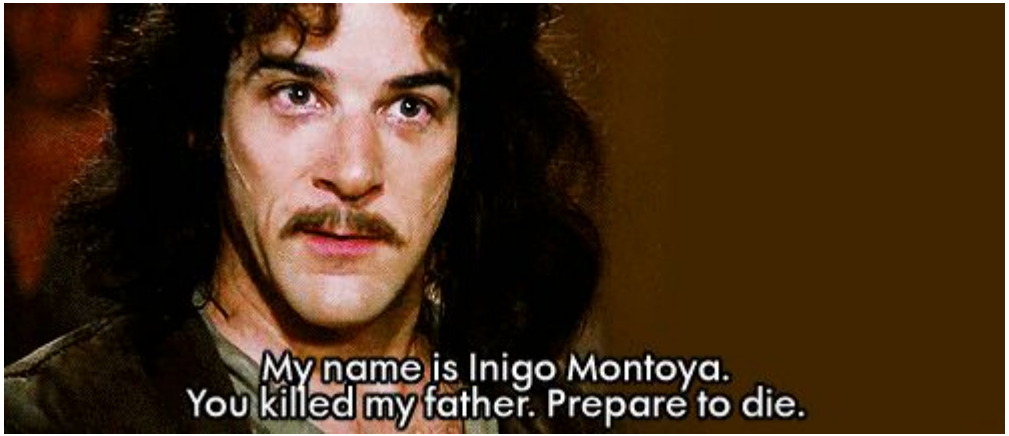
Airflow and DBT

Day one: Airflow

Introductions

- Polite Greeting
- Your Name
- Relevant Personal Link
- Clear Expectations



What is Airflow

Apache Airflow is an open-source workflow management platform for creation, scheduling, & management of data-engineering pipelines

- Implement programs from any language
- Workflows are written in Python
- Implements Workflows as Directed Acyclic Graphs (DAGs)
- Workflows are accessed via code, command line, or web-interface/REST API

What is dbt

Data Build Tool (dbt) is a development framework combining modular SQL with software engineering best-practices to enable reliable and fast data transformation

- Write custom business logic using SQL
- Build data pipelines
- Automate data quality testing
- Deliver data with documentation side-by-side with code

Day one: The plan

- 10.00 Airflow Basics:
  - Use case
  - Airflow in a nutshell
- 10.45 Short Break
- 11.00 Hands-on: Building a basic DAG
- 12.00 Lunch I’m sure
- 13.00 Airflow: Let's dive deeper
  - Internal architecture
  - XCOM
  - DAGs
  - Sensors
- 14.00 Short Break
- 14.15 Custom Operator: Python Code
- 16.00 Airflow Use Case: Centralized Pipelines
- 16.30 End

Use-case: Transparent pipelines with Airflow

We had a large, unorganized dataset of questionable quality. We were asked to fix the dataset to improve quality and enable the extraction of credible market insights for the company. We identified goals including:

- Extracting data from the central data team (AWS Redshift)
- Deduplication
- Cleaning
- Filling in missing data using business logic
- Filling in missing data or enhancing data using Machine Learning models
- Loading the final enhanced dataset into a PostgreSQL database for the webapp

How could we create a clear, transparent, and manageable pipeline? We chose Airflow

Use-case: Transparent pipelines with Airflow (2)

Airflow allowed us to:

- Build transparency into the workflow
- Optimize the development workflow
- Easily re-run failed Tasks
- Deliver insight into when Tasks ran, for how long, etc
- Integrate services as needed (Python, SQL, Redshift, Kubernetes, Lambdas)

And because Airflow is Python, it was fairly easy to get started on migrating our pipeline to a clean setup with CI/CD, including unit tests.

Airflow in a nutshell

Apache Airflow is an open-source workflow management platform for creation, scheduling, & management of data-engineering pipelines

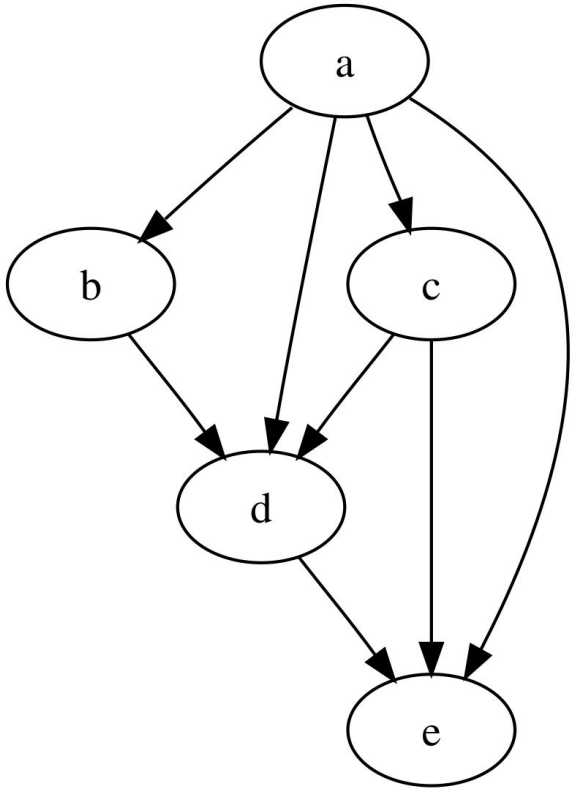
- DAGs
- Tasks
- UI features
- Jinja templating

What are DAGs

Directed Acyclic Graph (DAG)

A one-directional flow-chart of tasks

- Tasks
- Task dependencies
- Task details (Name, start date, owner, etc.)



What are Tasks?

Tasks are the basic executable unit in Airflow.

Tasks are organized into DAGs, with upstream and downstream dependencies (the flowchart / DAG) and are executable.

There are three types:

- **Operators:** pre-defined, reusable Tasks
- **Sensors**: a Task type that waits until a condition is met before executing
- **TaskFlow decorators**: custom python code (@task)

Operators: Pre-defined, reusable Tasks

- BashOperator: Executes a Bash script
- PythonOperator: Executes python code
- EmailOperator: Sends an email
- KubernetesPodOperator: Interacts with Kubernetes Pods

And many, many more. Some are also ecosystem dependent.

```
kubernetes_min_pod = GKEStartPodOperator(
    # The ID specified for the task.
    task_id="pod-ex-minimum",
    # Name of task you want to run, used to generate Pod ID.
    name="pod-ex-minimum",
    project_id=PROJECT_ID,
    location=CLUSTER_REGION,
    cluster_name=CLUSTER_NAME,
    # Entrypoint of the container, if not specified the Docker container's
    # entrypoint is used. The cmds parameter is templated.
    cmds=["echo"],
    # The namespace to run within Kubernetes, default namespace is
    # `default`.
    namespace="default",
    # Docker image specified. Defaults to hub.docker.com, but any fully
    # qualified URLs will point to a custom repository. Supports private
    # gcr.io images if the Composer Environment is under the same
    # project-id as the gcr.io images and the service account that Composer
    # uses has permission to access the Google Container Registry
```

```
email_notification = EmailOperator(
    task_id="email_notification",
    to="noreply@astronomer.io",
    subject="dag completed",
    html_content="the dag has finished",
)
```

Sensors: They sense things.

Sensors are Operators designed to execute when triggered by an event. Sensors can be time-based, wait for a file, or an external event, but they wait for a condition to be met and then execute so downstream Tasks can run.

Examples:

- **DayTimeSensor**: Waits for specific day and time
- **ExternalTaskSensor**: Triggers when another Task finishes
- **SQLSensor**: Waits for data to arrive

Sensors have one flaw, and we will discuss that this afternoon.

Taskflow decorators - Creating Tasks with API

TaskFlow decorators such as @task are programming patterns that add functionality to an object. In this case, they turn functions into tasks with an ID, facilitate passing data between Tasks, and define dependencies. Decorators create similar functionality to and are interoperable with classical Operators, including PythonOperator.

TaskFlow is newer and therefore preferable over building the same functionality in PythonOperator.

UI features

The Airflow UI offers a feature-rich interface for:

- Monitoring Workflows and Tasks
- Visualization for Workflows
- Visualizing connections
  - These are also a one-stop for auth information for DAGs and Tasks
- Inspect DAGs and code from the UI (easier debugging)
- One-stop for Gantt charts, logs, timing, re-running, etc.

Jinja Templating

Jinja is a lightweight templating engine for Python that embeds variables, logic, and control structures to dynamically generate text. It enables flexible configuration and parameterization, especially useful for defining dynamic content like SQL queries or configuration files.

Jinja improves DAGs by dynamically injecting task parameters, SQL code, and runtime values (like execution date) at runtime. This improves reusability, reduces hardcoding, and supports complex automation logic across time-based data pipelines.

Airflow use-cases: Why?

- **Schedule workflows**: Automates recurring tasks like daily data jobs or reports.

- **Visual DAGs**: Clear, graphical view of task dependencies and execution status.

- **Modular Python code**: Workflows are written in Python, making them readable and reusable. Also easier to use than Infrastructure as Code variants like AWS step functions.

- **Agnostic**: Airflow is an Apache tool and can be used locally or with any cloud platform.

- **Handles failures**: Built-in retry, alerting, and logging for robust pipelines.

- **Track and debug**: Easy to inspect past runs, logs, and task outputs.

- **Extensible**: Supports plugins, custom operators, and integrates with many tools.

- **Scalable**: Run small local jobs or scale out with distributed executors (Celery, Kubernetes).

- **Templating & parameters**: Dynamically generate SQL, API calls, or filenames with runtime context.

Short Break

Morning exercise

Let’s get started with Airflow! On the laptops, you’ll find that Docker, WSL and Airflow are installed. Our first order of business:

- Open Docker and run the container for Airflow.

This is the equivalent of a fresh install — the UI is there, but we haven’t created any DAGs yet.

Let’s create a DAG

main.md from airflow import DAG

```python
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.utils.dates import days_ago
from datetime import timedelta

with DAG(
    dag_id="simple_bash_graph",
    description="Demo DAG showing task dependencies with BashO
    schedule_interval=None,  # Manual trigger only
    start_date=days_ago(1),
    catchup=False,
    default_args={
        "retries": 0,
        "retry_delay": timedelta(minutes=1),
    },
    tags=["example"],
) as dag:

    bash1 = BashOperator(
        task_id="bash1",
```

```python
    bash1 = BashOperator(
        task_id="bash1",
        bash_command='echo "Running bash1 - {{ run_id }}"',
    )

    bash2 = BashOperator(
        task_id="bash2",
        bash_command='echo "Running bash2 - {{ run_id }}"',
    )

    bash3 = BashOperator(
        task_id="bash3",
        bash_command='echo "Running bash3 - {{ run_id }}"',
    )

    bash4 = BashOperator(
        task_id="bash4",
        bash_command='echo "Running bash4 - {{ run_id }}"',
    )

    bash5 = BashOperator(
        task_id="bash5",
        bash_command='echo "Running bash5 - {{ run_id }}"',
    )
```

Lunch

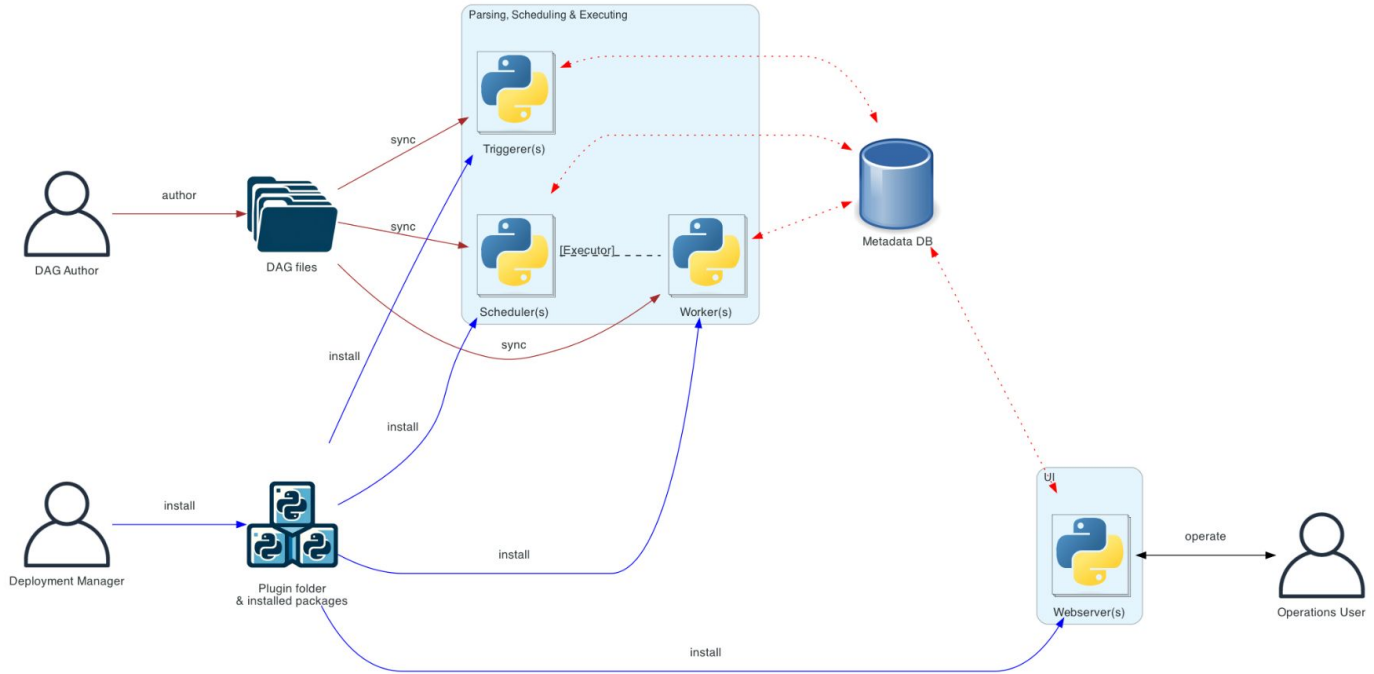
Airflow: Let's dive deeper

- Internal Architecture
- XCom
- Dynamic DAGs
- DAG Dependencies
- DAGs with logic

Airflow Architecture

- **Scheduler**: Triggers scheduled Workflows and submits tasks to the Executor.
- **Executor**: a configuration within the scheduler (not a separate component) determining how tasks are run (e.g., SequentialExecutor, LocalExecutor, CeleryExecutor, etc.).
- **DAG Processor**: Parses DAG files and serializes them into the metadata database.
- **Webserver**: Provides a UI to inspect, trigger, and debug DAGs and tasks.
- **DAG Folder**: Contains DAG files; the scheduler reads this to determine what to run and when.
- **Metadata Database**: Stores the state of workflows and tasks. Required for Airflow to function properly.

Airflow Architecture: Diagram



Execution in Airflow

Scheduler triggers Task instances based on:

- DAG schedule
- Task dependencies
- Execution state (e.g., success/failure)
- Each Task run needs one slot (available Task execution "seat" on a worker)
- Total slots = parallelism limit (per worker and globally)

If no slots are free:

- Tasks are queued until a slot is available
- Sensors can block slots, wasting resources, unless they are deferrable operators

XCOM: Cross-communication

Use case: Task 1 is uploading a file, after which,Task 2 must use that file. Both Tasks are executed asynchronously, but in the dependency order specified. How does Task 1 communicate the file path to Task 2?

XCOM is your answer. It simplifies passing messages between Tasks, but is **specifically for small amounts** of data.

For large amounts of data, cache it and pass that location onward instead.

XCOM: Cross-communication

```python
@task
def get_data() -> str:
    # Generate sample data
    df = get_data()
    filepath = "/tmp/sample_data.csv"
    df.to_csv(filepath, index=False)

    print(f"Data written to {filepath}")
    return filepath  # XCom value

@task
def load_into_database(csv_path: str):
    df = pd.read_csv(csv_path)
    print("Loading into database (simulated):")
    load_into_postgres(df)

with DAG(
    dag_id="xcom_csv_example",
    start_date=days_ago(1),
    schedule_interval=None,
    catchup=False,
    tags=["example"],
) as dag:
    filepath = get_data()
    load_into_database(filepath)

```

```python
def get_data(ti):
    ...
    print(f"Data written to {filepath}")

    # Manual XCom push
    ti.xcom_push(key="csv_path", value=filepath)

def load_into_database(ti):
    # Manual XCom pull
    filepath = ti.xcom_pull(key="csv_path", task_ids="get_data_task")
    ...

with DAG(
    ...
) as dag:

    get_data_task = PythonOperator(
        task_id="get_data_task",
        python_callable=get_data,
    )

    load_task = PythonOperator(
        task_id="load_into_database",
        python_callable=load_into_database,
    )

    get_data_task >> load_task

```

Dynamic DAGs: We're writing Python, aren't we?

DAGS generated with Python code are *dynamic*. When used with care, dynamic DAGS can simplify code and add real functionality. When used without care, they add amazing complexity and confusion.

```
1  with DAG("loop_example", ...):
2      first = EmptyOperator(task_id="first")
3      last = EmptyOperator(task_id="last")
4
5      options = ["branch_a", "branch_b", "branch_c", "branch_d"]
6      for option in options:
7          t = EmptyOperator(task_id=option)
8          first >> t >> last
```

DAG dependencies

DAGs are flowcharts but sometimes you want to flowchart your flowcharts.

DAG-to-DAG dependencies are defined by either Triggering or Waiting:

- TriggerDagRunOperator: Start another DAG
- ExternalTaskSensor: Wait for a Task (likely the end of a different DAG)

Warning: This is a loosely-coupled dependency that can turn your nice new tool for transparency and visualization into a nightmare. Use with care and don’t nest too deeply.

Logical DAGs: When there are different options

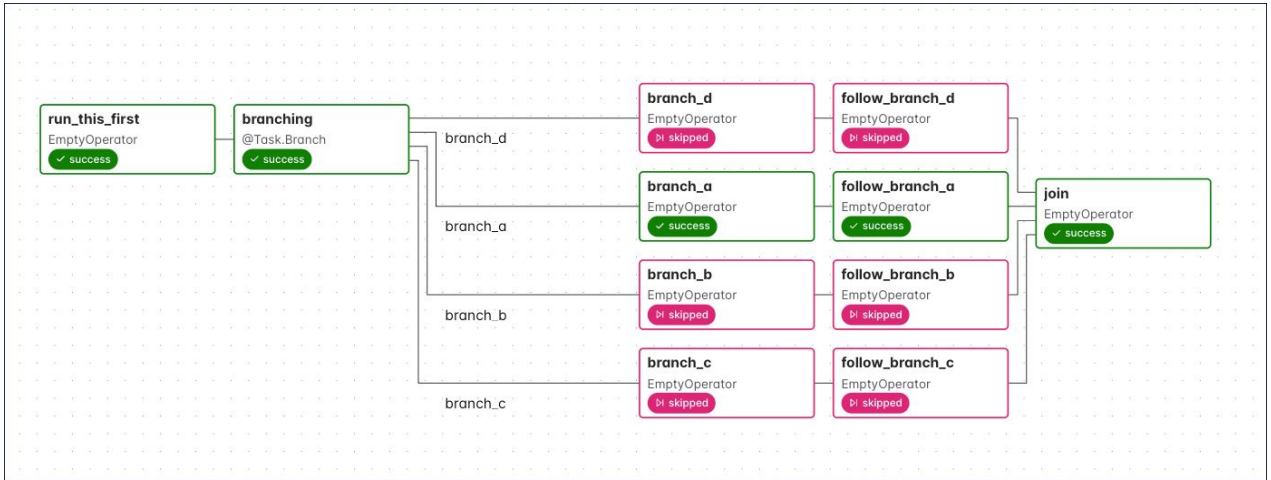
@task.branch (BranchPythonOperator) uses *if* logic to select flowchart Tasks, turning a DAG into a logic tool.

```python
# from airflow.sdk import task

result = 1

@task.branch
def choose_branch(result):
    if result > 0.5:
        return ['task_a', 'task_b']
    return ['task_c']

choose_branch(result)
```



The problem with sensors

- A sensor waits for e.g. 2 hours.
- It occupies a slot for that time.
- It’s blocking a slot for that time.

Solution:

Use a Deferrable Operator to free the slot during the wait.

- Deferrable Operators use Triggerer, which waits for events in the background, and, when the condition is met, resumes the Task and uses the slot.
- Triggerer requires an additional lightweight process to be run alongside Airflow, which is why it isn't enabled by default.

Break Time

Custom Operators: Python

Let’s write our own Operator using the @task decorator. Write a Python function to:

- Import the Iris dataset and store it locally.
- Define the mean of each feature per species
- Show correlation between petal length and petal width
- Define global mean vector (average flower)

```python
from sklearn.datasets import load_iris
import pandas as pd

# Load as a Bunch object
iris = load_iris()

# Convert to DataFrame
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df["target"] = iris.target

print(df.head())

```

The **Iris dataset** is one of the most well-known beginner datasets in machine learning. It contains 150 samples of 3 iris species with 4 numeric features (sepal/petal length/width).

Using @task

We can now easily add the @task decorator to turn our Functions into Tasks.

- How would you create the DAG from here?

```python
@task
def import_iris_dataset() -> str:
    """Download the Iris dataset and save it locally as a CSV file."""
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df["species"] = pd.Categorical.from_codes(iris.target, iris.target_names)
    filepath = "/tmp/iris.csv"
    df.to_csv(filepath, index=False)
    return filepath

@task
def mean_per_species(csv_path: str):
    """Compute mean of each feature per species."""
    df = pd.read_csv(csv_path)
    result = df.groupby("species").mean()
    print("Mean per species:\n", result)

@task
def petal_length_width_corr(csv_path: str):
    """Compute correlation between petal length and petal width."""
    df = pd.read_csv(csv_path)
    corr = df["petal length (cm)"].corr(df["petal width (cm)"])
    print(f"Correlation between petal length and width: {corr:.2f}")

@task
def global_mean_vector(csv_path: str):
    """Compute the global mean of all features."""
    df = pd.read_csv(csv_path)
    numeric_cols = df.select_dtypes(include="number")
    global_mean = numeric_cols.mean()
    print("Global mean vector:\n", global_mean)
```

Building DAGs: Intuitive Python (but only if you're dutch)

- Why does this work? Does it?

```python
from airflow import DAG
from airflow.utils.dates import days_ago

with DAG(
    dag_id="iris_stats_dag",
    start_date=days_ago(1),
    schedule_interval=None,
    catchup=False,
) as dag:
    path = import_iris_dataset()
    mean_per_species(path)
    petal_length_width_corr(path)
    global_mean_vector(path)

```

A long time ago in a company far away

Disjointed ETLs**:** Ingestion and Transformation were handled in 10+ tools

- Ingestion could be using Kubernets Pods, cronjobs, Pentaho, etc
- Transformation could be in Looker, BigQuery, Scheduled Queries, etc.
- Automations were run from local computers

The plan: Move to Airflow.

Standardisation: What were we doing?

When we took inventory, we were facing the following challenges:

- **Ingestion**: A Kubernetes Pod configured with our own Importer Library.
- **Transformation**: SQL statements that either wrote table-to-table or table-to-view.
  - Needed scheduling and dependency.
- **Custom Pods**: A few special cases, such as A/B testing, machine learning and recommender systems

So…

The Age of Airflow

- We wrote a new Operator/Task: The Importer
  - Configured our importer object and retrieved credentials to authenticate and ingest data
- We wrote a small library: sql-transformation-pods
  - Parsed yaml files with timing and dependencies (grouped in folders) into DAGs and Tasks
- We generalized for other use-cases: A pre-configured GKEPodOperator that allowed us to call for a specific container.

And there we had it: The New Grand Unified Data Platform Tool.

With Airflow, we moved ingestion and transformation from 10 tools to 1, created data visualizations for all pipelines, and centralized credentials and monitoring.

The Age of Airflow

Amusingly, we were using yaml files with SQL statements, dependencies, sometimes unit tests, timing, etc?

Sounds like the **data build tool**

Next week: data build tool (dbt)

They will get angry if you capitalize it. It’s in the certification.