# Homework #5 Report: Airflow DAGs with Snowflake Integration

**Abstract**

This report documents the implementation of Homework #5. Apache Airflow orchestrates an ETL and a forecasting pipeline with Snowflake as the data warehouse. Screenshots demonstrate configuration and successful runs. Concise, relevant code snippets are embedded per the assignment requirement.

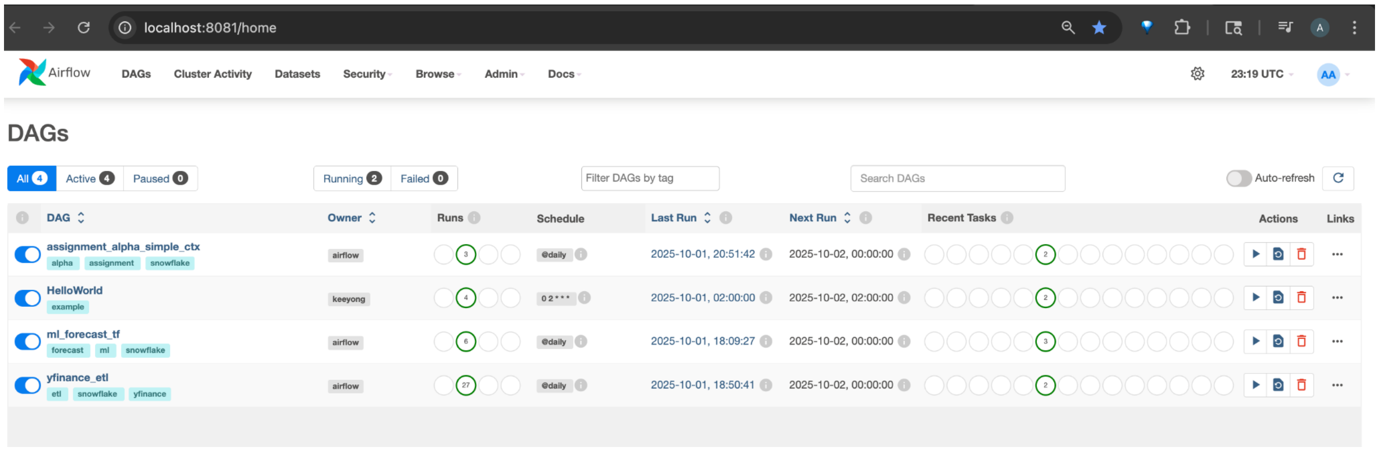
## I. Introduction

This submission shows:

- Airflow DAGs (`yfinance_etl`, `ml_forecast_tf`, and a simplified assignment DAG).
- Proper use of Airflow **Connections** and **Variables**.
- SQL **transaction** semantics for full refresh.
- Evidence of successful runs via the Airflow UI logs.

## II. Airflow Web UI: DAGs Loaded

Figure 1 confirms four DAGs are discovered by the scheduler and are active.

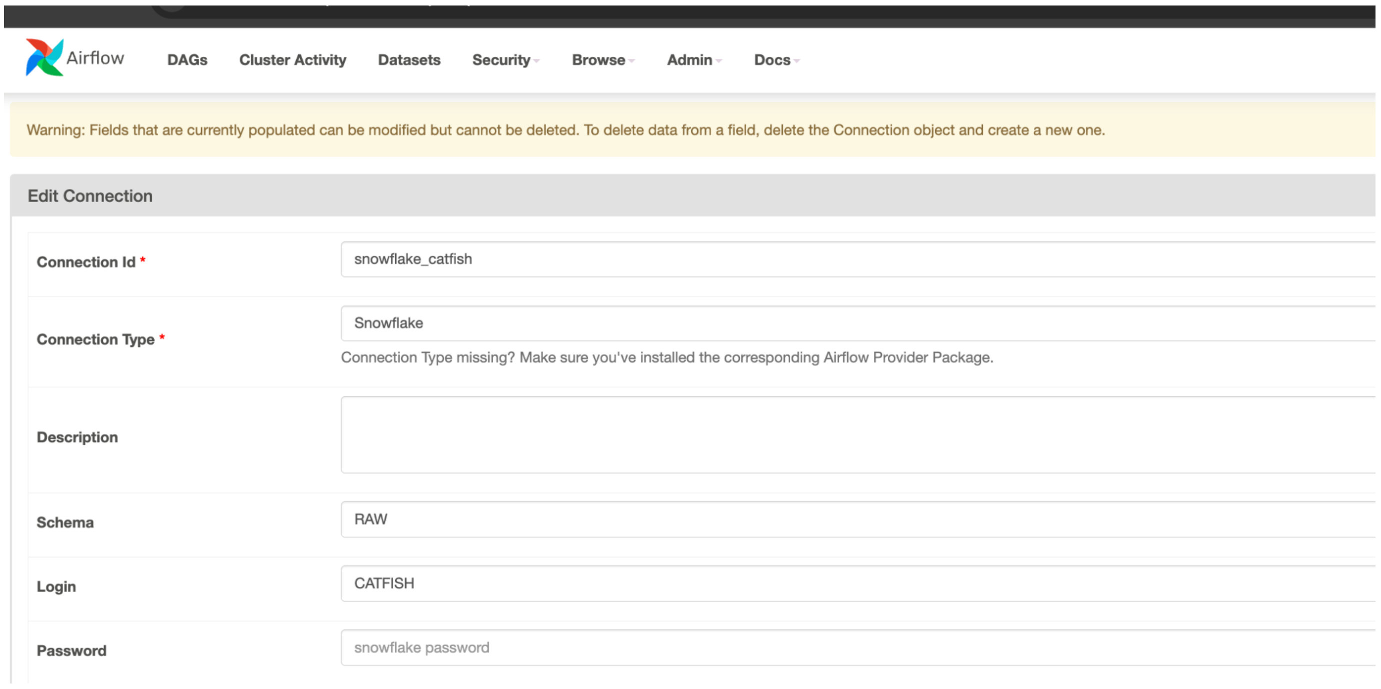


| DAG | Owner | Runs | Schedule | Last Run | Next Run | Recent Tasks | Actions | Links |
|---|---|---|---|---|---|---|---|---|
| assignment_alpha_simple_ctx<br>alpha assignment snowflake | airflow | 3 | @daily | 2025-10-01, 20:51:42 | 2025-10-02, 00:00:00 | 2 | | ··· |
| HelloWorld<br>example | keeyong | 4 | 0 2 * * * | 2025-10-01, 02:00:00 | 2025-10-02, 02:00:00 | 2 | | ··· |
| ml_forecast_tf<br>forecast ml snowflake | airflow | 6 | @daily | 2025-10-01, 18:09:27 | 2025-10-02, 00:00:00 | 3 | | ··· |
| yfinance_etl<br>etl snowflake yfinance | airflow | 27 | @daily | 2025-10-01, 18:50:41 | 2025-10-02, 00:00:00 | 2 | | ··· |

Fig. 1: Airflow Web UI showing all DAGs loaded.

## III. Admin → Connection (Snowflake)

The connection `snowflake_catfish` was created with type *Snowflake*. Account, warehouse, database, schema, and role are set in the Extra/fields as required.



Warning: Fields that are currently populated can be modified but cannot be deleted. To delete data from a field, delete the Connection object and create a new one.

Edit Connection

| | |
|---|---|
| Connection Id * | snowflake_catfish |
| Connection Type * | Snowflake<br>Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package. |
| Description | |
| Schema | RAW |
| Login | CATFISH |
| Password | snowflake password |

Fig. 2: Airflow Connection configuration for Snowflake.

*How to Configure the Airflow Snowflake Connection (UI)*

**Path:** `Admin → Connections → + Add a new record`

1) **Connection Id:** `snowflake_catfish` (must match the code)
2) **Connection Type:** `Snowflake`
3) **Login:** Snowflake user, e.g., `CATFISH`
4) **Password:** Snowflake password for the above user
5) **Schema:** Default schema for this connection, e.g., `RAW`
6) **Extra (JSON):** Provide account/warehouse/database/role (and optionally schema):

```
{
  "account":   "xxxx-xxxx",
  "warehouse": "COMPUTE_WH",
  "database":  "YOUR_DB",
  "schema":    "RAW",
  "role":      "SYSADMIN"
}
```

*Why this matters.*
The DAG uses `BaseHook.get_connection("snowflake_catfish")` to read these values at runtime. Keeping these secrets in the Airflow UI avoids hardcoding and makes deployments portable across environments.

*Where it is used in code (exact snippet).*

*How the Snowflake Connection is Used in Assignment_5.py*

In the DAG file (`Assignment_5.py`), the Snowflake connection is referenced through an Airflow Variable, which points to the connection ID defined in the Admin → Connections page.

*Code Snippet (from Assignment_5.py).*

```
from airflow.providers.snowflake.hooks.snowflake import SnowflakeHook
from airflow.models import Variable

# Connection Id is read from Airflow Variable, defaults to 'snowflake_catfish'
SNOWFLAKE_CONN_ID = Variable.get("SNOWFLAKE_CONN_ID", default_var="
    snowflake_catfish")
```

```

@task
def load_full_refresh():
    hook = SnowflakeHook(snowflake_conn_id=SNOWFLAKE_CONN_ID)
    conn = hook.get_conn()
    cur = conn.cursor()

    cur.execute("BEGIN")
    # transactional load into RAW schema
    cur.executemany(insert_sql, rows)
    cur.execute("COMMIT")
```

- The Variable `SNOWFLAKE_CONN_ID` allows flexibility: in the UI you can point it to any valid Airflow Connection without changing code.
- The actual credentials and account/warehouse/database details come from the Airflow Connection with ID `snowflake_catfish`.
- The task wraps operations inside `BEGIN`/`COMMIT` ensuring that inserts are atomic. If an exception occurs, a `ROLLBACK` can be executed for consistency.

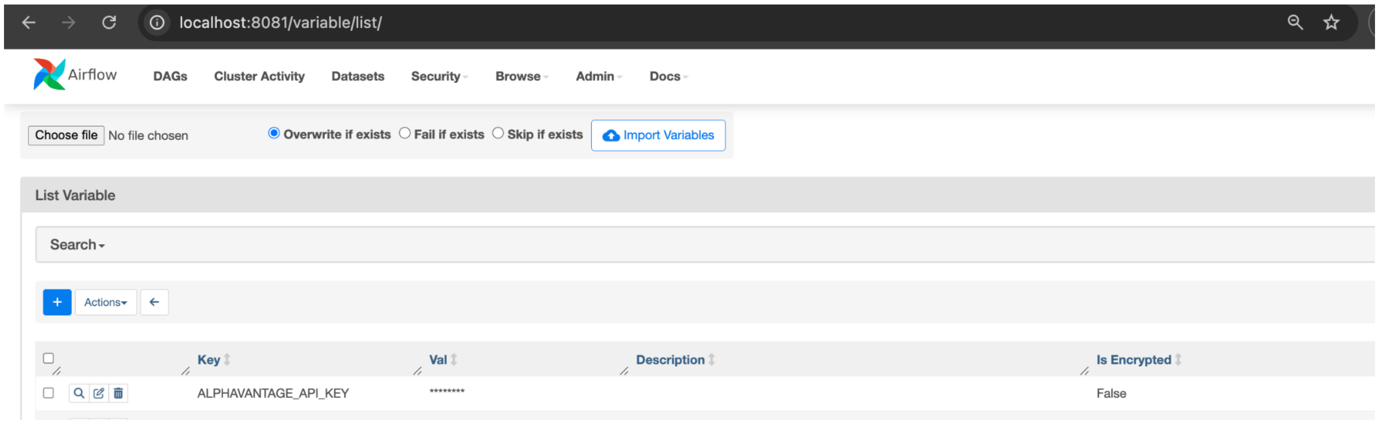
Thus, the connection part in your assignment is implemented via `SnowflakeHook`, pulling configuration securely from Airflow's Admin → Connections.

*Quick validation.*

After saving the connection, trigger a DAG run and check task logs. If the `conn_id` is wrong or fields are missing, Airflow will raise a clear error in the task log (e.g., "The conn_id `snowflake_catfish` isn't defined").

## IV. Admin → Variables

An example variable (`ALPHAVANTAGE_API_KEY`) is configured and consumed in code.



| Key | Val | Description | Is Encrypted |
|---|---|---|---|
| ALPHAVANTAGE_API_KEY | ******** | | False |

Fig. 3: Airflow Variables page.

### *How to Add and Use Airflow Variables*

**Path:** `Admin → Variables → +`

1) Click **+** and set:
   - **Key:** `ALPHAVANTAGE_API_KEY`
   - **Val:** `<your-actual-api-key>`
2) Save and verify it appears in the list (Figure 4 shows it in use via successful runs).

*Where it is used in code (exact snippet).*

```
from airflow.models import Variable

API_KEY = Variable.get("ALPHAVANTAGE_API_KEY")   # raises if missing

# Optionally make it non-fatal with a default:
# API_KEY = Variable.get("ALPHAVANTAGE_API_KEY", default_var=None)
```

*JSON and typed Variables (optional).*
When storing structured config (symbols, thresholds, etc.), keep it as JSON and parse it in code:

```
import json

symbols_json = Variable.get("stock_symbols", default_var='["AAPL","MSFT","TSLA"]')
SYMBOLS = json.loads(symbols_json)   # -> ["AAPL","MSFT","TSLA"]

lookback_days = int(Variable.get("lookback_days", default_var="365"))
```

*Best practices.*

- Use Variables for *non-secret* runtime knobs (symbols, lookback windows, flags).
- Keep *secrets* (passwords, tokens) in *Connections* or a secrets backend (Vault, AWS SM, etc.).
- Validate presence of critical Variables early in the DAG and fail fast with a clear message.

## V. Relevant Code Snippets

The following excerpts from `Assignment_5.py` capture the important mechanics: connection retrieval, ETL/ML orchestration, and transactional load into Snowflake. The full file is provided in the project and may be included as an appendix if required.

### A. Core DAG Logic

```
# dags/assignment_alpha_simple_ctx.py
from airflow import DAG
from airflow.decorators import task
from airflow.models import Variable
from airflow.providers.snowflake.hooks.snowflake import SnowflakeHook

from datetime import datetime
import requests

# Read the Snowflake connection id from a Variable
SNOWFLAKE_CONN_ID = Variable.get("SNOWFLAKE_CONN_ID", default_var="
    snowflake_catfish")

with DAG(
    dag_id="assignment_alpha_simple_ctx",
    start_date=datetime(2025, 9, 1),
    schedule="@daily",
    catchup=False,
    tags=["assignment", "alpha", "snowflake"],
) as dag:

    @task
    def extract() -> list[dict]:
        """Pull compact daily OHLCV from Alpha Vantage for a small symbol list."""
        api_key = Variable.get("ALPHAVANTAGE_API_KEY")   # must exist
        symbols_csv = Variable.get("ALPHAVANTAGE_SYMBOLS", default_var="AAPL,MSFT"
            )
        symbols = [s.strip().upper() for s in symbols_csv.split(",") if s.strip()]

        base = "https://www.alphavantage.co/query"
        rows: list[dict] = []

        for sym in symbols:
            r = requests.get(
                base,
                params={
                    "function": "TIME_SERIES_DAILY",
                    "symbol": sym,
                    "outputsize": "compact",
                    "datatype": "json",
                    "apikey": api_key,
                },
                timeout=30,
            )
            r.raise_for_status()
            data = r.json()
            ts = data.get("Time Series (Daily)")
            if not ts:
                continue
            for ds, f in ts.items():
                try:
                    rows.append(
                        {
                            "SYMBOL": sym,
                            "PRICE_DATE": ds,
                            "OPEN": float(f.get("1. open", 0) or 0),
                            "HIGH": float(f.get("2. high", 0) or 0),
                            "LOW":  float(f.get("3. low",  0) or 0),
                            "CLOSE": float(f.get("4. close", 0) or 0),
                            "VOLUME": int(float(f.get("5. volume", 0) or 0)),
                            "NOTE": None,
                        }
                    )
                except Exception:
                    # skip malformed row
                    pass

        # simple sanity filter
        return [r for r in rows if r["PRICE_DATE"] and r["CLOSE"] and r["CLOSE"] >
            0]

    @task
    def load_full_refresh(rows: list[dict]) -> str:
        """
        Full refresh into RAW.STOCK_PRICES_AV:
          - CREATE SCHEMA/TABLE IF NOT EXISTS
          - TRUNCATE
          - INSERT (executemany)
          - COMMIT / ROLLBACK
        """
        schema = Variable.get("SNOWFLAKE_SCHEMA", default_var="RAW")
        table  = Variable.get("SNOWFLAKE_TABLE",  default_var="STOCK_PRICES_AV")

        hook = SnowflakeHook(snowflake_conn_id=SNOWFLAKE_CONN_ID)
        conn = hook.get_conn()
        cur  = conn.cursor()

        try:
            cur.execute("BEGIN")
            cur.execute(f"CREATE SCHEMA IF NOT EXISTS {schema}")
            cur.execute(f"""
                CREATE TABLE IF NOT EXISTS {schema}.{table} (
                    SYMBOL     STRING,
                    PRICE_DATE DATE,
                    OPEN       FLOAT,
                    HIGH       FLOAT,
                    LOW        FLOAT,
                    CLOSE      FLOAT,
                    VOLUME     NUMBER,
                    NOTE       STRING
                )
            """)
            cur.execute(f"TRUNCATE TABLE {schema}.{table}")

            if rows:
                insert_sql = f"""
                    INSERT INTO {schema}.{table}
                    (SYMBOL, PRICE_DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, NOTE)
                    VALUES (%(SYMBOL)s, %(PRICE_DATE)s, %(OPEN)s, %(HIGH)s, %(LOW)
                       s, %(CLOSE)s, %(VOLUME)s, %(NOTE)s)
                """
                cur.executemany(insert_sql, rows)

            cur.execute("COMMIT")
            return f"Loaded {len(rows)} rows into {schema}.{table}"
        except Exception:
            cur.execute("ROLLBACK")
            raise
        finally:
            cur.close()
            conn.close()

    # wiring
    load_full_refresh(extract())
```

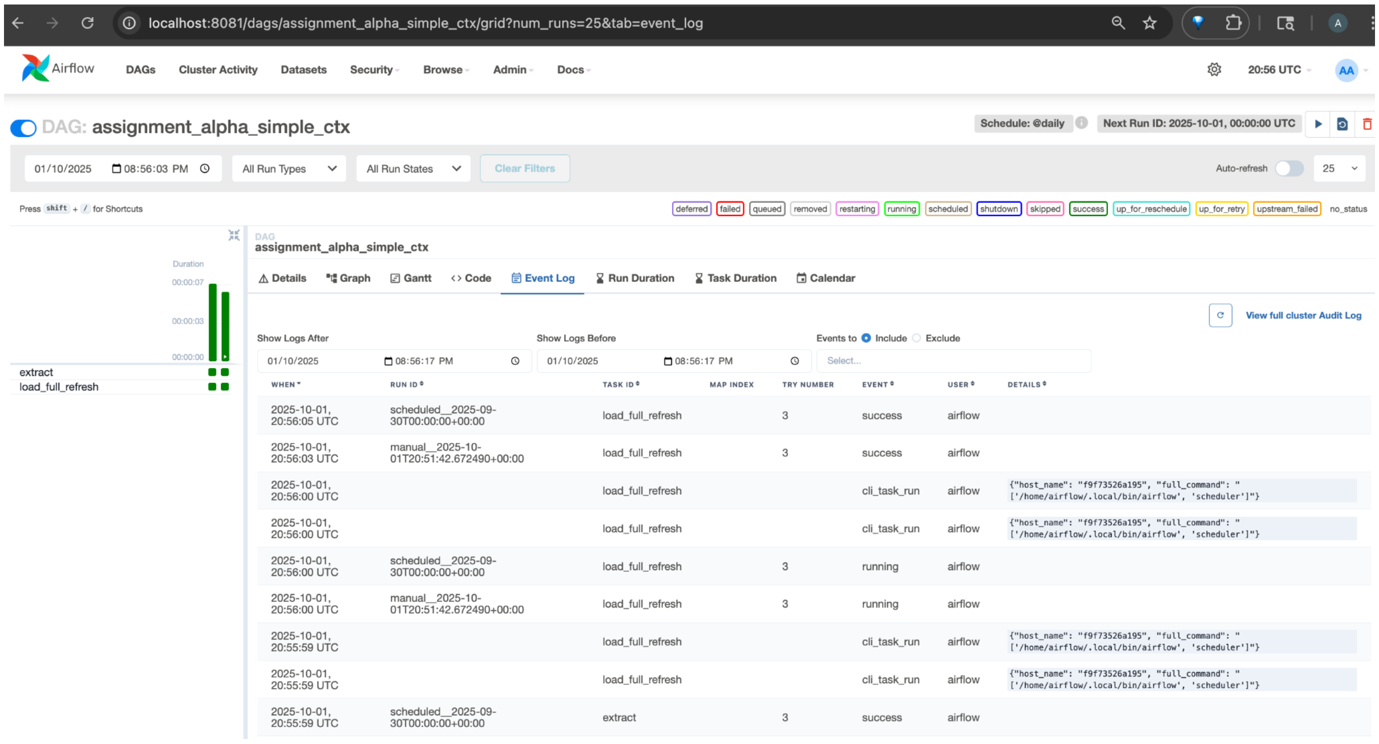
### B. SQL Transaction for Full Refresh (If Implemented in Python Task)

If the DAG uses Python to run SQL, ensure `BEGIN` / `COMMIT` / `ROLLBACK` are used appropriately, and MERGE/DELETE+INSERT are atomic.

```
    def load_full_refresh(rows: list[dict]) -> str:
        """
        Full refresh into RAW.STOCK_PRICES_AV:
          - CREATE SCHEMA/TABLE IF NOT EXISTS
          - TRUNCATE
          - INSERT (executemany)
          - COMMIT / ROLLBACK
        """
        schema = Variable.get("SNOWFLAKE_SCHEMA", default_var="RAW")
        table  = Variable.get("SNOWFLAKE_TABLE",  default_var="STOCK_PRICES_AV")

        hook = SnowflakeHook(snowflake_conn_id=SNOWFLAKE_CONN_ID)
        conn = hook.get_conn()
        cur  = conn.cursor()

        try:
            cur.execute("BEGIN")
            cur.execute(f"CREATE SCHEMA IF NOT EXISTS {schema}")
            cur.execute(f"""
                CREATE TABLE IF NOT EXISTS {schema}.{table} (
                    SYMBOL     STRING,
                    PRICE_DATE DATE,
                    OPEN       FLOAT,
                    HIGH       FLOAT,
                    LOW        FLOAT,
                    CLOSE      FLOAT,
                    VOLUME     NUMBER,
                    NOTE       STRING
                )
            """)
            cur.execute(f"TRUNCATE TABLE {schema}.{table}")

            if rows:
                insert_sql = f"""
                    INSERT INTO {schema}.{table}
                    (SYMBOL, PRICE_DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, NOTE)
                    VALUES (%(SYMBOL)s, %(PRICE_DATE)s, %(OPEN)s, %(HIGH)s, %(LOW)s, %(CLOSE)s, %(VOLUME)s, %(NOTE)s)
                """
                cur.executemany(insert_sql, rows)

            cur.execute("COMMIT")
            return f"Loaded {len(rows)} rows into {schema}.{table}"
        except Exception:
            cur.execute("ROLLBACK")
            raise
        finally:
            cur.close()
            conn.close()

    # wiring
    load_full_refresh(extract())
```

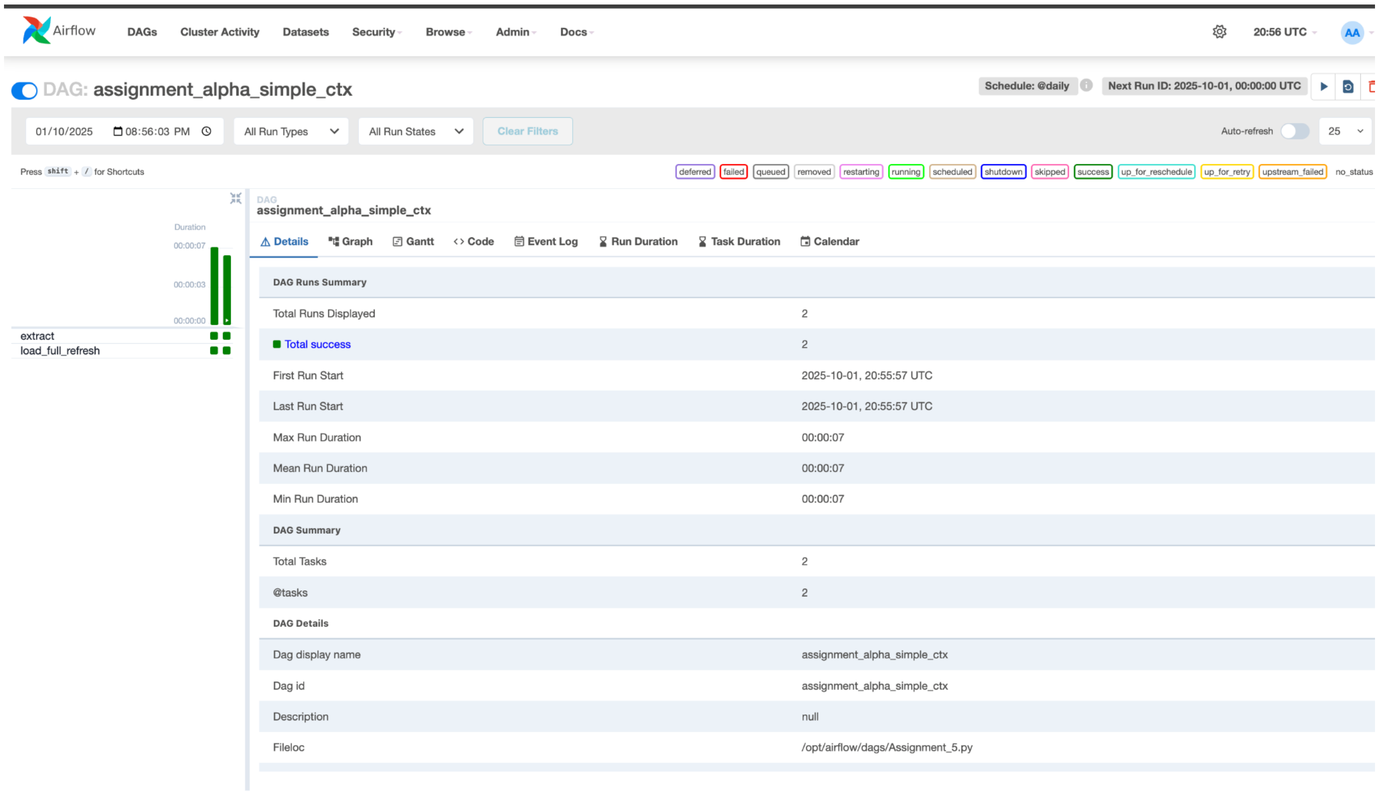
## VI. Execution Evidence: Event Logs

Figures 4 and 5 show successful runs of the two primary tasks (`extract` and `load_full_refresh`).



| WHEN | RUN ID | TASK ID | MAP INDEX | TRY NUMBER | EVENT | USER | DETAILS |
|---|---|---|---|---|---|---|---|
| 2025-10-01, 20:56:05 UTC | scheduled__2025-09-30T00:00:00+00:00 | load_full_refresh | | 3 | success | airflow | |
| 2025-10-01, 20:56:03 UTC | manual__2025-10-01T20:51:42.672490+00:00 | load_full_refresh | | 3 | success | airflow | |
| 2025-10-01, 20:56:00 UTC | | load_full_refresh | | | cli_task_run | airflow | {"host_name": "f9f73526a195", "full_command": "['/home/airflow/.local/bin/airflow', 'scheduler']"} |
| 2025-10-01, 20:56:00 UTC | | load_full_refresh | | | cli_task_run | airflow | {"host_name": "f9f73526a195", "full_command": "['/home/airflow/.local/bin/airflow', 'scheduler']"} |
| 2025-10-01, 20:56:00 UTC | scheduled__2025-09-30T00:00:00+00:00 | load_full_refresh | | 3 | running | airflow | |
| 2025-10-01, 20:56:00 UTC | manual__2025-10-01T20:51:42.672490+00:00 | load_full_refresh | | 3 | running | airflow | |
| 2025-10-01, 20:55:59 UTC | | load_full_refresh | | | cli_task_run | airflow | {"host_name": "f9f73526a195", "full_command": "['/home/airflow/.local/bin/airflow', 'scheduler']"} |
| 2025-10-01, 20:55:59 UTC | | load_full_refresh | | | cli_task_run | airflow | {"host_name": "f9f73526a195", "full_command": "['/home/airflow/.local/bin/airflow', 'scheduler']"} |
| 2025-10-01, 20:55:59 UTC | scheduled__2025-09-30T00:00:00+00:00 | extract | | 3 | success | airflow | |

Fig. 4: Event log view with task successes.



| | |
|---|---|
| **DAG Runs Summary** | |
| Total Runs Displayed | 2 |
| 🟩 Total success | 2 |
| First Run Start | 2025-10-01, 20:55:57 UTC |
| Last Run Start | 2025-10-01, 20:55:57 UTC |
| Max Run Duration | 00:00:07 |
| Mean Run Duration | 00:00:07 |
| Min Run Duration | 00:00:07 |
| **DAG Summary** | |
| Total Tasks | 2 |
| @tasks | 2 |
| **DAG Details** | |
| Dag display name | assignment_alpha_simple_ctx |
| Dag id | assignment_alpha_simple_ctx |
| Description | null |
| Fileloc | /opt/airflow/dags/Assignment_5.py |

Fig. 5: DAG details summary confirming successful runs.

## VII. Conclusion

This report demonstrates a working Airflow setup with Snowflake integration, proper configuration via Connections/Variables, transactional loading, and verified successful DAG runs.

## APPENDIX (OPTIONAL): FULL CODE

```python
# dags/assignment_alpha_simple_ctx.py
from airflow import DAG
from airflow.decorators import task
from airflow.models import Variable
from airflow.providers.snowflake.hooks.snowflake import SnowflakeHook

from datetime import datetime
import requests

# Read the Snowflake connection id from a Variable
SNOWFLAKE_CONN_ID = Variable.get("SNOWFLAKE_CONN_ID", default_var="
    snowflake_catfish")

with DAG(
    dag_id="assignment_alpha_simple_ctx",
    start_date=datetime(2025, 9, 1),
    schedule="@daily",
    catchup=False,
    tags=["assignment", "alpha", "snowflake"],
) as dag:

    @task
    def extract() -> list[dict]:
        """Pull compact daily OHLCV from Alpha Vantage for a small symbol list."""
        api_key = Variable.get("ALPHAVANTAGE_API_KEY")  # must exist
        symbols_csv = Variable.get("ALPHAVANTAGE_SYMBOLS", default_var="AAPL,MSFT"
            )
        symbols = [s.strip().upper() for s in symbols_csv.split(",") if s.strip()]

        base = "https://www.alphavantage.co/query"
        rows: list[dict] = []

        for sym in symbols:
            r = requests.get(
                base,
                params={
                    "function": "TIME_SERIES_DAILY",
                    "symbol": sym,
                    "outputsize": "compact",
                    "datatype": "json",
                    "apikey": api_key,
                },
                timeout=30,
            )
            r.raise_for_status()
            data = r.json()
            ts = data.get("Time Series (Daily)")
            if not ts:
                continue
            for ds, f in ts.items():
                try:
                    rows.append(
                        {
                            "SYMBOL": sym,
                            "PRICE_DATE": ds,
                            "OPEN": float(f.get("1. open", 0) or 0),
                            "HIGH": float(f.get("2. high", 0) or 0),
                            "LOW":  float(f.get("3. low",  0) or 0),
                            "CLOSE": float(f.get("4. close", 0) or 0),
                            "VOLUME": int(float(f.get("5. volume", 0) or 0)),
                            "NOTE": None,
                        }
                    )
                except Exception:
                    # skip malformed row
                    pass

        # simple sanity filter
        return [r for r in rows if r["PRICE_DATE"] and r["CLOSE"] and r["CLOSE"] >
            0]

    @task
    def load_full_refresh(rows: list[dict]) -> str:
        """
        Full refresh into RAW.STOCK_PRICES_AV:
          - CREATE SCHEMA/TABLE IF NOT EXISTS
          - TRUNCATE
          - INSERT (executemany)
          - COMMIT / ROLLBACK
        """
        schema = Variable.get("SNOWFLAKE_SCHEMA", default_var="RAW")
        table  = Variable.get("SNOWFLAKE_TABLE",  default_var="STOCK_PRICES_AV")

        hook = SnowflakeHook(snowflake_conn_id=SNOWFLAKE_CONN_ID)
        conn = hook.get_conn()
        cur  = conn.cursor()

        try:
            cur.execute("BEGIN")
            cur.execute(f"CREATE SCHEMA IF NOT EXISTS {schema}")
            cur.execute(f"""
                CREATE TABLE IF NOT EXISTS {schema}.{table} (
                    SYMBOL      STRING,
                    PRICE_DATE DATE,
                    OPEN        FLOAT,
                    HIGH        FLOAT,
                    LOW         FLOAT,
                    CLOSE       FLOAT,
                    VOLUME      NUMBER,
                    NOTE        STRING
                )
            """)
            cur.execute(f"TRUNCATE TABLE {schema}.{table}")

            if rows:
                insert_sql = f"""
                    INSERT INTO {schema}.{table}
                    (SYMBOL, PRICE_DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, NOTE)
                    VALUES (%(SYMBOL)s, %(PRICE_DATE)s, %(OPEN)s, %(HIGH)s, %(LOW)
                        s, %(CLOSE)s, %(VOLUME)s, %(NOTE)s)
                """
                cur.executemany(insert_sql, rows)

            cur.execute("COMMIT")
            return f"Loaded {len(rows)} rows into {schema}.{table}"
        except Exception:
            cur.execute("ROLLBACK")
            raise
        finally:
            cur.close()
            conn.close()

    # wiring
    load_full_refresh(extract())
```