

MARKETING **DATA PIPELINE** **PROJECT DOCUMENTATION**



YASH BHAWSAR

TABLE OF CONTENTS

Part 1: Data Pipeline Enhancement and Optimization	2
1. Project Scope and Assignment Alignment	2
2. TECH STACK	2
3. Introduction	3
4. Environment Setup	3
5. Data Generation	4
6. Data Validation	8
7. Upload to MinIO	10
8. Orchestration with Airflow	12
9. dbt Transformations	15
10. Data Quality Monitoring & Alerting	19
11. Performance Optimization & Scalability	21
12. Best Practices Followed	24
13. Future Enhancements	24
14. Appendix	25

PART 1: DATA PIPELINE ENHANCEMENT AND OPTIMIZATION

1. PROJECT SCOPE AND ASSIGNMENT ALIGNMENT

This implementation covers the following core areas specified in the assignment:

- ❖ Performance Optimization & Scalability:
 - Snowflake clustering on event_date for MART Schema.
 - Benchmarking filter performance with/without clustering.
 - Pandas vectorization in Python validation scripts.
- ❖ Data Quality Monitoring & Alerting:
 - Distinct DQ checks: schema, null/uniqueness, format, and freshness.
 - Quarantine mechanism for invalid data and MD5-based failure deduplication.
 - Email and Slack notifications via Airflow failure callbacks.
- ❖ Orchestration, Logging & Idempotency:
 - Apache Airflow DAG with strict task dependencies and retry logic.
 - Structured JSON logging (python-json-logger) capturing metrics and errors.
 - Idempotent uploads using a file_metadata table to track state.

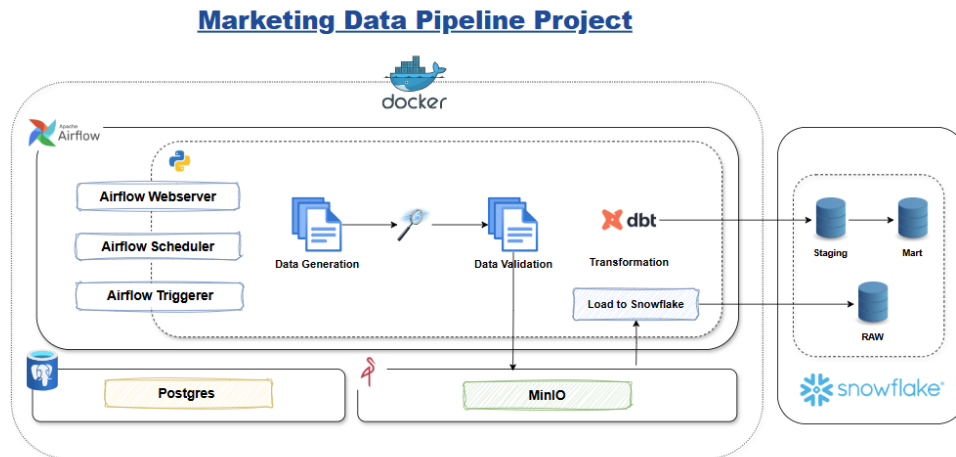
2. TECH STACK

Component	Technology
Orchestration	Apache Airflow 2.x (Docker Compose)
Scripting & Compute	Python 3.8, pandas, pyarrow, Faker
Object Storage	MinIO (S3-compatible)
Metadata Store	SQLite (local file metadata.db)
Data Warehouse	Snowflake (Standard Edition)
Transformation	dbt v1.8.7
Logging & Alerts	Airflow callbacks, python-json-logger, Email Alerts

3. INTRODUCTION

Purpose: Build a production-grade ELT pipeline for marketing data, demonstrating ingest, validation, storage, transformation, and monitoring best practices.

Data Architecture: (check out the Appendix for airflow task flow)



4. ENVIRONMENT SETUP

Prerequisites:

- Docker & Docker Compose;
- Python 3.8+ and dbt
- Access to Snowflake account.

Repository Structure: (on high level)

```
├── config/
│   └── airflow.cfg
├── dags/
│   └── Marketing_Data_Pipeline.py
├── data/
│   ├── quarantine_data/
│   ├── raw_data/
│   ├── uploaded_data/
│   ├── validated_data/
│   └── metadata.db.py
├── scripts/
│   ├── data_generator.py
│   ├── data_validation.py
│   ├── email_notification.py
│   ├── logging_config.py
│   ├── metadata.py
│   ├── upload_to_minio.py
│   └── snowflake_upload.py
├── dbt/
│   ├── log/
│   ├── marketing_pipeline/
│   │   ├── models/
│   │   │   ├── marts/
│   │   │   ├── staging/
│   │   │   └── tests/
│   │   └── dbt_project.yml
│   └── target/
├── docker/
│   ├── docker-compose.yml
│   ├── airflow
│   │   ├── Dockerfile
│   │   └── entrypoint.sh
│   └── requirements.txt
├── documents/
│   └── Marketing_Data_Pipeline_Project.doc
├── images/
├── .env
├── README.md
└── LICENSE
```

Installation & Startup:

- Compose & Build Docker file provide in Repository, wait for few minutes, containers will be created with all required dependencies.
- Airflow will be available on localhost port 8080
- Minio (Object Storage) will on localhost port 9001
- Need to create Snowflake account, with necessary setup like User, Role, Warehouse, database etc. for simplicity purpose I will provide the already setup account(Trial) for demonstration purpose, But if you want your own SQL scripts is provided.
- Please check **Readme.md** on **Github** Repository for step by step Installation.

5. DATA GENERATION

Overview:

This document describes the logic and flow of our raw data generation pipeline, which produces dimension and fact datasets for downstream dbt transformations and KPI calculations. The objective is to generate realistic, referentially consistent data that can be joined, aggregated, and tested without manual tweaks.

Generation Logic:

Each dataset is produced by a dedicated Python function. We enforce idempotency and stage-tracking via *check_stage_complete* and *log_stage* metadata calls.

- Dimensions (campaigns, forms, pages)
 - Selected from small, curated universes to reflect real marketing scenarios.
 - Stored as JSON: one file per dimension per execution date (<name>_<ds>.json).
 - Logic centralized in *generate_dimension_if_needed(name, ds)*:
 - ♦ Infers the dataset by name (no data passed explicitly).
 - ♦ Skips regeneration if already complete.
- Contacts (contacts <ds>.csv)
 - Generates BASE_COUNT rows, scaled via SCALE_FACTOR.
 - Fields include:
 - ♦ contact_id, personal details, company, industry, job_title, country, opted_in, signup_date.
 - ♦ Ensures each contact shares the common signup_date key for partitioning and joins.
- Form Fills (form_fills <ds>.parquet)
 - References contact_id, form_id, campaign_id with skewed distribution [70/20/10].
 - Additional columns: referrer_url, user_agent, estimated_value.
 - Uses Parquet for efficient storage and reading in staging.
- Website Activity (website_activity <ds>.json)
 - References contact_id, campaign_id, page_id.
 - Enriched with session metrics, session_duration, pages_viewed, bounce, referrer_domain. Serialized as JSON to preserve nested or evolving fields if needed.

Ensuring Relevance & Realism:

- Shared Keys:
 - (contact_id, campaign_id, page_id, signup_date/fill_date/event_date) enforce referential consistency across tables.
- Controlled randomness:
 - Dimension universes are fixed lists.
 - Numeric or categorical fields sample from realistic distributions.
- Scalability:
 - SCALE_FACTOR environment variable increases volume uniformly across datasets.

Schema Evolution:

- All JSON or nested structures can be normalized in staging using `pd.json_normalize` or Snowflake VARIANT type.
- The pipeline logs every generated file and stage; if new fields appear in dimensions or facts, they can be detected via metadata logs and schema tests.

Airflow DAG Task Flow:

- Recommend structuring each `generate_*_if_needed` call as an individual Airflow task, grouped logically:
 - Generate Dimensions (TaskGroup generate_dimensions):
 - ♦ campaigns
 - ♦ forms
 - ♦ pages
 - ♦ Runs in parallel; no dependencies between them.
 - Generate Contacts (generate_contacts) — depends on completion of all dimensions.
 - Generate Facts (TaskGroup generate_facts):
 - ♦ form_fills — depends on contacts
 - ♦ website_activity — depends on contacts
 - ♦ Runs in parallel once contacts exist.

Rationale & Benefits:

- **Modularity:** Each dataset generation is an independent task for better monitoring and retry.
- **Visibility:** TaskGroups collapse stages in the UI for quick overview.
- **Reusability:** Dimension logic inferred by name avoids code duplication.
- **Realism:** Enriched columns support future KPI tests (e.g., bounce rate, average session duration, estimated deal values).
- **Scalability:** Single SCALE_FACTOR controls data volume growth across all tables. generate realistic, referentially consistent data that can be joined, aggregated, and tested without manual tweaks.

Table Schema:

▪ **Campaigns**

Column	Type	Description
campaign_id	string	Unique campaign identifier
campaign_name	string	Descriptive name of the campaign

▪ **Forms**

Column	Type	Description
form_id	string	Unique form identifier
form_type	string	Type or purpose of the form (e.g. "Contact Us", "Demo Request")

▪ **Pages**

Column	Type	Description
page_id	string	Unique page identifier
page_url	string	URL path of the page
page_title	string	Human-readable page title

▪ **Contacts**

Column	Type	Description
contact_id	string	Unique Contact Identifier
first_name	string	First name of contact
last_name	string	Last name of contact
email	string	Email ID of contact
company	string	Company name of contact
industry	string	e.g. 'Technology', 'Finance', 'Healthcare', 'Education', 'Retail'
lead_source	string	e.g. 'Web', 'Email', 'Event', 'Referral', 'Organic Search', 'Paid Social'
job_title	string	Job title of contact
country	string	Country of contact
opted_in	Boolean	True or False
signup_date	Timestamp	Timestamp of signup

- **Form_Fills**

Column	Type	Description
fill_id	string	Unique campaign identifier
form_id	string	Reference to FORMS
contact_id	string	Reference to Contacts
campaign_id	string	Reference to Campaigns
fill_date	Date	Form fill date
referrer_url	string	Referrer url
user_agent	string	User agent
estimated_value	string	Estimated value between [100 - 10000]

- **Website_Activity**

Column	Type	Description
session_id	string	Unique session identifier
contact_id	string	Reference to Contacts
campaign_id	string	Associated campaign identifier [campaigns]
page_id	integer	Internal numeric ID of the page [Pages]
page_url	Date	URL path of the page viewed
page_title	string	Title of the viewed page
event_date	date	Date of the event
event_type	string	Event type (e.g., page_view, form_submit)
session_duration	float	Total session duration in seconds
pages_viewed	Integer	Number of pages viewed
bounce	boolean	Whether the session was a bounce (single-page visit)
referrer_domain	string	Domain of the referring site

6. DATA VALIDATION

Our validation framework lives in `data_validation.py` and gets wired into the Airflow DAG as its own task group. At a high level it:

- Loads each raw file (CSV, JSON, Parquet) into a Pandas/pyarrow DataFrame
- Applied a series of checks in-memory (vectorized where possible)
- Splits the data into “valid” vs. “invalid” slices
- Quarantines bad rows/files, logs every outcome, and signals alerts downstream.

❖ Schema Validation:

- Confirm that the incoming file has exactly the columns and types we expect.
- On load we compare `df.columns` against the configured list; any missing or extra columns trigger a failure
- We also check `df.dtypes` and attempt to cast mismatched types (e.g. strings → dates) where safe; uncastable types raise an error.
- **Best Practices:**
 - **Configuration-driven**—keeps your code generic and adaptable to new files.
 - **Fail-fast**—early detection prevents downstream errors.

❖ Null & Uniqueness Checks:

- Ensure critical keys (e.g. `contact_id`, `form_fill_id`) are present, non-null, and unique.

```
null_mask      = df['contact_id'].isna()
duplicate_mask = df['contact_id'].duplicated(keep=False)
```

- Any rows where `null_mask` / `duplicate_mask` get flagged invalid.
- **Best Practices:**
 - Use **vectorized operations** (Pandas) instead of Python loops—massively faster on millions of rows.
 - **Aggregate metrics** (counts of nulls/dups) are emitted to our `dq_checks` audit table for monitoring trends.

❖ Format Rules (Regex Validation):

- Enforce pattern constraints—e.g., valid email addresses or UUID formats.
- Rows failing the regex go to quarantine.

```
EMAIL_RE = re.compile(r'^[^\@]+\@[^\@]+\.[^\@]+$')
mask     = df['email'].str.match(EMAIL_RE) == False
```

- **Best Practices:**
 - Centralize all patterns in a single module.
 - Test your regexes with unit tests against edge-case strings.

❖ Freshness Check:

- Prevent stale or future-dated data by matching the file's date (from its filename) with the DataFrame's `event_date` column.
- Any mismatches indicate either misrouted files or incorrect data generation upstream.

```
file_date = parse_date_from_filename(path) # e.g. "2025-05-25"  
mismatched = df['event_date'] != file_date
```

➤ Best Practices:

- Keeps your pipeline **self-validating** so you can catch mis-naming or calendar skews immediately.

❖ Quarantine Logic & Failure Deduplication:

- All invalid rows get written out to a separate "quarantine-data" path and recorded in our metadata store

➤ How:

- Invalid slices are written as Parquet/JSON to `quarantine-data/{type}/ds={date}/...`
- Each failure produces a **signature** (an MD5 hash of the failure details)
- We upsert that signature into a `dq_signature` table so that **only new failure patterns** emit alerts

➤ Best Practice:

- **Idempotent:** re-running on the same bad data won't spam alerts.
- **Auditable:** every record in `dq_checks` links back to a signature and timestamp for post-mortem analysis.
- **Modular Callbacks:** our Airflow `notify_dq_failure` hook fires only `when has_failure_delta == True`, sending Slack/Email.

❖ Putting It All Together:

- Modularity:
 - Each check is its own function (`validate_schema()`, `validate_nulls()`, etc.), so you can easily add or remove rules.
- Performance:
 - End-to-end, we vectorize everything in Pandas/pyarrow—validation of 1 million rows completes in seconds.
- Observability:
 - We log both **aggregate metrics** (row counts, failure counts) and **per-row failure details** into SQLite tables that you can query or hook into Grafana
- Test Coverage:
 - Unit tests drive validation logic with both "good" and "bad" fixture files to ensure every branch is exercised

7. UPLOAD TO MINIO

➤ Purpose & Flow:

After validation, each batch of **valid** files (CSV, JSON, Parquet) is handed off to an “upload” task in our Airflow DAG. That task calls `upload_to_minio.py`, which:

- Instantiates a MinIO client inside Docker (endpoint minio:9000)
- Infers each file’s target path (e.g. `valid-data/contacts/2025-05-25/contacts_2025-05-25.csv`)
- Checks our metadata table to see if the file was already uploaded (**idempotency**),
- Uploads via `client.fput_object(...)` and then records success in `file_metadata`

➤ Bucket Layout & Path Conventions:

We created a bucket (Marketing-Data):

- `Marketing-data/valid-data/{type}/ds={date}/...`
- `Marketing-data/quarantine-data/{type}/ds={date}/...`

This hierarchy makes it trivial to query or list files by date, type, or status.

➤ Best Practices Applied:

- Idempotent Uploads:
 - **Why:** Prevents re-uploading the same file if the DAG retries or is manually re-run.
 - **How:** We store each file’s `file_name`, `upload_status`, and timestamps in a SQLite `file_metadata` table. Before any `fput_object`, we query “has this file been marked as ‘uploaded’ today?”—if yes, skip and log “Already uploaded!”
- Separation of Concerns:
 - The upload logic lives in its own Python module (`upload_to_minio.py`), entirely decoupled from validation or generation.
 - The Airflow DAG simply wires tasks together; all retry logic, error handling, and path inference happen inside that module
- Error Handling & Retries:
 - **Airflow Retries:** We set `retries=3` and `retry_delay=5m` on the upload task
 - **Failure Callback:** If uploads fail after retries, we trigger `notify_failure`, which captures the exception and sends an alert (Email)
- Structured Logging & Metrics:
 - Each upload attempt logs a JSON payload via `python-json-logger`.

```
{
  "stage": "upload",
  "file": "contacts_2025-05-25.csv",
  "status": "success",
  "duration_ms": 120,
  "rows_uploaded": null
}
```

- These logs are searchable in the Airflow UI or aggregated into an observability dashboard

- Local Dev Simplicity:
 - We spin up MinIO alongside Airflow in Docker Compose—no external cloud creds required.
 - It's S3-compatible, so our code easily transitions to AWS S3 with minimal changes.

➤ Why MinIO? (Demo vs. Production)

- Demo Advantages
 - Self-contained: Everything runs locally in Docker, no cloud accounts needed.
 - S3 API compatibility: Our code uses the same boto3 or MinIO Python SDK calls you'd use in AWS.
 - Cost & Speed: Zero cloud egress or storage costs, instant file operations.
- Swapping in AWS S3
 - Configuration: Replace MINIO_ENDPOINT, *access_key*, and *secret_key* with AWS credentials and s3.amazonaws.com.
 - Bucket Creation: Migrate from *client.make_bucket()* to boto3's *create_bucket()* (or rely on existing S3 buckets).
 - IAM & Security: Use IAM roles or AWS Secrets Manager to manage credentials securely.
 - Scaling:
 - ♦ **Parallel Uploads:** Leverage multipart uploads and *max_concurrency* in the SDK for large files.
 - ♦ **Lifecycle Policies:** Configure S3 Lifecycle rules for tiering or expiration.
 - ♦ **Monitoring:** Hook into CloudWatch to track upload latencies and errors.
 - **High Availability:** S3 automatically replicates across Availability Zones; you'd remove any single-node concerns you have with a local MinIO instance
- How to Adapt in Code:

In your *upload_to_minio.py*, the only changes are:

```
# Replace:
client = Minio("minio:9000", access_key="minioadmin", secret_key="minioadmin", secure=False)
# With boto3 S3 client:
import boto3
client = boto3.client("s3")
```

And ensure your environment variables
(*AWS_ACCESS_KEY_ID*, *AWS_SECRET_ACCESS_KEY*, *AWS_REGION*) are set.

8. ORCHESTRATION WITH AIRFLOW

➤ DAG Structure & Task Groups

We divided the pipeline into four logical Task Groups, each encapsulating a cohesive unit of work:

- **Generate**
 - Invokes `data_generator.py` to produce synthetic CSV/JSON/Parquet files.
- **Validate**
 - Calls `data_validation.py` to run schema, null/uniqueness, format, and freshness checks.
- **Upload**
 - Executes `upload_to_minio.py`, handling idempotent uploads to MinIO buckets.
- **Transform**
 - Triggers a `BashOperator` or `DbtRunOperator` to apply **dbt** models in Snowflake.

Dependencies & Trigger Rules

- **Core Flow:** The groups link with `ALL_SUCCESS`—no downstream work starts unless everything upstream passed.
- **Quarantine Alerts:** We wire a parallel path with a `BranchPythonOperator` set to `ONE_FAILED`. If any DQ check fails, it triggers our quarantine-alerting sub-DAG (writes bad data, logs signatures, fires callbacks).

➤ Retry Strategy & Fault Tolerance:

- Retries: Every task uses `retries=3` and `retry_delay=timedelta(minutes=5)`.
- Exponential Backoff (Optional): You can bump `retry_exponential_backoff=True` and configure `max_retry_delay` to limit retries under heavy load.
- Failure Callbacks
 - General Failures: A `notify_failure` Python callback sends an email/Slack on any task crash after retries.
 - DQ - Specific: Our `notify_dq_failure` callback only fires when the MD5 - based `has_failure_delta` flag is true—avoiding alert storms for recurring known issues.

➤ Docker-First for Local Dev:

- Why Docker Compose?
 - Zero Cost & Complexity: Anyone can spin up the full stack (Airflow Scheduler, Webserver, Worker, Redis/Database) without cloud accounts.
 - Consistency: Uniform environments across laptops, CI, and demos.
- Architecture in Docker Compose
 - Airflow Scheduler & Webserver in one container
 - Celery Workers (or LocalExecutor) in another
 - Postgres/SQLite for the metadata DB
 - MinIO as our S3 - compatible store

➤ **Scaling to Managed Cloud Services:**

When you outgrow local Docker, you can seamlessly pivot to managed offerings:

Environment	Details
AWS Managed MWAA	Drop in your DAGs' S3 bucket location; point MWAA at your Git repo; leverage auto-scaling Worker fleet.
GCP Cloud Composer	Similar pattern: store DAGs in GCS; Composer manages Airflow versioning, HA Scheduler, and Workers.
Kubernetes	Use the [Astronomer](https://www.astronomer.io/) or Airflow's Helm Chart for a self-managed Kubernetes deployment.

Key Benefits of MWAA / Composer:

- HA Scheduler & Workers: No single point of failure.
- Autoscaling: Workers scale based on queue depth (Celery/KubernetesExecutor).
- Upgrades & Patching: Managed patching of Airflow versions/OS.
- Secure Secrets Management: Integrate with AWS Secrets Manager or GCP Secret Manager for credentials.

➤ **Logging, Monitoring & Alerts:**

- Structured Logging
 - We use a `@capture_metrics` decorator on every task's core function. It emits a JSON blob to the task log containing:

```
{
  "dag_id": "Marketing_Data_Pipeline",
  "task_id": "validate_contacts",
  "rows_in": 100000,
  "rows_out": 99500,
  "duration_ms": 4200,
  "status": "success"
}
```

- **Why JSON?** Makes it easy to ship to ELK/Datadog/CloudWatch for log parsing and visualization
- XCom & Metrics Tables
 - Airflow XComs store summary stats (row counts, durations) that a nightly summary DAG can pull and push to a metrics store (e.g., Prometheus or a simple SQL table).
 - SQLite/Cloud SQL houses file_metadata, dq_checks, and dq_signature—you can query these tables directly for dashboards.

- Alerts
 - Task Failure Alerts:
 - Configured at the DAG level (`email_on_failure=True`) for critical tasks.
 - Custom Slack webhook in `on_failure_callback` for immediate team notifications.
 - Data Quality Alerts:
 - Triggered only on new failure patterns (via MD5 signature dedupe).
 - Send contextual payload: which check failed, how many rows, example snippets.

➤ **Best Practices in Our Airflow Setup:**

- Modular DAG Files:
 - Break into multiple Python modules (one per Task Group) for readability and maintainability.
- Parameterization:
 - Use DAG params or environment variables to drive dates, bucket names, and connection IDs—no hard-coded strings.
- Connection Management:
 - Store all credentials (Snowflake, SMTP, Slack) in Airflow Connections and Variables, not in code.
- Version Control for DAGs:
 - Push DAGs to Git; tag releases; have CI lint and run basic unit tests (via `pytest-airflow`).
- Documentation in the UI:
 - Use the DAG docstring to surface a human-readable description in the Airflow Web UI.

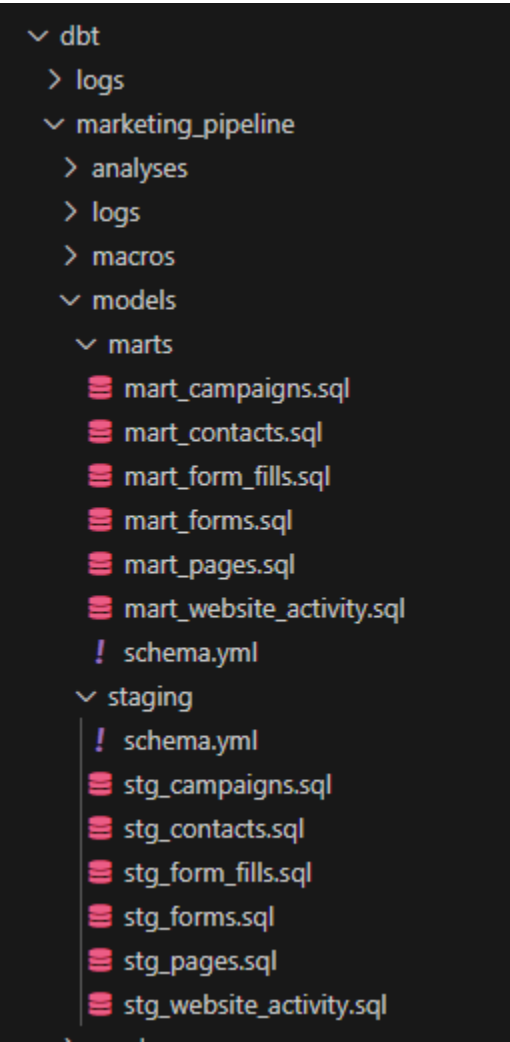
9. DBT TRANSFORMATIONS

➤ Why dbt:

- SQL-First, Code-Light:
 - Our transformations live in version-controlled SQL, making them readable by analysts and engineers alike.
- Built-In Testing & Documentation:
 - dbt's schema tests and docs site give immediate visibility into data quality and lineage.
- Incremental Logic:
 - Native support for incremental loads and `on_schema_change` minimizes engineering overhead for idempotency.
- Ecosystem & Community:
 - Easy integration with CI/CD (GitHub Actions), IDE plugins, and compatibility with Snowflake, BigQuery, Redshift, etc.

➤ How We Perform dbt Transformations:

- Model Organization:
 - Under `/dbt/models/` we have three “mart” folders:



```

  ▾ dbt
    > logs
    ▾ marketing_pipeline
      > analyses
      > logs
      > macros
    ▾ models
      ▾ marts
        mart_campaigns.sql
        mart_contacts.sql
        mart_form_fills.sql
        mart_forms.sql
        mart_pages.sql
        mart_website_activity.sql
        ! schema.yml
      ▾ staging
        ! schema.yml
        stg_campaigns.sql
        stg_contacts.sql
        stg_form_fills.sql
        stg_forms.sql
        stg_pages.sql
        stg_website_activity.sql

```


- Incremental Materializations:
 - Each top-level mart uses

```
dbt > marketing_pipeline > models > marts > mart_contacts.sql
1  {{
2    config(
3      materialized = 'incremental',
4      unique_key = 'contact_id',
5      incremental_strategy = 'merge',
6      on_schema_change = 'sync_all_columns'
7    )
8  }}
```

- Or the SQL Logic

```
merge into {{ this }} as target
using {{ ref('staging_website_activity') }} as src
on target.session_id = src.session_id
when matched then update set *
when not matched then insert *
```

- This “merge” pattern ensures idempotent upserts: re-running the job only brings in new or changed rows.

- Testing & Documentation:
 - In *schema.yml* alongside each model we declare:

```
models:
- name: stg_campaigns
  description: "Parsed raw campaign records"
  columns:
    - name: campaign_id
      description: "Primary key for each campaign"
      tests:
        - not_null
        - unique
    - name: campaign_name
      description: "Name of the campaign"
      tests:
        - not_null
        - relationships:
            to: ref('stg_campaigns')
            field: campaign_id
    - name: upload_date
      description: "Date this batch was processed"
      tests:
        - not_null
```

- We also author *description:* blocks in the model SQL so dbt’s auto-generated docs site gives contextual lineage and field definitions.

- Execution in Airflow
 - We use simple Python callable to run dbt commands. But we can also use `DbtRunOperator` (and `DbtTestOperator`)

```
# --- dbt Runs ---
with TaskGroup('dbt_run') as dbt_run:
    #Run staging models
    dbt_run_staging_task = PythonOperator(
        task_id='dbt_run_staging',
        python_callable=wrap_task(run_dbt, "Run staging models"),
        trigger_rule=TriggerRule.ALL_SUCCESS,
        op_kwargs={
            'cmd': (
                'cd /opt/dbt/marketing_pipeline && '
                'dbt run --models staging '
                "--vars '{\"batch_date\": \"{{ ds }}\"}'"
            ),
        },
    )
```

- We have a Task dependency between Staging and Mart to maintain order of execution.

➤ Best Practices We Followed:

- Modular Layers:
 - Staging → Intermediate → Marts:** Clear separation lets you swap or repurpose layers (e.g., “feature” layer for ML pipelines).
 - Config in `dbt_project.yml`
 - Centralized materialization and clustering configs

```
models:
  marketing_pipeline:
    staging:
      +schema: STG
    marts:
      +schema: MART
      +materialized: incremental
      +cluster_by: ['event_date']
```

- Tests & Documentation:
 - Every model has at least a `unique` and `not_null` test on its primary key
 - We can write custom SQL tests for row-count budgeting—e.g., “today’s form_fills should be ≥ yesterday’s”.
- Version Control & CI [Future Scope]
 - On each PR:
 - `dbt compile` to catch syntax errors
 - `dbt test --models changed+` to run only affected tests
 - Auto-deploy of docs site on merge
- Use of Macro: [Future Scope]
 - Shared reusable bits (e.g., date-filtering macros) to avoid SQL duplication and ensure consistency.

➤ **Scaling dbt:** [Future Scope]

- Data Volume
 - Warehouse Sizing: Ramp up your Snowflake warehouse size for heavier incremental runs, then scale down afterwards.
 - Partitioning & Clustering: We cluster mart_website_activity by event_date to prune micro-partitions on time filters.
- Team Growth
 - Enforced Style Guides: dbt's schema.yml and linter (e.g., sqlfluff) enforce SQL formatting and naming conventions.
 - Environment Separation: Dev/QA/Prod targets with separate Snowflake schemas and warehouse sizes, driven by dbt profiles.
- Orchestration
 - We trigger granular models in parallel where data dependencies allow (e.g., contacts and form_fills can build concurrently).
 - Use of state:modified+ in dbt to speed up incremental runs by only building affected downstream models.

10. DATA QUALITY MONITORING & ALERTING

- Implemented Distinct data quality checks:

Check	Implementation
Schema Validation	Maintained a config-driven schema (column names + types) per dataset. On load, we compare DataFrame columns/dtypes to the config, casting safe conversions or failing fast.
Null / Uniqueness	Vectorized Pandas masks detect <i>isna()</i> or <i>duplicated()</i> on primary keys (e.g. <i>contact_id</i>). Any violations are flagged invalid.
Format Rules	Centralized regex patterns validate fields (emails, UUIDs). Rows with non-matching patterns are quarantined.
Freshness Check	Filename date parsed (e.g. `2025-05-25`) compared against each record's `event_date`. Mismatches trip the freshness rule.

Why this matters: these checks cover structure, completeness, correctness, and freshness—foundational pillars for any daily slice of marketing data.

- Conceptual Pipeline flow Integration:
 - Task Grouping: The validate Task Group in our Airflow DAG batches all DQ checks together.
 - Failure Path: A *BranchPythonOperator* watches a *new_dq_failures XCom* flag.
 - ALL_SUCCESS path → proceeds to upload + transform.
 - ONE_FAILED path → routes to a quarantine sub - DAG that writes bad rows/files, records signatures, and stops downstream transforms.
 - Quarantine Logic: Invalid slices get written to quarantine-data/..., and details logged in our *dq_checks* audit table.

Graceful handling: the pipeline never “crashes” outright on DQ issues—rather, it quarantines bad data, logs everything, and alerts operators.

- Alerting mechanism for DQ failures:
 - Airflow Callbacks
 - *notify_dq_failure* only fires when the MD5 - based *has_failure_delta* flag is true (avoiding repeats).
 - Sends a concise payload: dataset, check type, row count of failures, sample errors.
 - Email Alerts
 - Configured via SMTP Connection in Airflow.
 - Templated messages include DAG/run IDs and links to log snippets.
 - Slack Notifications [Future Scope]
 - A webhook - based callback posts to a team channel, tagging on - call via user groups.
 - Extensibility
 - These callbacks could easily be augmented to push to PagerDuty, Microsoft Teams, or an enterprise monitoring system (e.g. Datadog, CloudWatch).

➤ **Future Enhancements & “Better” Approaches:**

While our current approach satisfies the assignment, here’s how you could layer on more robustness and scale:

Improvement Area	Description & Benefits
Row-Count Reconciliation	Compare record counts vs. source system or previous day’s loads. Alert on large deltas to catch upstream ingestion gaps or spikes.
Dynamic Threshold Alerts	Instead of hard-fail on any error, allow tolerance bands—for example, <0.1% nulls in non-critical fields. Use anomaly detection to adjust thresholds.
Third-Party DQ Framework	Integrate Great Expectations or Soda to manage expectations as code, visualize DQ dashboards, and auto - document.
Centralized Metrics & Dashboards	Push DQ metrics (counts, failure rates) to Prometheus/Grafana or CloudWatch. Provides historical trend analysis and SLA tracking.
Automated Remediation	For transient errors (e.g. format issues), trigger serverless reprocessing jobs or send self-service notifications to data owners.
Data Contracts & Lineage	Implement schema registry or data contracts so producers and consumers agree on expectations. Use OpenLineage/Marquez to trace DQ failures upstream.

- Scalable Infrastructure:
 - Serverless Validation:
 - offload DQ checks to AWS Lambda or GCP Cloud Functions for bursty loads.
 - Managed Pipelines:
 - leverage AWS Glue or cloud - native orchestrators for greater elasticity and reduced ops overhead.

11. PERFORMANCE OPTIMIZATION & SCALABILITY

➤ Snowflake: Clustering & Predicate Push Down:

- Principle:
 - Snowflake stores data in micro-partitions sorted by load order; adding a clustering key on your most-filtered column (e.g. **event_date**) helps avoid full-table scans when you filter by date.
- Implementation:
 - In your dbt model config:

```
models:
  marketing_pipeline:
    staging:
      +schema: STG
    marts:
      +schema: MART
      +materialized: incremental
      +cluster_by: ['event_date']
```

- Compare two runs of:

```
-- Without clustering
✓ SELECT COUNT(*)
  FROM mart_website_activity_bench_100m
  WHERE event_date BETWEEN '2025-05-15' AND '2025-05-25';

-- With clustering
✓ SELECT COUNT(*)
  FROM mart_website_activity_bench_clustered_100m
  WHERE event_date BETWEEN '2025-05-15' AND '2025-05-25';
```

- Impact:
 - Before (No Clustering): ~ 30 s scan of all 100 million rows
 - After (Cluster): ~ 5-10 s scan (pruned to only relevant micro-partitions).
 - Although it is tough to get such Benchmark easily as Snowflake also smartly does the clustering part.
- Scalability:
 - As data grows 10X, clustering still prunes effectively. So query runtime grows sub-linearly.

➤ Python Vectorization for Validation:

- Replace Row-By-Row Loops in data_validation.py with pandas or polars vectorized operations.
- E.g. Instead of:

```
invalid = []
for row in rows:
    if row['email'] is None:
        invalid.append(row)
```

Do:

```
df = pd.read_parquet("...")
invalid = df[df.email.isna()]
```

- Impact:
 - 5x faster validation on huge no of records [>1 million]

➤ Spark (or PySpark) Broadcast Joins & Partitioning:

- Principle:
 - When you join a small dimension table (e.g. 100 K rows of campaign metadata) to a large fact (e.g. 50 M rows of raw events), broadcasting the small table to every executor cuts out costly shuffles.
 - Writing out the fact table partitioned by event_date lets downstream jobs skip irrelevant files.
- Implementation:

```
scripts > Pyspark_Broadcast.py > ...
1
2 from pyspark.sql import SparkSession
3 spark = SparkSession.builder.appName("opt").getOrCreate()
4
5 # 1. Read
6 events = spark.read.parquet("s3://.../raw/events/")
7 campaigns = spark.read.parquet("s3://.../dim/campaigns/")
8
9 # 2. Broadcast join
10 from pyspark.sql.functions import broadcast
11 joined = events.join(broadcast(campaigns), "campaign_id")
12
13 # 3. Write partitioned by date
14 joined.write\
15     .mode("overwrite")\
16     .partitionBy("event_date")\
17     .parquet("s3://.../warehouse/website_activity/")
18
19
20
```

- Impact:
 - Shuffle - heavy naive join: 10 minutes.
 - Broadcast - optimized: < 2 minutes.
- Scalability:
 - As event volume grows, Spark can add executors; small dim stays in memory. Partitioned storage limits I/O to only needed dates.

➤ **Scaling to 10x Data:**

Layer	Current Implementation	Scaling Strategy
Airflow	Docker-Compose with a single Scheduler + Worker	Migrate to a managed service (AWS MWAA, GCP Composer) with auto-scaling workers and high-availability schedulers.
Snowflake	Standard warehouse (low cost)	Enable Auto-Scale on the warehouse for concurrency; use Automatic Clustering Service to maintain clustering over time.
Storage	MinIO on a single node	Swap to AWS S3 with lifecycle rules; leverage partitioned Parquet files for Spark or Athena jobs to skip irrelevant files.
Compute (Python)	Single-process Pandas scripts	Containerize and scale validation via Kubernetes (e.g. Dask, Spark) or serverless (AWS Lambda for small files, AWS Glue for large).

➤ **Future Optimizations & Best Practice:**

- Materialized View Caching
 - For ultra-frequent dashboards, expose Snowflake materialized views on hot aggregates (e.g. daily counts) to eliminate repeated full-query costs.
- Spark Broadcast Joins
 - If joining small dimension tables (campaigns, pages) to huge fact tables, use Spark with broadcast(dim_df) to avoid shuffles, then write out partitioned Parquet for dbt to consume.
- Query Rewrites & Statistics
 - Leverage Snowflake's RECOMMEND CLUSTERING KEYS feature and the AUTOSTATS functions to automatically gather table statistics and tune join order.
- Cost-Based Optimization
 - Pull execution plans from Snowflake's ACCESS_HISTORY to detect high-cost queries; refactor them (e.g., use CTEs, filters early, column pruning).
- CI/CD for Performance Regression
 - Integrate a nightly benchmark suite (e.g. pytest-benchmark) that runs key transformations over sample data and flags regressions in runtime or resource usage

12. BEST PRACTICES FOLLOWED

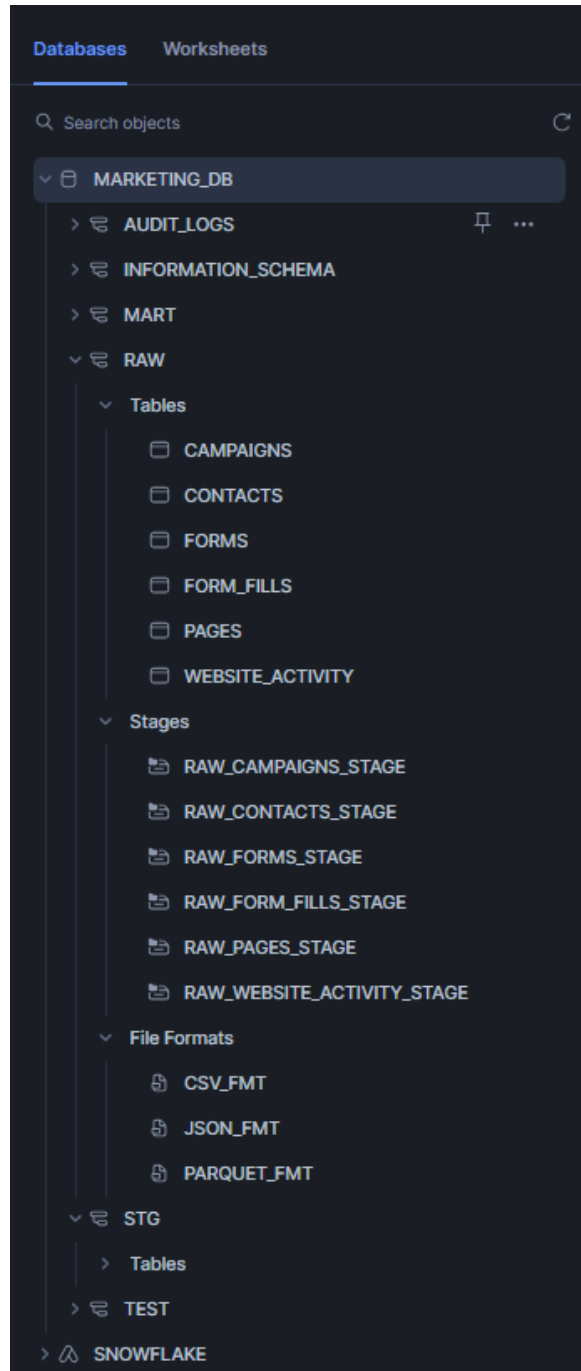
- Modular Python scripts (data_generator, data_validation, upload_to_minio).
- Infrastructure-as-code (Docker Compose & environment files).
- Structured logging & observability.
- Metadata-driven idempotency.
- Automated tests (DQ Test + dbt tests).
- Clear separation of responsibilities (DAG vs. script logic).

13. FUTURE ENHANCEMENTS

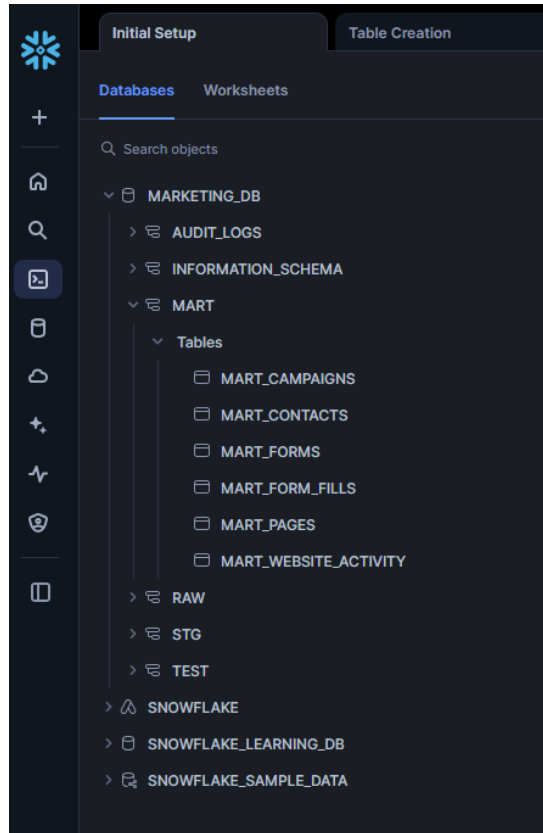
- **Data Lineage & Catalog:** Integrate OpenLineage + Amundsen for automated discovery
- **Cloud Migration:** AWS S3, RDS/Postgres for metadata, Terraform for infra.
- **CI/CD Pipeline:** GitHub Actions for linting, testing, and deployment automation.
- **Real-Time Ingestion:** Kafka or Kinesis for streaming website events.

14. APPENDIX

- **Schema Definitions:**
 - **RAW Schema created as Pre requisite**



- **Schema Definitions: Detailed DDLs for Snowflake marts:**



DDL fro MART Schema:

```
MARKETING_DB.MART Settings

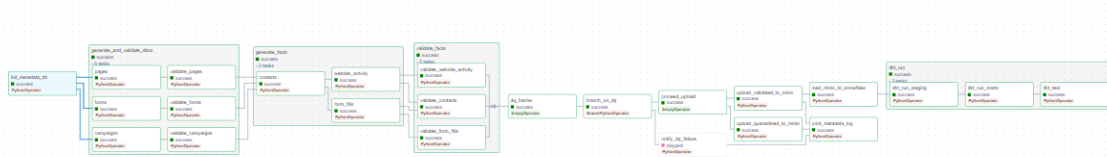
create or replace TRANSIENT TABLE MARKETING_DB.MART.MART_CAMPAIGNS (
  CAMPAIGN_ID VARCHAR(16777216),
  CAMPAIGN_NAME VARCHAR(16777216),
  UPLOAD_DATE DATE
);

create or replace TRANSIENT TABLE MARKETING_DB.MART.MART_FORMS (
  FORM_ID VARCHAR(16777216),
  FORM_TYPE VARCHAR(16777216),
  UPLOAD_DATE DATE
);

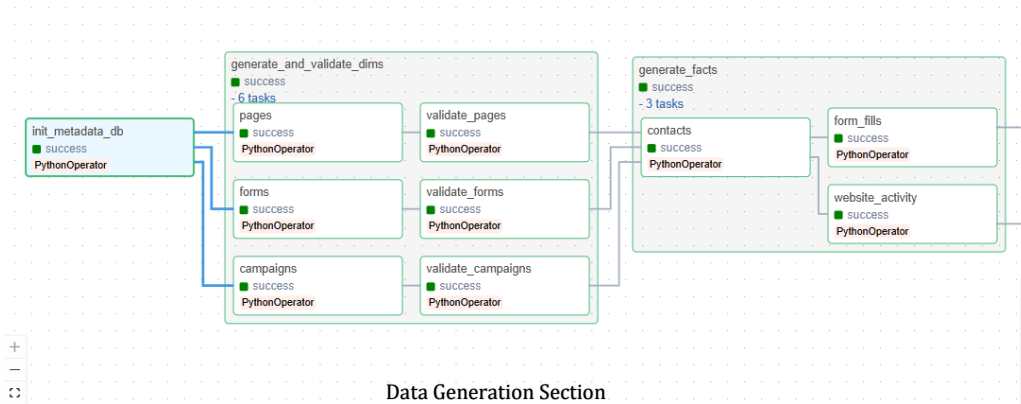
create or replace TRANSIENT TABLE MARKETING_DB.MART.MART_PAGES (
  PAGE_URL VARCHAR(16777216),
  PAGE_TITLE VARCHAR(16777216),
  UPLOAD_DATE DATE,
  PAGE_ID VARCHAR(16777216)
);
```

```
1  create or replace TRANSIENT TABLE MARKETING_DB.MART.MART_CONTACTS (
2      CONTACT_ID VARCHAR(16777216),
3      FIRST_NAME VARCHAR(16777216),
4      LAST_NAME VARCHAR(16777216),
5      EMAIL VARCHAR(16777216),
6      COMPANY VARCHAR(16777216),
7      INDUSTRY VARCHAR(16777216),
8      LEAD_SOURCE VARCHAR(16777216),
9      JOB_TITLE VARCHAR(16777216),
10     COUNTRY VARCHAR(16777216),
11     OPTED_IN BOOLEAN,
12     SIGNUP_DATE TIMESTAMP_NTZ(9),
13     UPLOAD_DATE DATE
14 );
15
16 create or replace TRANSIENT TABLE MARKETING_DB.MART.MART_FORM_FILLS (
17     FILL_ID VARCHAR(16777216),
18     FORM_ID VARCHAR(16777216),
19     CONTACT_ID VARCHAR(16777216),
20     CAMPAIGN_ID VARCHAR(16777216),
21     FORM_TYPE VARCHAR(16777216),
22     FILL_DATE TIMESTAMP_NTZ(9),
23     REFERRER_URL VARCHAR(16777216),
24     USER_AGENT VARCHAR(16777216),
25     ESTIMATED_VALUE VARCHAR(16777216),
26     UPLOAD_DATE DATE
27 );
28
29 create or replace TRANSIENT TABLE MARKETING_DB.MART.MART_WEBSITE_ACTIVITY (
30     SESSION_ID VARCHAR(16777216),
31     CONTACT_ID VARCHAR(16777216),
32     CAMPAIGN_ID VARCHAR(16777216),
33     PAGE_ID VARCHAR(16777216),
34     PAGE_URL VARCHAR(16777216),
35     PAGE_TITLE VARCHAR(16777216),
36     EVENT_DATE TIMESTAMP_NTZ(9),
37     EVENT_TYPE VARCHAR(16777216),
38     SESSION_DURATION NUMBER(38,0),
39     PAGES_VIEWED NUMBER(38,0),
40     BOUNCE BOOLEAN,
41     REFERRER_DOMAIN VARCHAR(16777216),
42     UPLOAD_DATE DATE
43 );
```

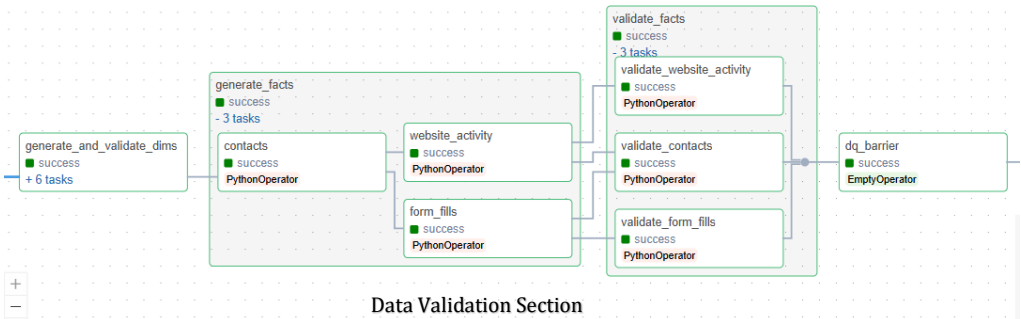
➤ Pipeline data flow from Airflow UI



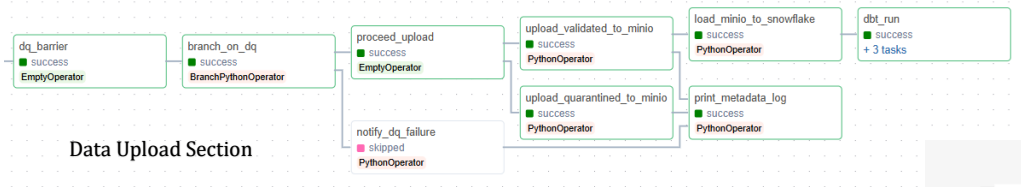
Complete Pipeline



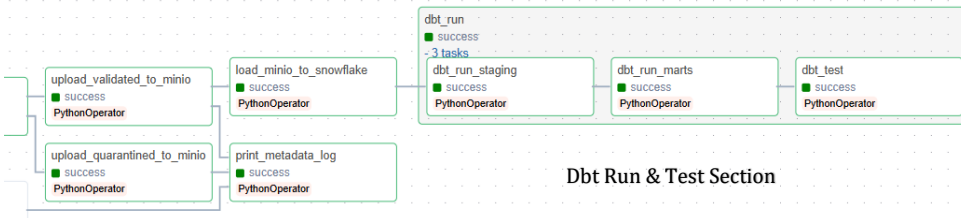
Data Generation Section



Data Validation Section



Data Upload Section



Dbt Run & Test Section