Step By Step Apache Airflow

Comprehensive Guide to code in Spark and Python to implement ETL pipeline.

By Dhandapani Yedappalli Krishnamurthi Sep 8, 2025

Below is a **practical, copy-pasteable, step-by-step procedure** to build an end-to-end Airflow + PySpark ETL pipeline on your machine (you said Airflow & other software are already installed). It uses a *real-world style* dataset (NYC taxi style: trips with times, distances, fares) and shows:

- where files go,
- the PySpark ETL application,
- the Airflow DAG that orchestrates download → transform → validate → load,
- how to configure Airflow/Spark,
- how to test locally, and
- production best practices.

I'll keep steps linear so you can implement them now.

0 — Assumptions (what I assume is already true)

- You have **Airflow installed** and **runnable** (webserver + scheduler + worker or the single-node dev compose).

- You have Spark installed and `spark-submit` is available to the process that will run the Spark job (or you will configure `spark_home` in the Airflow connection).
- You can place files under your Airflow `dags/` folder (or mount them into the worker).
- You have network access to place dataset files (or will manually copy them to `data/raw/`).

1 — Project layout (create these folders/files)

```
airflow-pyspark/
├─ dags/
│   ├─ pyspark_etl_dag.py
│   └─ scripts/
│       └─ transform.py
├─ data/
│   └─ raw/
│       └─ nyc_trips_sample.csv    # your input CSV
(or download from public source)
└─ outputs/
    └─ processed/                   # Spark will
write Parquet here
```

Place the `airflow-pyspark` folder where your Airflow can access it (Airflow's `dags_folder` should contain `pyspark_etl_dag.py` and the `scripts/transform.py` must be visible to the worker).

---

2 — Pick / prepare a real-world dataset

Example schema (NYC taxi style):

- `trip_id`, `pickup_datetime` (`yyyy-MM-dd HH:mm:ss`), `dropoff_datetime`, `passenger_count`, `trip_distance`, `fare_amount`, `tip_amount`, `total_amount`, `payment_type`, `pickup_borough`, `dropoff_borough`

If you already have a dataset, copy it as `data/raw/nyc_trips_sample.csv`. If not, download any public CSV to that path (one file is fine for testing).

---

3 — PySpark ETL application (`dags/scripts/transform.py`)

This script:

- reads CSV,
- cleans types and filters anomalies,
- enriches with `trip_duration_minutes` and `fare_per_km`,
- writes Parquet partitioned by `year` and `month`.

Create `dags/scripts/transform.py`:

```
# dags/scripts/transform.py
import arg parse
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col,
to_timestamp, unix_timestamp, round as
spark_round, year, month

def build_spark():
    return SparkSession.builder \
        .appName("nyc_trips_etl") \
        .getOrCreate()

def transform(spark, input_path, output_path,
min_distance=0.1, max_fare=2000):
    df = (spark.read
          .option("header", "true")
          .option("inferSchema", "true")
          .csv(input_path))

    # Parse datetimes and compute duration
    df2 = (df
        .withColumn("pickup_ts",
to_timestamp(col("pickup_datetime"), "yyyy-MM-dd
HH:mm:ss"))
        .withColumn("dropoff_ts",
to_timestamp(col("dropoff_datetime"), "yyyy-MM-dd
HH:mm:ss"))
```

```
        .withColumn("trip_seconds",
unix_timestamp(col("dropoff_ts")) -
unix_timestamp(col("pickup_ts")))
        .withColumn("trip_duration_min",
(col("trip_seconds")/60).cast("double"))
        .withColumn("fare_amount",
col("fare_amount").cast("double"))
        .withColumn("trip_distance",
col("trip_distance").cast("double"))
    )

    # Basic data quality filters
    df3 = (df2

.filter(col("trip_distance").isNotNull() &
(col("trip_distance") >= float(min_distance)))

.filter(col("trip_duration_min").isNotNull() &
(col("trip_duration_min") > 0))
           .filter(col("fare_amount").isNotNull()
& (col("fare_amount") >= 0) & (col("fare_amount")
< float(max_fare)))
    )

    # Add derived metrics
```

```
    df4 = (df3.withColumn("fare_per_km",
spark_round(col("fare_amount") /
(col("trip_distance") + 1e-6), 2))
               .withColumn("year",
year(col("pickup_ts")))
               .withColumn("month",
month(col("pickup_ts"))))

    # Write partitioned parquet
    (df4.write
        .mode("overwrite")
        .partitionBy("year", "month")
        .parquet(output_path)
    )
    row_count = df4.count()
    print(f"ETL finished: wrote {row_count} rows
to {output_path}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--input", required=True)
    parser.add_argument("--output",
required=True)
    parser.add_argument("--min_distance",
required=False, default=0.1)
```

```
    parser.add_argument("--max_fare",
required=False, default=2000)
    args = parser.parse_args()

    spark = build_spark()
    transform(spark, args.input, args.output,
args.min_distance, args.max_fare)
    spark.stop()
```

Notes:

- Accepts CLI args so Airflow can pass parameters.
- Uses `mode("overwrite")` — see idempotency notes later.

---

4 — Airflow DAG to orchestrate the ETL (`dags/pyspark_etl_dag.py`)

This DAG:

1. optional `download` task to fetch CSV,
2. `pre_check` verifying input exists,
3. `spark_submit` runs `transform.py` with `SparkSubmitOperator`,
4. `validate` ensures Parquet output exists,
5. `load_to_db` (optional) writes aggregated results to Postgres using Spark JDBC.

Create `dags/pyspark_etl_dag.py`:

```
# dags/pyspark_etl_dag.py
from datetime import datetime, timedelta
import os
from airflow import DAG
from airflow.operators.python import
PythonOperator
from airflow.operators.empty import EmptyOperator
from
airflow.providers.apache.spark.operators.spark_su
bmit import SparkSubmitOperator
from airflow.models import Variable

DEFAULT_ARGS = {
    "owner": "you",
    "depends_on_past": False,
    "email_on_failure": False,
    "retries": 1,
    "retry_delay": timedelta(minutes=5),
}

DAG_ID = "nyc_trips_pyspark_etl"

DATA_DIR = "/opt/airflow/data"
```

```
RAW_INPUT = os.path.join(DATA_DIR, "raw",
"nyc_trips_sample.csv")
OUTPUT_DIR = os.path.join(DATA_DIR, "processed",
"nyc_trips_parquet")

def download_csv(**context):
    # OPTIONAL: implement if you want Airflow to
download dataset
    # Example: use urllib to download a public
CSV to RAW_INPUT
    import urllib.request
    url = Variable.get("nyc_trips_csv_url",
default_var=None)
    if not url:
        # no URL provided; assume file already
present
        return
    os.makedirs(os.path.dirname(RAW_INPUT),
exist_ok=True)
    urllib.request.urlretrieve(url, RAW_INPUT)
    print(f"Downloaded {url} to {RAW_INPUT}")

def pre_check(**context):
    if not os.path.exists(RAW_INPUT):
        raise FileNotFoundError(f"Input file not
found: {RAW_INPUT}")
```

```
    print("Input found")

def post_validate(**context):
    # very basic: check output directory exists
and non-empty
    if not os.path.exists(OUTPUT_DIR):
        raise FileNotFoundError(f"Output not
found: {OUTPUT_DIR}")
    # check at least one parquet file
    found = False
    for root, _, files in os.walk(OUTPUT_DIR):
        for f in files:
            if f.endswith(".parquet") or
f.endswith(".snappy.parquet"):
                found = True
                break
        if found:
            break
    if not found:
        raise FileNotFoundError("No parquet files
found in output")
    print("Post validate OK")

def load_to_postgres(**context):
    # OPTIONAL: load aggregated results to
Postgres via Spark JDBC
```

```
    # Provide connection details via Airflow
Variables or Connections
    jdbc_url = Variable.get("pg_jdbc_url",
default_var=None)
    pg_table = Variable.get("pg_table",
default_var="public.nyc_trips_agg")
    if not jdbc_url:
        print("No pg_jdbc_url Variable set -
skipping load_to_postgres")
        return

    from pyspark.sql import SparkSession
    spark =
SparkSession.builder.appName("load_to_postgres").
getOrCreate()

    # read parquet output and compute a small
aggregation (example)
    df = spark.read.parquet(OUTPUT_DIR)
    agg = (df.groupBy("year", "month",
"pickup_borough")
             .agg({"trip_duration_min":"avg",
"fare_amount":"sum", "trip_id":"count"}))

    (agg.write
        .format("jdbc")
```

```
        .option("url", jdbc_url)
        .option("dbtable", pg_table)
        .option("user", Variable.get("pg_user",
""))
        .option("password",
Variable.get("pg_password", ""))
        .mode("append")
        .save()
    )
    spark.stop()

with DAG(dag_id=DAG_ID,
         default_args=DEFAULT_ARGS,
         schedule_interval=None,
         start_date=datetime(2025, 1, 1),
         catchup=False) as dag:

    start = EmptyOperator(task_id="start")

    download =
PythonOperator(task_id="download_csv",
python_callable=download_csv,
provide_context=True)

    pre = PythonOperator(task_id="pre_check",
python_callable=pre_check)
```

```
    spark_submit = SparkSubmitOperator(
        task_id="spark_transform",

application="/opt/airflow/dags/scripts/transform.
py",   # must be reachable in worker
        conn_id="spark_default",
        application_args=["--input", RAW_INPUT,
"--output", OUTPUT_DIR],
        conf={"spark.executor.memory": "2g",
"spark.driver.memory": "1g"},
        verbose=True,
    )

    validate =
PythonOperator(task_id="post_validate",
python_callable=post_validate)

    load_db =
PythonOperator(task_id="load_to_postgres",
python_callable=load_to_postgres)

    end = EmptyOperator(task_id="end")

    start >> download >> pre >> spark_submit >>
validate >> load_db >> end
```

Important:

- `application` path must be visible to the process executing the SparkSubmitOperator (usually the Airflow worker).
- `conn_id="spark_default"` uses the Airflow Spark connection — configure it below.

---

5 — Configure `spark_default` connection in Airflow UI

Open Airflow UI → Admin → Connections → Create/Edit `spark_default`.

Recommended fields (example):

- Conn Id: `spark_default`
- Conn Type: `Spark`
- Host: `local` or `spark://spark-master:7077` (if standalone master) or `yarn`
- Extra (JSON) example:

```
{
  "spark_home": "/opt/spark",
  "spark_binary": "spark-submit",
  "deploy_mode": "client"
}
```

If you run `spark-submit` in PATH then Host/Extras are less critical. If you submit to YARN/EMR or k8s, set `host` and `extra` accordingly.

---

6 — Make Spark driver & JDBC dependencies available

- If you plan to use Spark JDBC (e.g., Postgres), place the JDBC jar (`postgresql-<version>.jar`) in `SPARK_HOME/jars/` or pass it with `--jars` in `SparkSubmitOperator` arguments. Example:

```
spark_submit = SparkSubmitOperator(
    ...,
    jars="/opt/spark/jars/postgresql-42.5.0.jar",
    ...
)
```

- For extra Python libs used by the Spark program, use `--py-files` or package them into a zip and pass via SparkSubmitOperator's `py_files` or `files` args.

---

7 — Test the PySpark script locally (before running Airflow)

From a shell that has `spark-submit`:

```
spark-submit \
  --master local[*] \
  dags/scripts/transform.py \
  --input data/raw/nyc_trips_sample.csv \
  --output outputs/processed/nyc_trips_parquet
```

Confirm `outputs/processed/nyc_trips_parquet` contains partitioned Parquet.

---

8 — Test tasks in Airflow

If you use the CLI (local installation):

```
# list DAGs
airflow dags list

# test a single task (runs task in process,
useful for debugging)
airflow tasks test nyc_trips_pyspark_etl
pre_check 2025-09-08

# trigger the DAG
airflow dags trigger nyc_trips_pyspark_etl
```

If using Docker Compose: run the equivalent `docker compose exec <worker> airflow ...` commands targeting your worker container.

9 — Unit tests for the PySpark script (optional, recommended)

Create `tests/test_transform.py` (pytest). This uses a local SparkSession to run a small CSV and verify output. Example:

```
# tests/test_transform.py
import shutil
from pyspark.sql import SparkSession
from dags.scripts.transform import transform

def test_transform(tmp_path):
    spark =
SparkSession.builder.master("local[2]").appName("
test").getOrCreate()

    # prepare sample CSV
    csv = tmp_path / "input.csv"

csv.write_text("trip_id,pickup_datetime,dropoff_d
atetime,trip_distance,fare_amount\n" +
                   "1,2025-01-01
10:00:00,2025-01-01 10:10:00,2.5,10.5\n")
```

```
    out = str(tmp_path / "out")
    transform(spark, str(csv), out)
    # assert some parquet files exist
    files = list((tmp_path /
"out").rglob("*.parquet"))
    assert len(files) > 0
    shutil.rmtree(out, ignore_errors=True)
    spark.stop()
```

Run with:

```
pytest -q
```

---

10 — Idempotency, data partitioning & atomic writes (best practices)

- **Idempotency**: avoid corrupting previous runs. Use:
  - write to a temporary path then move/rename atomically, OR
  - write using `mode("overwrite")` but be careful with concurrent runs (prefer per-partition overwrite).
- **Partitioning**: write Parquet partitioned by `year/month` (as in script) for faster reads/aggregations.
- **Large files**: coalesce partitions for small datasets before writing to avoid many tiny files (`df.coalesce(10)`).

- **Atomicity**: spark writes create `_temporary` folders — wait for successful write then rename.

---

11 — Monitoring, logging & retries

- Airflow task logs capture the `spark-submit` stdout/stderr — use them for debugging driver errors.
- Configure retries & `retry_delay` in DAG default_args.
- Use SLAs, sensors or separate monitoring tasks for downstream checks.
- Avoid running heavy Spark drivers inside the scheduler; run on worker/container/remote cluster.

---

12 — Data quality & validation examples

Add small PythonOperator tasks to:

- check row counts (compare source vs output),
- check null ratios for important columns,
- run range checks (e.g., `trip_distance` within expected range),
- write results to a monitoring table.

Example of a quick row-count check (PythonOperator):

```
def check_counts(**ctx):
    from pyspark.sql import SparkSession
```

```
    spark =
SparkSession.builder.appName("dq").getOrCreate()
    src =
spark.read.option("header","true").csv(RAW_INPUT)
    out = spark.read.parquet(OUTPUT_DIR)
    if src.count() == 0:
        raise ValueError("source 0 rows")
    if out.count() == 0:
        raise ValueError("output 0 rows")
    spark.stop()
```

---

13 — Performance tuning notes

- Set executor / driver memory via `conf` in SparkSubmitOperator.
- Use appropriate number of executors and cores on a cluster.
- Push heavy aggregations to Spark (not Python operator).
- Avoid UDFs when possible — prefer built-in functions for speed.

---

14 — Production deployment tips

- Put your DAG files in a Git repo; use CI/CD to deploy to Airflow.
- Use Airflow Connections and Variables (not hardcoded passwords). Use Secrets backend if available.
- Keep jobs idempotent and add schema/versioning to your datasets.

- Consider using Delta Lake or Iceberg for ACID semantics and easier merges.
- For large-scale Spark run on YARN/EMR/Databricks/K8s and configure Airflow to submit to those clusters.

---

15 — Quick troubleshooting checklist

- `spark-submit` not found? Ensure `spark_home` is set in connection or spark is in PATH.
- App fails on missing jars (JDBC)? Add jar to `SPARK_HOME/jars` or pass via `jars` param.
- DAG shows "not synced"? Put DAG file under Airflow `dags_folder` and check scheduler logs.
- `FileNotFoundError` for input? Ensure path is visible inside the container running the task (container mount).

Wrap-up & next steps

You now have a **complete pipeline** blueprint:

- `transform.py` — PySpark ETL app (read CSV → clean → write Parquet),
- `pyspark_etl_dag.py` — Airflow DAG using `SparkSubmitOperator` plus DQ and optional DB load,
- tests & run/test commands.