Name: Sebastian James Sampao Section: 3DSA

*Formative Lab Exercise #5:*

Workflow Orchestration with Apache Airflow

**Learning Objectives**

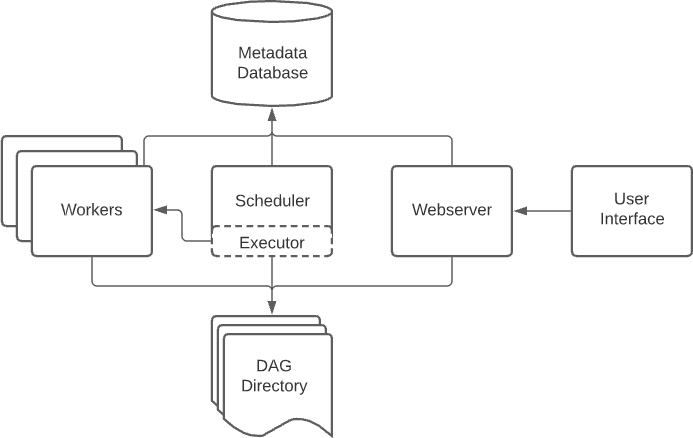
By the end of this lab, students will be able to:

* Set up Apache Airflow in a containerized environment.
* Define a Directed Acyclic Graph (DAG) using Python.
* Configure task operators and dependencies.
* Trigger DAG runs and monitor task execution.
* Apply orchestration concepts to real-world ingestion pipelines.

# Prerequisites

* Installed: Linux-like CLI (WSL), Docker, Docker Compose, pgcli, pgAdmin, Python, IDE
* Knowledge of: Python, Pandas, SQL basics, basic bash scripting, DAG concepts (from lecture)

# Airflow Overview



Reference: <https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html>

## Airflow Architecture

* **Web server:** GUI to inspect, trigger and debug the behaviour of DAGs and tasks. Available at http://localhost:8080.
* **Scheduler:** Responsible for scheduling jobs. Handles both triggering & scheduled workflows, submits Tasks to the executor to run, monitors all tasks and DAGs, and then triggers the task instances once their dependencies are complete.
* **Worker:** This component executes the tasks given by the scheduler.
* **Metadata database (postgres):** Backend to the Airflow environment. Used by the scheduler, executor and webserver to store state.

##### Other components (seen in docker-compose services):

* + **redis:** Message broker that forwards messages from scheduler to worker.
  + **flower:** The flower app for monitoring the environment. It is available at http://localhost:5555.
  + **airflow-init:** initialization service (customized as per this design)

All these services allow you to run Airflow with CeleryExecutor. For more information, see [Architecture Overview](https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html).

## Project Structure

* **./dags** - DAG\_FOLDER for DAG files
* **./logs** - contains logs from task execution and scheduler
* **./plugins** - for custom plugins

## Workflow Components

* **DAG:** Directed acyclic graph, specifies the dependencies between a set of tasks with explicit execution order, and has a beginning as well as an end. (Hence, “acyclic”)
  + **DAG Structure:** DAG Definition, Tasks (Operators), Task Dependencies (control flow: >> or << )
* **Task:** a defined unit of work (aka, operators in Airflow). The Tasks themselves describe what to do, be it fetching data, running analysis, triggering other systems, or more.
  + Common Types: Operators, Sensors, TaskFlow decorators
  + Sub-classes of Airflow's BaseOperator
* **DAG Run:** an individual execution/run of a DAG
  + scheduled or triggered
* **Task Instance:** an individual run of a single task. Task instances also have an indicative state, which could be “running”, “success”, “failed”, “skipped”, “up for retry”, etc.
  + Ideally, a task should flow from none, to scheduled, to queued, to running, and to success.

## References

* <https://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html>
* <https://airflow.apache.org/docs/apache-airflow/stable/concepts/tasks.html>

# Lab Part 0: Getting Started with Airflow

### Task 0.1: Launch the Airflow stack with Docker Compose

Ensure the following project structure exists:

***Directory***

project-root/

├── dags/

│ └── (your DAGs will go here)

├── data/

├── docker-compose.yml

├── Dockerfile

├── requirements.txt

# ← Folder to store DAG definitions

# ← Folder to hold downloaded .parquet files # ← Provided in Appendix 1

# ← Provided in Appendix 2

# ← Provided in Appendix 3

Use the provided files to create docker images on your working directory:

* docker-compose.yaml (Appendix 1)
* Dockerfile (Appendix 2)
* requirements.txt (Appendix 3)

After ensuring these are present in your pwd, in your terminal, run:

#### Step 1: Build the Docker image

***Bash***

docker-compose build

#### Step 2: Initialize the database and create admin user

***Bash***

docker-compose up airflow-init

#### Step 3: Run the webserver and scheduler

***Bash***

docker-compose up

***NOTE:*** Wait for the Airflow UI to be accessible at http://localhost:8080

#### Step 4: Access the Airflow UI

***Web Browser***

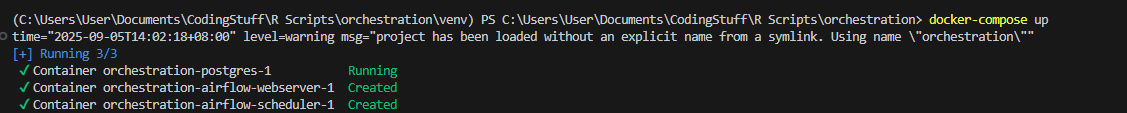
Access: http://localhost:8080 Username: admin

Password: admin

📸 Screenshot Submission 0.1.1:

Show screen of Terminal with successful container launch and accessible Airflow UI in the browser.

***A screen shot of a computer

AI-generated content may be incorrect.***

📸 Screenshot Submission 0.1.2:

Show screen of Browser window showing the Airflow UI loaded at http://localhost:8080

***A screenshot of a computer

AI-generated content may be incorrect.***

🧠 Checkpoint Question Submission 0.1.1:E

1. What are the different Airflow services launched by the Docker Compose setup?
2. Why is airflow-init required before the main run?

✔ *Keep your answer short and clear, using your own words.*

1. Airflow Webservices and Airflow Scheduler
2. It initializes the user credentials for airflow, initializes as well the airflow metadata database that are necessary to run the different airflow services, and to allow the synchronization of the database schema across all airflow services to ensure that the same structure and schema is consistent.

# Lab Part 1: Writing Your First DAG on Airflow

In this part, you will create a simple DAG composed of sequential Bash tasks, load it into the Airflow UI, and verify it runs successfully alongside understanding the structure of a DAG and core Airflow concepts.

### Task 1.1: Create dag\_simple.py

Create a new file inside the dags/ directory named dag\_simple.py, and paste the following code:

***Python***

from airflow import DAG

from airflow.operators.bash import BashOperator from datetime import datetime

with DAG(dag\_id="dag\_simple",

start\_date=datetime(2025, 1, 1), schedule\_interval="@daily", catchup=False) as dag:

task\_1 = BashOperator( task\_id="print\_date", bash\_command="date"

)

task\_2 = BashOperator( task\_id="print\_hello", bash\_command="echo 'Hello from Airflow!'"

)

task\_1 >> task\_2 # Task dependency: task\_1 must finish before task\_2

#### Expected DAG Behavior

* Runs daily, starting from Jan 1, 2025
* prints the system date

**Task 1:**

**Task 2:**

* prints a hello message
* DAG name in UI: dag\_simple

#### Steps to Verify in Airflow UI:

1. Open Airflow at http://localhost:8080
2. Log in using the credentials set-out in the build
3. Locate dag\_simple in the list
4. Toggle it **ON** (enable)
5. Click the Play button ▶ (This triggers the DAG)
6. Navigate to Graph View or Tree View to monitor execution

📸 Screenshot Submission 1.1.1:

Screenshot showing the Airflow UI with your dag\_simple.py loaded and visible.

A screenshot of a computer

AI-generated content may be incorrect.

📸 Screenshot Submission 1.1.2:

Screenshot showing the Graph View with the two-task dependency (print\_date → print\_hello)

A screenshot of a computer

AI-generated content may be incorrect.

🧠 Checkpoint Question Submission 1.1.1:

1. What does the >> operator do between task\_1 and task\_2?
2. Why is catchup=False often used during development or testing?

✔ *Keep your answer short and clear, using your own words.*

1. task\_1 >> task\_2 means that task\_2 is dependent on task\_1 finishing and executing
2. Catchup = False is used in testing environments since it's not necessary to execute or generate historical data from start date up until the time execution since testing is more interested if all workers are running properly and if the orchestration of workers is fit and runs properly.

### Task 1.2: Create a 3rd task inside dag\_simple.py

Create a third task:

***Python***

task\_3 = BashOperator( task\_id="print\_goodbye",

bash\_command="echo 'Goodbye from Airflow!'"

)

task\_2 >> task\_3

Re-deploy the DAG and screenshot the updated DAG graph.

📸 Screenshot Submission 1.2.1:

Screenshot showing the Airflow UI with your updated dag\_simple.py loaded and visible.

A screenshot of a computer

AI-generated content may be incorrect.

📸 Screenshot Submission 1.2.2:

Screenshot showing the Graph View with the three-task dependency.

A screen shot of a computer

AI-generated content may be incorrect.

# Lab Part 2: Building an Ingestion DAG on Airflow

In this part, you will define a multi-task Airflow DAG for ingesting data using Bash and Python operators. Focus is on authoring correct task dependencies and verifying DAG loading in the UI.

### Task 2.0: Download New York City Taxi Data

Go to the following website: <https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>

Download the parquet data for the Yellow Taxi in January 2025 to your working directory. You may download manually or use the following bash command:

***Bash***

curl -O https://d37ci6vzurychx.cloudfront.net/trip-data/yellow\_tripdata\_2025-01.parquet

### Task 2.1: Create a new DAG dag\_ingest.py

Create a DAG that ingests a publicly available Yellow Taxi dataset by:

* Downloading a .parquet file from a remote [URL](https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page).
* Storing it in a shared Docker volume (/opt/airflow/data)
* Running a Python script to ingest the data into a Postgres table

Create a new file inside the dags/ directory named dag\_ingest.py, and paste the following code:

***Python***

from airflow import DAG

from airflow.operators.bash import BashOperator from airflow.operators.python import PythonOperator from datetime import datetime

import os

URL = "https://d37ci6vzurychx.cloudfront.net/trip-data/yellow\_tripdata\_2025-01.parquet" OUTPUT\_PATH = "/opt/airflow/data/yellow\_tripdata\_2025-01.parquet"

default\_args = { "owner": "airflow", "retries": 1,

}

def ingest\_to\_postgres(): import ingest\_script ingest\_script.main()

with DAG(

dag\_id="dag\_ingest", default\_args=default\_args, start\_date=datetime(2025, 1, 1), schedule\_interval=None, catchup=False,

description="Download and optionally ingest taxi data", tags=["ingestion", "parquet"]

) as dag:

download\_data = BashOperator(

task\_id="download\_data",

bash\_command=f"curl -sS -o {OUTPUT\_PATH} {URL}"

)

# Optional ingestion step ingest\_data = PythonOperator(

task\_id="ingest\_data\_to\_pg", python\_callable=ingest\_to\_postgres

)

download\_data >> ingest\_data

Be mindful of the following:

* Do not trigger the DAG yet. Save and verify it appears in the UI.
* Ensure /opt/airflow/data/ exists and is mounted via docker-compose.yml

***YAML***

volumes:

* ./dags:/opt/airflow/dags
* ./data:/opt/airflow/data
* Ensure ingest\_script.py (Appendix 4) must be inside the mounted /opt/airflow path (for example: ./dags/ingest\_script.py, copied into the container).

#### Step 1: Rebuild the container if you updated files:

***Bash***

docker-compose down docker-compose up --build

#### Step 2: Open Airflow UI

📸 Screenshot Submission 2.1.1:

Screenshot showing the Airflow UI with your dag\_ingest.py loaded and visible (status: “Off” is fine).

A screenshot of a computer

AI-generated content may be incorrect.

📸 Screenshot Submission 2.1.2:

Screenshot showing the Graph View of the DAG showing two correctly linked tasks.

A screenshot of a computer

AI-generated content may be incorrect.

🧠 Checkpoint Question Submission 2.1.1:

1. What would happen if you accidentally created a cycle in the task dependencies?
2. Why does Airflow enforce that DAGs must be acyclic?

✔ *Keep your answer short and clear, using your own words.*

1. Essentially it will be dependent on itself’s output, and creating a self-referencing issue, that will create a loop that runs essentially forever.
2. To ensure that there are no loops in the workflow and ensure that there are no errors that will be brought upon by this error.

Syntax Check

If the DAG does not appear in the UI, use the scheduler logs to diagnose:

***Bash***

docker-compose logs scheduler

Alternatively, you can run the following to check if your DAGs are correctly parsed and recognized by Airflow from inside the running container:

***Bash***

docker-compose exec airflow-webserver airflow dags list

# Lab Part 3: Executing and Monitoring Ingestion DAG on Airflow

In this part, you will trigger your previously authored DAG, observe task states, and inspect logs and dependency graphs using the Airflow UI. This introduces you to the monitoring, scheduling, and logging features of Airflow.

### Task 3.1: Trigger and Monitor the DAG

#### Step 1: Open the Airflow UI at:

***Web Browser***

Access: http://localhost:8080 Username: admin

Password: admin

#### Step 2: Locate your dag\_ingest DAG and turn it “ON” (toggle switch). Step 3: Trigger the DAG manually via the UI:

* Click the Play button ▶
* Select Trigger DAG w/ config

#### Step 4: Observe the following:

* DAG run status (for example: “queued”, “running”, “success”)
* Graph View and Tree View
* Individual task logs (click each task and view Logs tab)

📸 Screenshot Submission 3.1.1:

Screenshot showing the Airflow UI with your dag\_ingest DAG run with success state.

A screenshot of a computer

AI-generated content may be incorrect.

📸 Screenshot Submission 3.1.2:

Screenshot showing the Graph View of the DAG showing task dependencies and statuses.

A screenshot of a computer

AI-generated content may be incorrect.

📸 Screenshot Submission 3.1.3:

Screenshot showing the task log output from download\_data and ingest\_data steps.

🧠 Checkpoint Question Submission 3.1.1:

1. What is the difference between a DAG-level failure and a task-level failure?
2. What are some built-in retry mechanisms in Airflow that help handle failures?

✔ *Keep your answer short and clear, using your own words.*

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

1. A DAG-level failure occurs when the entire DAG run cannot complete successfully, often because one or more critical tasks fail without retries thus stopping the entire workflow. A task-level failure happens when an individual task run fails due to issues like invalid parameters, code errors, or resource problems. Unlike DAG-level failures, task failures can often be retried without stopping the entire DAG.
2. One would be setting the parameter default\_args when defining a DAG in python. It defines the amount of retries, retry delay, etc.

#### Tips & Debugging

|  |  |
| --- | --- |
| **Tip** | **Command** |
| List DAGs | docker-compose exec airflow-webserver airflow dags list |
| List DAG tasks | docker-compose exec airflow-webserver airflow tasks list dag\_ingest |
| View DAG run logs | Use the Logs tab in the UI or run: docker-compose logs airflow-scheduler |
| Reset DAG runs\* | docker-compose exec airflow-webserver airflow dags delete dag\_ingest |

*\*clears history*

#### Directory Structure Reminder

By this time, your directory structure should look like:

***Directory***

project-root/

├── dags/

# ← Folder to store DAG definitions

│

│

│

├── dag\_ingest.py

└── ingest\_script.py

└── dag\_simple.py

# ← Provided in Appendix 4

├── data/ # ← Folder to hold downloaded .parquet files

├── docker-compose.yml # ← Provided in Appendix 1

├── Dockerfile # ← Provided in Appendix 2

└── requirements.txt # ← Provided in Appendix 3

# Lab Part 4: DAG Parameterization on Airflow

In this part, you will modify your ingestion DAG so that it uses the Airflow execution date to dynamically name the downloaded file. You’ll gain experience using Jinja templating to make your workflows

date-aware, a foundational practice for production-grade pipelines.

### Task 4.1: Use {{ ds }} in Your DAG

#### Step 1: Modify your existing dag\_ingest.py:

* Open dag\_ingest.py and modify the bash\_command in download\_data task as follows:

***Python***

from airflow.utils.dates import days\_ago

with DAG(

dag\_id="dag\_ingest\_param", default\_args=default\_args,

start\_date=days\_ago(1), # Easier to test templating schedule\_interval="@daily",

catchup=True,

description="Ingestion DAG with dynamic date-based filenames", tags=["parameterized", "templating"]

) as dag:

download\_data = BashOperator( task\_id="download\_data", bash\_command=(

"curl -sS -o /opt/airflow/data/yellow\_tripdata\_{{ ds[:7] }}.parquet " "https://d37ci6vzurychx.cloudfront.net/trip-data/yellow\_tripdata\_{{ ds[:7] }}.parquet"

)

)

ingest\_data = PythonOperator( task\_id="ingest\_data\_to\_pg", python\_callable=ingest\_to\_postgres

)

download\_data >> ingest\_data

#### Step 2: Set catchup=True to generate historical runs with different dates.

Step 3: Trigger the DAG from a backfill execution date (for example: January 1, 2025):

* In Airflow UI: DAG → Trigger DAG → Pick Execution Date: 2025-01-01

#### Step 4: Navigate to ./data/ volume (via host or inside container) and confirm:

* yellow\_tripdata\_2025-01.parquet

📸 Screenshot Submission 4.1.1:

Screenshot showing the DAG Graph View showing a successful backfilled run.

A screenshot of a computer

AI-generated content may be incorrect.

📸 Screenshot Submission 4.1.2:

Screenshot showing the Terminal or container log showing the correctly named downloaded file:

/opt/airflow/data/yellow\_tripdata\_2025-01-05.parquet

🧠 Checkpoint Question Submission 4.1.1:

1. Why is it useful to make DAGs parameterized by date ({{ ds }})? Why is it ds[:7] in the script?
2. How does this help when ingesting data from time-partitioned sources?

✔ *Keep your answer short and clear, using your own words.*



1. The Jinja-based macro {{ ds }} is very useful when it comes to scheduling workflows as it tracks the execution date of that particular work and returns it as a string value. ds[:7] is obtained as it returns YYYY-MM

# Lab Part 5: Workflow Orchestration with Kestra

In this part, you are expected to duplicate the lab structure and learning outcomes from Airflow exercises and re-implement them using Kestra. This includes the same set of tasks, checkpoint questions, screenshots, and appendices. This time, all must be adapted to Kestra’s YAML-based syntax and interface.

**Deliverables**

Structure your submission using the following sections, exactly as you did for Airflow:

1. Kestra Overview
   1. Brief explanation of Kestra and its key features
   2. Differences you observe vs. Airflow
2. Lab Part 0: Getting Started with Kestra
   1. Setting up Kestra using Docker Compose
   2. Accessing the Kestra UI
   3. Show that Kestra is running on Docker and the UI is accessible.
   4. Checkpoint: What are the main services that Kestra launches when run with Docker Compose?
   5. Checkpoint: How does the Kestra UI help you understand what’s happening in the background?
3. Lab Part 1: Writing Your First DAG on Kestra
   1. Create a simple sequential DAG in YAML
   2. Screenshot: YAML file for your first DAG on Kestra
   3. Screenshot: Show your simple Kestra flow loaded and a successful run.
   4. Checkpoint: In your YAML, what structure is used to define a sequence of tasks?
   5. Checkpoint: Compared to Airflow’s BashOperator, how does Kestra’s equivalent task behave?
4. Lab Part 2: Building an Ingestion DAG on Kestra
   1. Use a download task to fetch NYC Yellow Taxi data
   2. Screenshot: YAML file for your Ingestion DAG
   3. Screenshot: Show your ingestion DAG and log output of data download.
   4. Checkpoint: Which Kestra task did you use to download the taxi dataset?
   5. Checkpoint: How did you make sure the file was saved to the correct location?
5. Lab Part 3: Executing and Monitoring DAG on Kestra
   1. Trigger your flow from the UI
   2. Explore Graph View or Run History
   3. Screenshot: Show the execution status and the run view (graph view).
   4. Screenshot: Show the execution status and the run view (timeline view).
   5. Checkpoint: What’s the difference between a failed flow run and a failed task?
   6. Checkpoint: What features does Kestra provide to help you debug failed flows?
6. Lab Part 4: DAG Parameterization on Kestra
   1. Use input variables (like run\_date) to make file paths dynamic
   2. Screenshot: Show your flow using a dynamic date-based filename and execution logs.
   3. Checkpoint: How did you inject parameters like the date into your filename or commands?
   4. Checkpoint: Why is it powerful to pass values into a DAG at runtime?
7. Appendices
   1. Include your customized docker-compose.yml, sample YAML flows, and any configs used

**Tips for Success**

* Refer to the [Kestra documentation](https://kestra.io/docs/) for YAML structure and task types
* Mirror your logic from Airflow flows as closely as possible
* Use the Echo, Download, and Sequential tasks as building blocks

# Appendix 1: docker-compose.yaml

***YAML***

services:

postgres:

image: postgres:13 environment:

POSTGRES\_USER: airflow POSTGRES\_PASSWORD: airflow POSTGRES\_DB: airflow

volumes:

- postgres-db-volume:/var/lib/postgresql/data ports:

- "5432:5432"

airflow-init:

build:

context: . depends\_on:

- postgres entrypoint: > bash -c "

airflow db init && airflow users create \

--username admin \

--firstname Air \

--lastname Flow \

--role Admin \

--email [admin@example.com](mailto:admin@example.com) \

--password admin

"

environment:

AIRFLOW CORE EXECUTOR: LocalExecutor

AIRFLOW CORE SQL\_ALCHEMY\_CONN: postgresql+psycopg2://airflow:airflow@postgres:5432/airflow volumes:

- ./dags:/opt/airflow/dags

airflow-webserver:

build:

context: . depends\_on:

- postgres environment:

AIRFLOW CORE EXECUTOR: LocalExecutor

AIRFLOW CORE SQL\_ALCHEMY\_CONN: postgresql+psycopg2://airflow:airflow@postgres:5432/airflow AIRFLOW WEBSERVER EXPOSE\_CONFIG: "True"

volumes:

- ./dags:/opt/airflow/dags ports:

- "8080:8080"

command: webserver

airflow-scheduler:

build:

context: . depends\_on:

* airflow-webserver environment:

AIRFLOW CORE EXECUTOR: LocalExecutor

AIRFLOW CORE SQL\_ALCHEMY\_CONN: postgresql+psycopg2://airflow:airflow@postgres:5432/airflow volumes:

* ./dags:/opt/airflow/dags command: scheduler

volumes:

postgres-db-volume:

# Appendix 2: Dockerfile

***Dockerfile***

FROM apache/airflow:2.7.3

# Set working directory

ENV AIRFLOW\_HOME=/opt/airflow

# Switch to airflow user USER airflow

# Copy and install extra Python dependencies COPY requirements.txt /requirements.txt

RUN pip install --no-cache-dir -r /requirements.txt

# Appendix 3: requirements.txt

***Text***

pyarrow

# Appendix 4: ingest\_script.py

Remember the Python script you wrote for Formative Lab Exercise #4 | Lab Part 4: Data Ingestion with Python Script? We will re-use that ingestion script for this exercise. Make sure to rename it accordingly to ingest\_script.py.