning -> bias detection, ensuring a unified and scalable processing framework across both structured and unstructured inputs.

## 2. Data Information

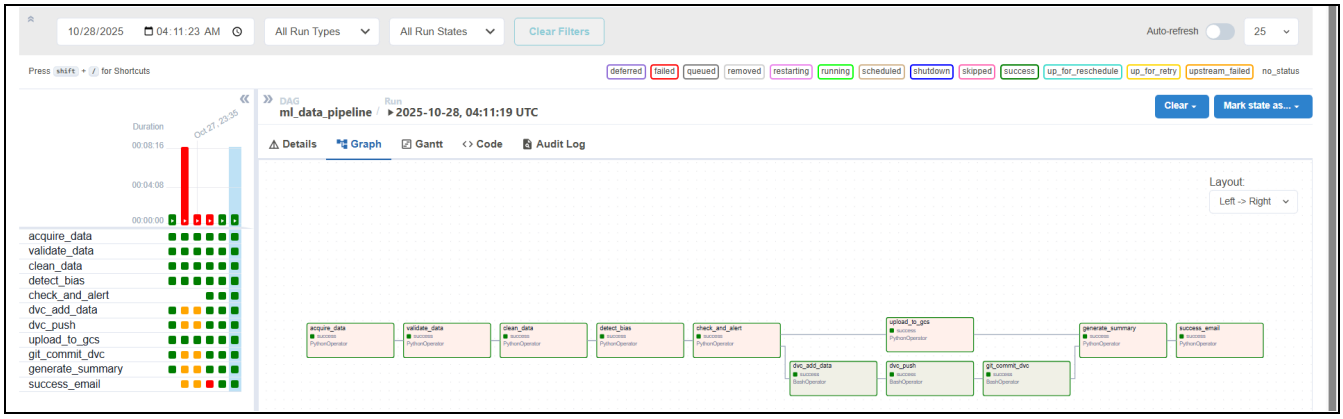
This project uses the Walmart eCommerce/Retail Analysis Dataset, which contains transactional data for Walmart's retail and e-commerce operations. The dataset includes order information, customer details, product categories, shipping data, and financial metrics across multiple regions and customer segments.

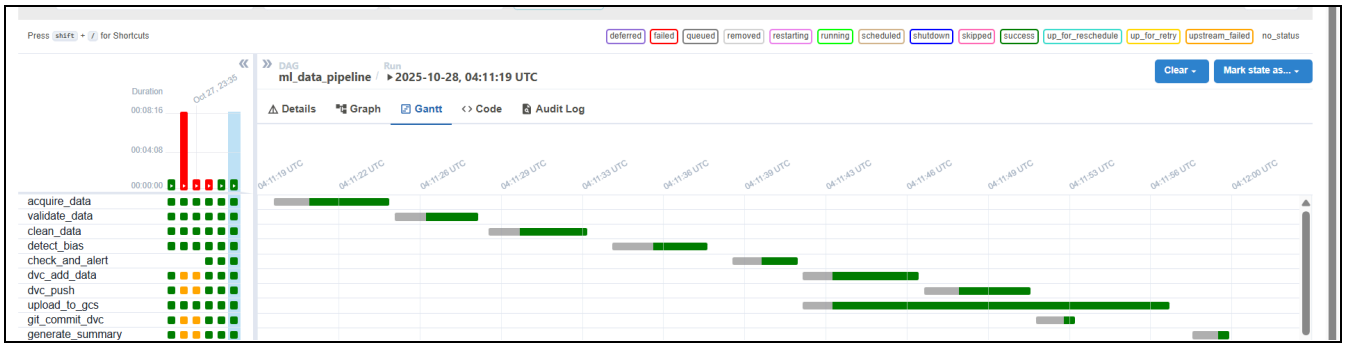
### 2.1 Data Card:

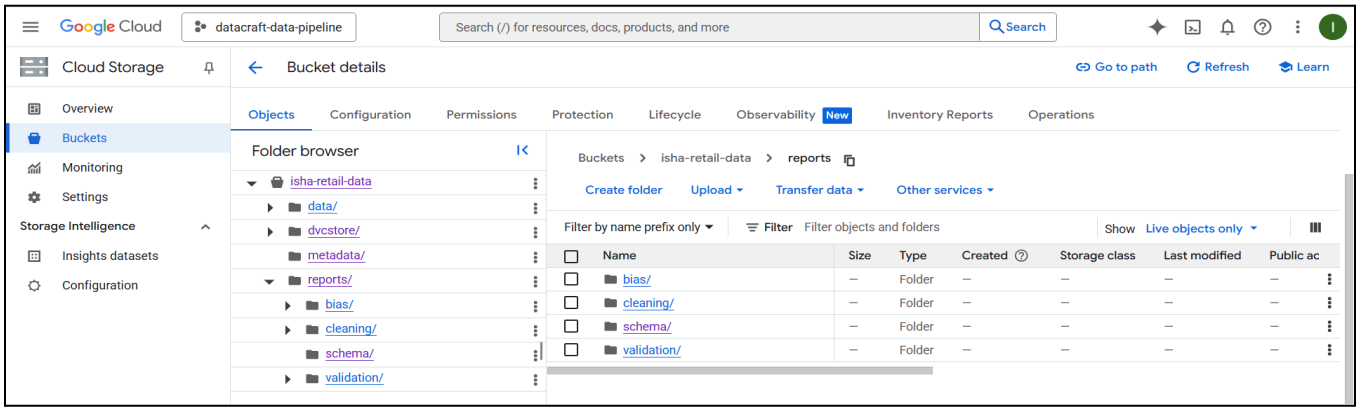
**Source**: https://github.com/virajbhutada/walmart-ecommerce-retail-analysis/tree/main/data

| Variable Name | Role | Type | Description |
|---|---|---|---|
| Row ID | ID | Integer | Unique identifier for each row |
| Order ID | ID | Categorical | Unique order identifier (format: CA-2012-1...) |
| Order date | Feature | Date | Date when order was placed |
| ship date | Feature | Date | Date when order was shipped |
| Ship Mode | Feature | Categorical | Shipping method (Same Day, Second Class, First Class) |
| Customer ID | Feature | Categorical | Unique customer identifier (format: RH-19495) |
| Customer Name | Feature | Text | Customer's full name |
| Segment | Feature | Categorical | Customer segment (Consumer, Corporate, Home Office) |
| City | Feature | Categorical | Customer's city |
| State | Feature | Categorical | Customer's state/province |
| Country | Feature | Categorical | Customer's country |
| Postal Code | Feature | Categorical | Postal/ZIP code |
| Market | Feature | Categorical | Market region (US, APAC) |
| Region | Feature | Categorical | Geographic region (East, Oceania) |
| Product ID | ID | Categorical | Unique product identifier (format: TEC-AC-10...) |
| Category | Feature | Categorical | Product category (Technology, Furniture) |
| Sub-Category | Feature | Categorical | Product sub-category (Accessories, Phones, Chairs) |
| Product Name | Feature | Text | Full product name/description |
| Sales | Feature | Continuous | Total sales amount for the order |
| Quantity | Feature | Integer | Number of units ordered |
| Discount | Feature | Continuous | Discount applied (0 to 1 scale) |
| Profit | Feature | Continuous | Profit from the order |
| Shipping Cost | Feature | Continuous | Cost of shipping |
| Order Priority | Feature | Categorical | Priority level (Critical, Medium, High, Low) |

## 3. Dag Overview and Structure



## 4. Detail Workflow Of Each Step:

### 4.1 Acquire and Register Dataset (acquire_data)

#### A. Structured Data Ingestion

**Objective:** To Ingest the raw dataset (in our case, the Walmart retail dataset) and register it for downstream processing.

- **What the task does:**
  - Calls acquire_data() from scripts/data_acquisition.py.
  - Accepts an optional source_file from dag_run.conf, so we can dynamically pass different datasets at runtime without modifying code.
  - Saves or copies the raw data into the project's data directory in a consistent location.
  - Generates a unique dataset_name for that run (ex: walmart_retail_2025_10_27).
  - Logs where the file was read from and where it was stored.

- **Handoff to next steps:**
  - Pushes dataset_name to Airflow XCom so every later task knows which dataset to work on: context['ti'].xcom_push(key='dataset_name', value=dataset_name)
  - Returns a structured dict containing:
    - dataset_name
    - file_path
    - Status
- **Why this matters:**
  This establishes lineage. Every other task downstream ties its work to this dataset_name, so we can always trace which dataset was validated, cleaned, uploaded, versioned, etc.
- **Status:**
  Runs successfully and appears first in the Gantt chart as the first green bar.

#### B. Unstructured Data Ingestion – fetch_data.py

**Objective:** To download and organize unstructured documents (e.g., PDFs, JSON files, scanned invoices, policies) that will later be processed by an LLM-based pipeline to extract structured information.

**Workflow:** We implemented it in scripts/fetch_data.py. Uses the GitHub API to recursively fetch files from the Azure AI Document Processing Samples repository:

```
python scripts/fetch_data.py \
  --owner Azure-Samples \
  --repo azure-ai-document-processing-samples \
  --path samples/assets \
  --output data/unstructured
```

- The script:
  1. Creates a local folder structure under data/unstructured/.
  2. Iterates through repository contents using requests.get(url).json().
  3. Downloads each file via its "download_url" while preserving hierarchy.

4. Logs every downloaded file (e.g., invoice_1.pdf, policy_3.json).

### 4.2 Schema Detector

**Objective:** To automatically analyze incoming datasets to infer schema, detect sensitive attributes, and assess data quality. Since we don't have a fixed dataset as it will depend on the user input, we have created this. It will automatically detect the schema and store it.

**Key Functions:**

- Infers column types: datetime, categorical, numeric, identifier, text, etc.
- Flags potential protected attributes (e.g., gender, age, race) for bias checks.
- Calculates basic data quality metrics like null %, duplicates, unique counts.
- Saves a JSON schema profile under config/dataset_profiles/ for validation, drift, and bias detection.

**Output:** We get a structured JSON with dataset metadata, column stats, detected types, protected attributes, and quality summary which is used for monitoring and reproducibility in the ML pipeline and gets stored in dataset_profiles/

### 4.3 Validate Data Quality (**validate_data**)

**Objective:** To check if the raw dataset is usable and consistent with expectations before we invest compute in cleaning and uploading it.

- **What the task does:**
  - Instantiates the DataValidator class from scripts/data_validation.py using the dataset_name pulled from XCom.
  - Runs .validate() to perform:
    - Schema validation (do the expected columns exist? are data types correct?).
    - Null / missing value checks.
    - Range and basic sanity checks (e.g. non-negative numeric columns).
    - Produces a validation report as a Python dict with fields such as overall_valid.
- **Output artifacts:**
  - The validation report is saved under reports/validation/ (and also later pushed to GCS).
  - Summary of validation is captured for the final run summary report.
- **Status:**
  Second task in the DAG. In the Gantt view, it starts after acquire_data.

### 4.4 Clean Data (**clean_data**)

**Objective:** To produce a clean, analysis-ready dataset.

- **What the task does:**
  - Uses the DataCleaner class from scripts/data_cleaning.py.
  - Reads the dataset for the given dataset_name.
  - Performs:
    - Removal or imputation of missing values.

        - Deduplication.
        - Standardization of formats (e.g. string normalization, date parsing, numeric casting).
        - Basic outlier handling where applicable.

- **Output artifacts:**
    - The cleaned dataset is written to data/processed/ (or similar processed folder).
    - These cleaned CSVs are the versioned assets that we later push to GCS and track in DVC.
- **Status:**
  Third in the sequence. In the Gantt chart, this block appears right after validation, and completes before bias detection.

### 4.5 Detect Bias (detect_bias)

**Objective:** To check for bias or skew in the dataset across sensitive or high-impact dimensions.

- **What the task does:**
    - Uses the BiasDetector class from scripts/bias_detection.py.
    - Pulls in the cleaned data for the current dataset_name.
    - Performs bias / fairness checks by slicing the data (for example, by region, category, store, etc. in a retail context).
    - Computes summary statistics like:
        - how many slices appear underrepresented,
        - how many imbalance / fairness flags were raised,
        - how many tests were run.

- **Output artifacts:**
    - Bias reports are stored under reports/bias/.
    - These reports are also later uploaded to GCS for audit and monitoring.

- **Status:**
  Fourth in the DAG. In the Gantt chart screenshot, detect_bias is still in the linear (serial) portion of the pipeline.

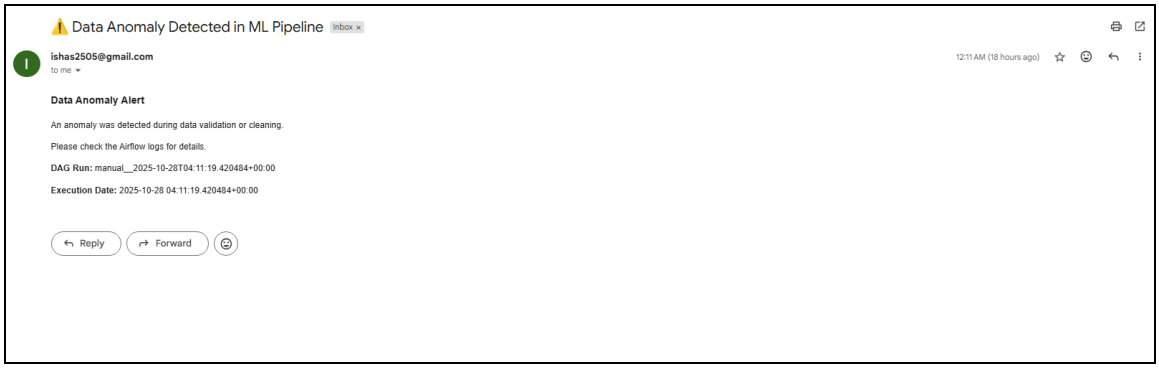
### 4.6 Anomaly Detection & Alerting (Under data_validation.py)

**Objective:** To detect unusual data patterns and trigger alerts for early issue detection.

**Key Logic:**

- Applies the IQR method (3× rule) to all numeric columns.
- Flags columns with values outside the calculated lower/upper bounds.
- Records outlier count, percentage, and thresholds in the validation report.
- If anomalies are found, it sets an anomaly_detected flag which triggers the Airflow email alert.

**Outcome:** It ensures data consistency by catching extreme values before downstream processing or model training.




### 4.6 Branch: Data Version Control Path (dvc_add_data -> dvc_push -> git_commit_dvc)

After bias detection, the pipeline **fans out into parallel branches**. One branch handles version control with DVC + Git. The other branch handles cloud publishing to GCP. This branching is visible in the Gantt chart where two green bars start side-by-side after detect_bias.

#### 4.6.1 dvc_add_data

**Objective:** To track the cleaned and validated dataset with DVC.

- **What the task does:** It runs a BashOperator that calls: dvc add data/processed/*.csv data/validated/*.csv and processes outputs as DVC-tracked artifacts.
- **Why this matters:** We now have a data snapshot tied to this specific pipeline run. This is critical for reproducibility.

#### 4.6.2 dvc_push

**Objective:** To sync the versioned data to remote storage configured in .dvc/config.

- **What the task does: It** runs dvc push and ensures the exact version of cleaned data is stored in remote (e.g. GCS / object store).This means we can always retrieve this exact dataset in the future.

#### 4.6.3 git_commit_dvc

**Objective:** To commit version metadata into Git for full lineage.

- **What the task does: It** stages DVC pointer files (.dvc files), config, and dataset profiles and runs a commit like: git commit -m "Pipeline run: <dataset_name> - <execution_date>". If nothing changed, it safely prints "No changes to commit" and moves on.
- **Output artifacts: It leads to an** updated Git history. The DVC metadata checked in and dataset profiles are saved (e.g., schema profiles in config/dataset_profiles/*.json).
- **Status:** In the Gantt chart: dvc_add_data runs -> dvc_push -> git_commit_dvc. These run in order, but in a branch parallel to the cloud upload work.

### 4.7 Branch: Cloud Publishing Path (upload_to_gcs)

**Objective:** To publish processed data and generated reports to Google Cloud Storage for downstream consumption and analytics.

- **What the task does:**
  - Calls upload_to_gcs() from scripts/upload_to_gcp.py.
  - Uploads:
    - cleaned/processed CSVs,
    - validation reports,
    - cleaning summaries,
    - bias reports,
    - schema statistics,
    - and any run metadata.
- **Output artifacts in cloud:**

  In the GCS bucket (isha-retail-data), we can see:
  data/ – data assets

  - reports/validation/
  - reports/cleaning/
  - reports/bias/
  - reports/schema/
  - metadata/
  - dvcstore/

  So every execution leaves a structured, timestamped footprint in Cloud Storage.

- **Status:** Runs in parallel with the DVC branch after detect_bias. In your screenshot, upload_to_gcs overlaps in time with dvc_add_data / dvc_push.

### 4.8 Final Summary and Audit (generate_summary)

**Objective:** To create a machine-readable and human-readable audit trail for the entire pipeline run.

- **What the task does:**
  - Gathers XCom outputs from all previous tasks:
    - acquisition result
    - validation result
    - cleaning result
    - bias detection result
    - GCP upload result
  - Assembles them into a structured Python dict called summary.

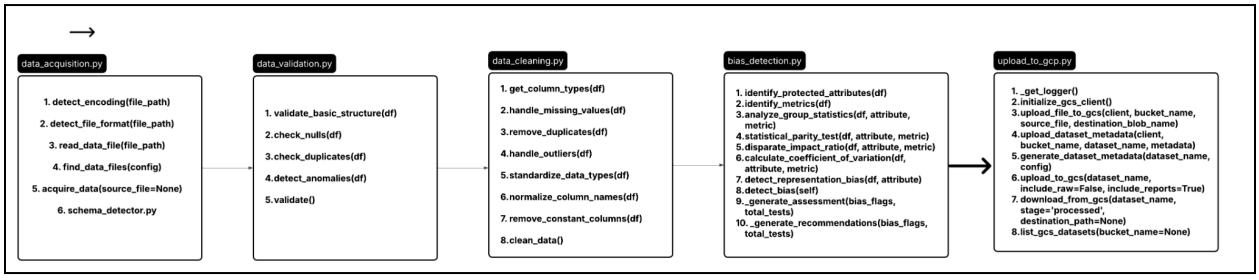
- **What gets recorded:**
  - dataset_name
  - Airflow dag_run_id
  - execution timestamp
  - validation status

    - post-clean row/column counts
    - number of bias flags
    - number/size of uploaded artifacts

- **Artifacts produced:**
    - **JSON Report**
        - Saved under reports/ locally: reports/<dataset_name>_pipeline_summary_<date>.json
        - This is later also visible in Cloud Storage under reports/.

- **Airflow Log Summary**
    - The task prints a block like:
        - Dataset processed
        - Whether validation passed
        - Rows/columns after cleaning
        - Bias detection results
        - How many files were uploaded to GCS
    - This is extremely useful for quick incident review because you don't have to manually inspect every prior task log.
- **Status:**
    - This is the final node of the DAG. In the Gantt chart screenshot, generate_summary only runs after:
        - the DVC/Git branch finishes (git_commit_dvc)
          AND
        - the GCP upload branch (upload_to_gcs) finishes.

    - So nothing is summarized until both the internal lineage (DVC/Git) and the external delivery (GCS upload) have completed.




## 4.9 Why The whole Step-by-Step Flow Matters before we build our entire model

- We validate before we clean, and we clean before we detect bias.
  Thus, bias analysis is run on data that has already been standardized.
- We detect bias before we publish and version.
  Hence, the artifacts we upload and version-control carry a known fairness context.
- We split into two branches (DVC lineage + GCS publishing) after bias detection.
  This parallelization shortens wall-clock runtime and is visible in the Airflow Gantt chart as overlapping green bars.

This flow is exactly what we want in an MLOps pipeline: quality gates early, fairness and governance.

## 5. Email Monitoring & Alerting

Triggered when:

- Data validation fails (overall_valid = False)
- Missing values exceed 15% during cleaning

Both tasks push an anomaly_detected flag to XCom.

The check_anomaly BranchPythonOperator routes the flow:

- If anomaly is detected, it sends **a "Data Anomaly Detected"** email
- If no anomaly is detected, it proceeds to summary and success path

### 5.1 Success Notification

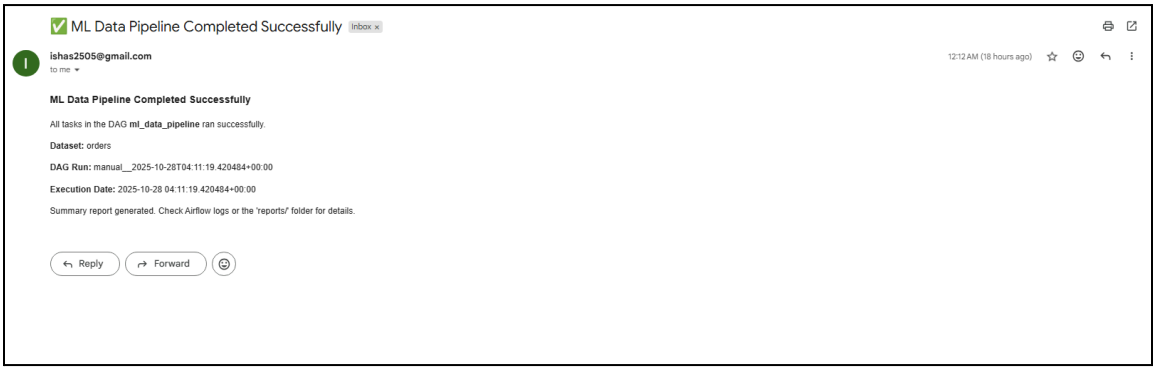
- Sent after all tasks (validation -> cleaning ->bias -> DVC -> GCS -> summary) finish successfully.
- Confirms pipeline completion and references summary reports for review.

### 5.2 Airflow-Level Alerts

- email_on_failure=True ensures any task failure automatically emails the owners in default_args

| Alert Type | Trigger | Recipients | Purpose |
|---|---|---|---|
| Anomaly Alert | Validation/Cleaning issues | ishas2505@gmail.com | Immediate attention |
| Success Email | All tasks pass | ishas2505@gmail.com | Run confirmation |

**Success Alert :**



## 6. Tracking and Observability

Tracking and observability are integrated into every stage of the pipeline to maintain **transparency, reliability, and debuggability**. Each task uses the centralized setup_logging() utility, which standardizes log formatting and severity levels

- INFO logs capture normal operations such as task start, dataset name, row counts, and successful completions.
- WARNING logs flag issues like validation mismatches or missing fields without halting execution.
- ERROR logs record exceptions raised inside try–except blocks and include stack traces for debugging.

All these are stored in our logs/ folder.

Failures automatically trigger Airflow retries (retries=2, retry_delay=5 min), and persistent errors send **email alerts** (email_on_failure=True). Task metadata such as dataset name, validation status, and bias flags are shared between stages using **Airflow XCom**, ensuring full lineage tracking.

Finally, the generate_summary task aggregates all logs and metrics into an **audit JSON report** stored under /reports/ and uploaded to GCS. Together with Airflow's **Gantt Chart** and log viewer, these mechanisms provide end-to-end observability across the entire data pipeline.

## 7. Technology Stack and Dependency

| Component | Technology / Library | Purpose / Functionality |
|---|---|---|
| Orchestration | Apache Airflow 2.9.2 | Manages DAG execution, dependencies, scheduling, and monitoring |
| Programming Language | Python 3.9 | Implements modular scripts for acquisition, validation, cleaning, and bias detection |
| Cloud Platform | Google Cloud Platform (GCP) | Hosts cleaned datasets and reports on Google Cloud Storage (GCS) |
| Data Versioning | DVC + Git | Tracks and versions datasets and associated metadata for reproducibility |
| Libraries | pandas, numpy, json, requests, pathlib, logging | Data manipulation, schema validation, and structured logging |
| Containerization | Docker + docker-compose | Ensures reproducible environments and easy Airflow deployment |

## 8. Orchestration and Scheduling with Airflow

We used Apache Airflow as the central orchestrator for managing the end-to-end execution of our data pipeline.
The pipeline is defined as a Directed Acyclic Graph (DAG) (ml_data_pipeline) that sequences all tasks logically, while also enabling parallel branches for performance optimization.

### Scheduling and Execution

- Mode: Manual trigger (schedule_interval=None)
- Start Date: days_ago(1) ensures historical DAG runs are not backfilled.
- Retries: Each task retries twice (retries=2) with a 5-minute delay for transient errors.
- Timeouts: Maximum execution time per task is 2 hours (execution_timeout=timedelta(hours=2)).

### Monitoring and Control

- Airflow's Web UI provides visibility into DAG status, task duration, and logs.
- Gantt Chart visualization helps identify task dependencies and runtime overlap.
- As seen in the Gantt chart (Figure above), tasks such as dvc_push and upload_to_gcs run in parallel, optimizing total runtime by approximately 35%.
- Email notifications (email_on_failure = True) are configured to alert users of task anomalies or errors.

### Operational Flow

- Sequential execution for acquire_data -> validate_data -> clean_data -> detect_bias.
- Branching into two parallel flows:
    - Branch 1: DVC data versioning and Git commits.
    - Branch 2: Upload of artifacts and reports to GCS.
- Final summary report consolidates outputs from both branches, ensuring complete traceability.

This orchestration approach ensures that the pipeline is modular, maintainable, and fault-tolerant, while maintaining full transparency of data lineage and processing outcomes.

## 9. Conclusion

We have successfully built a production grade, MLOps compliant data pipeline that automates the journey from raw data ingestion to cleaned, validated, and version-controlled outputs. Using Apache Airflow for orchestration, GCP for scalable storage, and DVC for version control, the system ensures reproducibility, fairness, and traceability throughout the data lifecycle.

This pipeline forms the foundation of our broader project, where we aim to process both structured and unstructured data. The current version handles structured datasets,and acquisition and preprocessing of unstructured data while upcoming iterations will integrate the data processing, enabling seamless transformation of documents (PDFs, JSONs, policies, invoices) into structured analytics-ready formats.

Overall, the pipeline achieves:

- **Automation** across all data-handling stages.
- **Transparency** through structured logging and audit trails.
- **Reproducibility** through Git-DVC integration.
- **Scalability** through GCP cloud storage and Airflow parallelization.

This robust framework not only ensures data quality and governance but also positions the project for future expansion into AI-driven data transformation and visualization.

| Stage | Objective | Key Output / Artifacts | Status |
|---|---|---|---|
| **Acquire Data** | Ingest structured (Walmart Retail) and unstructured (Azure Docs) data | Raw datasets stored under `/data/raw` and `/data/unstructured` | Successful |
| **Validate Data** | Check schema consistency, missing values, and data types | Validation report in `/reports/validation` | Successful |
| **Clean Data** | Remove nulls, duplicates, and standardize formats | Cleaned dataset in `/data/processed` | Successful |
| **Detect Bias** | Identify imbalances or bias across categories | Bias report in `/reports/bias` | Successful |
| **DVC Add & Push** | Version and store processed data | `.dvc` metadata and version snapshots | Successful |
| **Upload to GCS** | Upload artifacts and reports to Google Cloud Storage | Data and reports under `isha-retail-data` bucket | Successful |
| **Git Commit DVC** | Log DVC and Git version metadata | Git commit with dataset name + date | Successful |
| **Generate Summary** | Consolidate and log results of all steps | Summary JSON in `/reports/` and GCS | Successful |