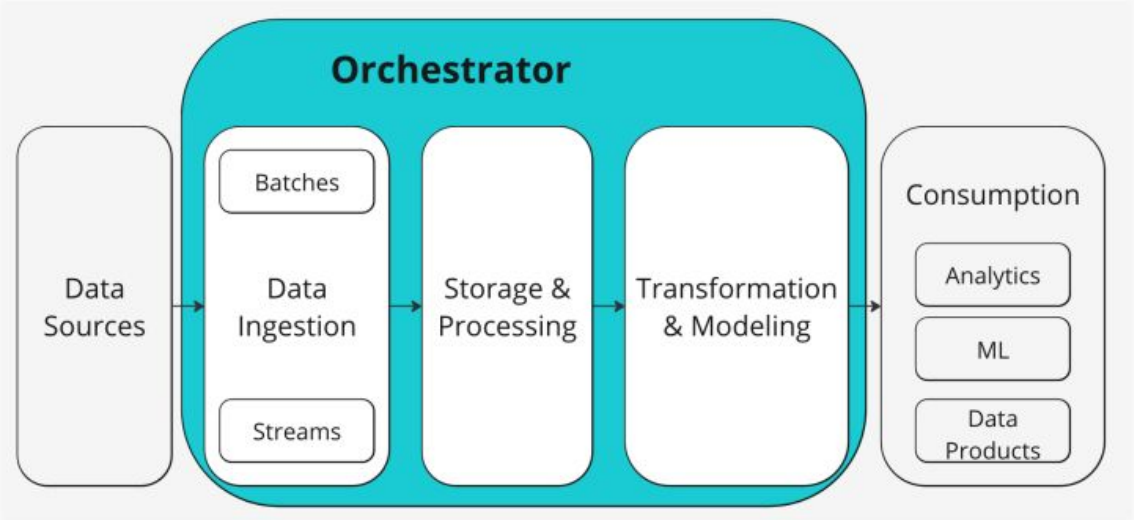
Orchestration and airflow

Orchestration

Automated process of managing, coordinating, and optimizing data pipelines and workflows. It ensures that data flows efficiently and accurately from one system to another, maintaining quality and integrity.



GCP workflows

Google Workflows

Fully managed orchestration for APIs and cloud services

Serverless orchestration: No infrastructure to manage

Step-by-step execution : Write workflows in YAML/JSON

Native GCP integration BigQuery, Cloud Functions, Pub/Sub, Cloud Run, etc.

API-first Connects internal & external REST APIs easily

Built-in error handling & retries

Pay-per-use Cost scales with executions

YAML

```yaml
main:
    params: [event]
    steps:
        - init:
            assign:
                - project_id: ${sys.get_env("GOOGLE_CLOUD_PROJECT_ID")}
                - event_bucket: ${event.data.bucket}
                - event_file: ${event.data.name}
                - target_bucket: ${"input-" + project_id}
                - job_name: parallel-job
                - job_location: us-central1
        - check_input_file:
            switch:
                - condition: ${event_bucket == target_bucket}
                  next: run_job
                - condition: true
                  next: end
        - run_job:
            call: googleapis.run.v1.namespaces.jobs.run
            args:
                name: ${"namespaces/" + project_id + "/jobs/" + job_name}
                location: ${job_location}
                body:
                    overrides:
                        containerOverrides:
                            env:
                                - name: INPUT_BUCKET
                                  value: ${event_bucket}
                                - name: INPUT_FILE
                                  value: ${event_file}
            result: job_execution
        - finish:
            return: ${job_execution}
```

When to use it?

You need to orchestrate **serverless services** (Cloud Functions, Cloud Run, APIs).

You want a **lightweight, low-maintenance** solution with no cluster management.

Your workflow is primarily API calls, event-driven tasks, or short-lived automations.

You prefer **pay-per-use** over running a cluster 24/7.

For more complex tasks workflows may not be solution

Complex Data Pipelines: Many tasks with dependencies, branching, and backfills

Heavy Compute or Batch Jobs Long-running jobs (Spark, ML training, big ETL)

Advanced Monitoring & Observability No DAG visualization or per-task SLA dashboards. Harder to debug large, multi-step pipelines

Python-First Development Needed

- If your team prefers writing logic in Python vs. YAML/JSON

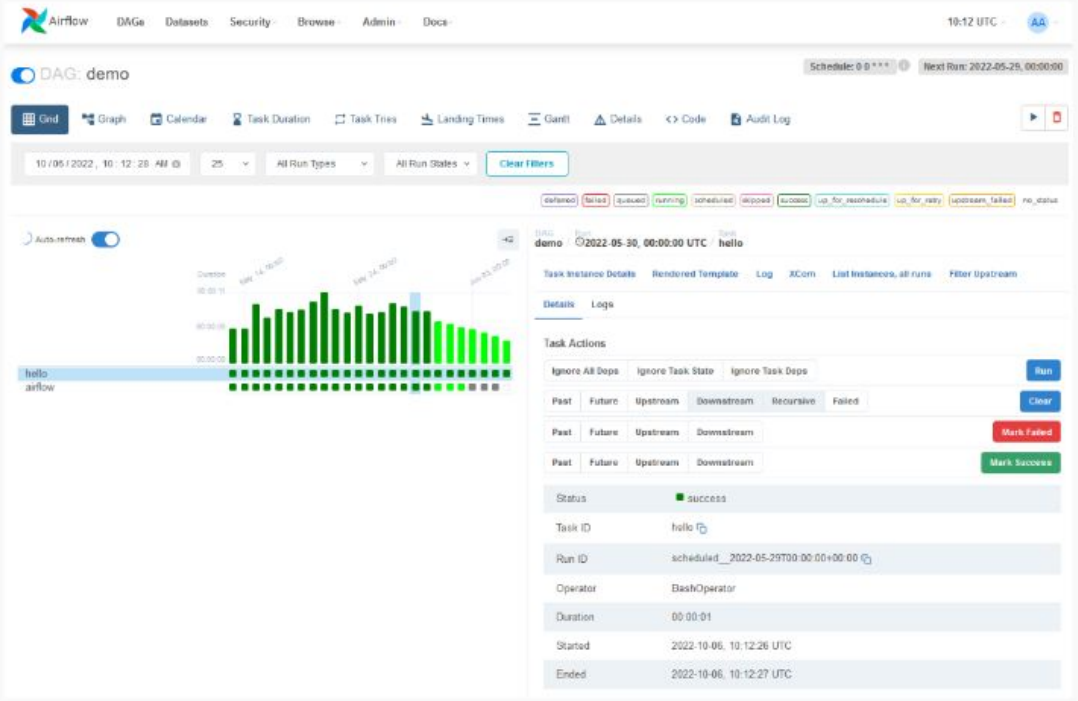
Apache Airflow

Apache Airflow is an open-source platform used to programmatically author, schedule, and monitor workflows.

It allows you to define workflows as code, making it easier to maintain and reproduce complex data pipelines.

It is written in Python and uses Directed Acyclic Graphs (DAGs) to represent workflows.

Historical fact: Originally made by Airbnb to manage their workflows and was open source, becoming an Apache incubator project.



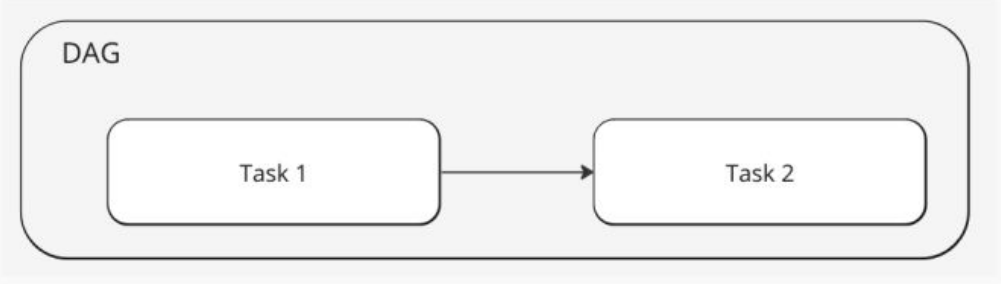
DAGs

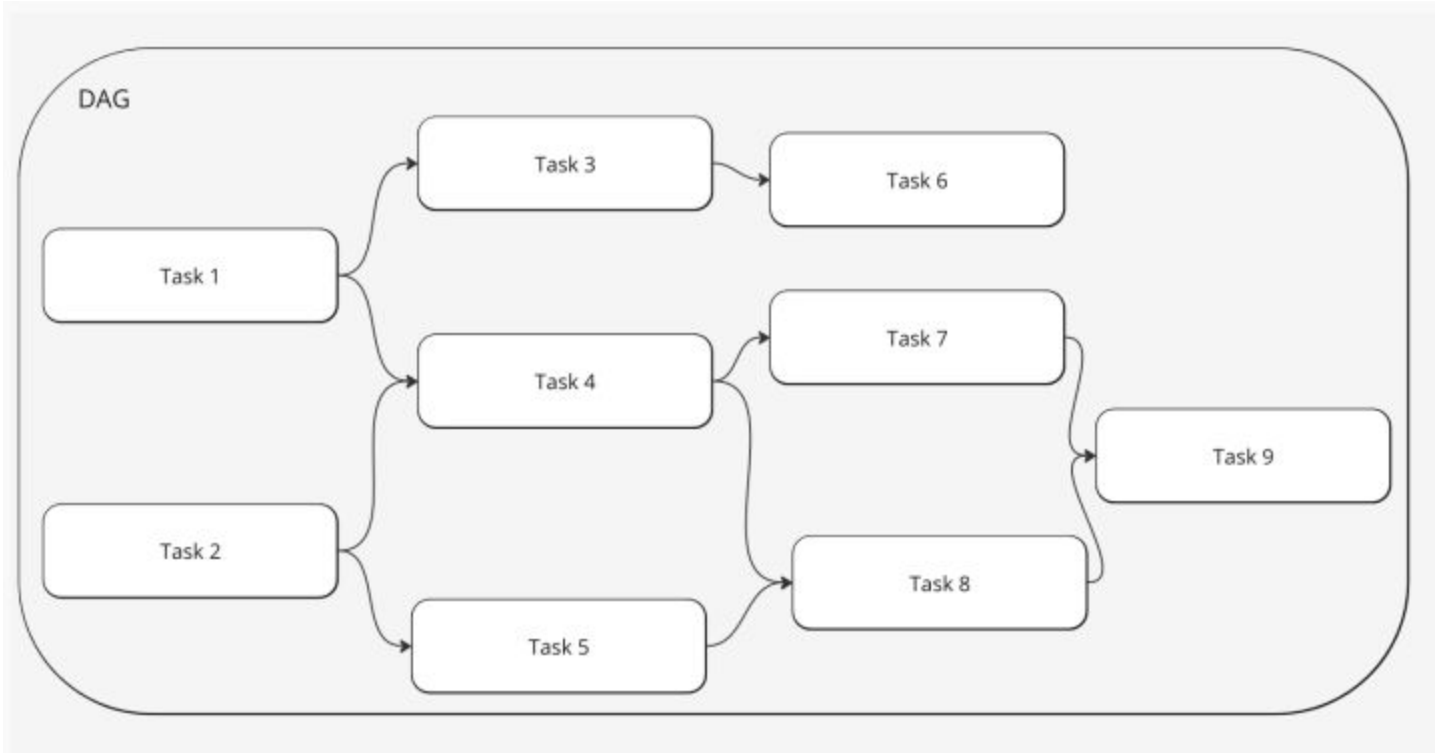
Workflows in Airflow are called **DAG**s

DAG stands for **Directed Acyclic Graphs**

DAGs consist of **tasks** and **dependencies** between them. **DAG**s ensure that workflows are executed in a specific order without cycles

Directed - Each DAG points somewhere
Acyclic - DAG only points one direction, not backwards
Graph - the tasks map out in a network of tasks



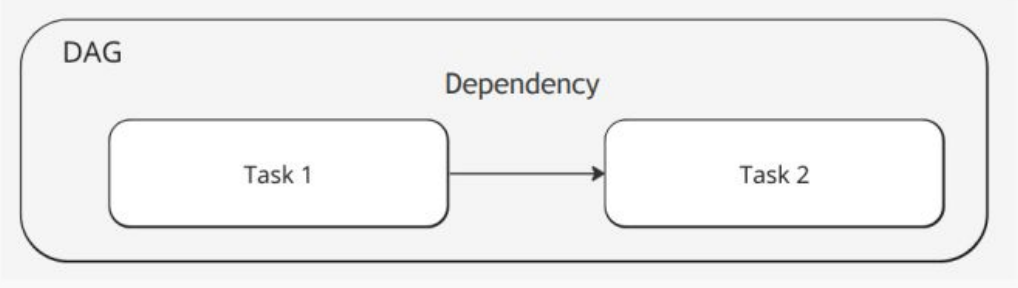

DAG
Task 1
Task 2
Task 3
Task 4
Task 5
Task 6
Task 7
Task 8
Task 9

Tasks, Dependencies and Operators

Tasks represent units of work in a **DAG.** It represents a callable or an executable action (e.g., running a Python function, executing a shell command).

Dependencies define the order in which **tasks** should be executed. Task 2 is the **downstream** of task 1. Task 1 is the **upstream** of task 2.

Operators are the building blocks of DAGs in Airflow. They define a single task, which can be anything from running a shell command to executing Python code or interacting with external systems. If we say **DAGs** describe how to run a workflow, then **operators** determines what gets done in a task



Operators

Operators are prebuilt task templates that let you connect to systems and run work without writing all the boilerplate.

Category	Examples	Purpose
Core Operators	`PythonOperator`, `BashOperator`, `BranchPythonOperator`	Glue code, scripts, control flow
Cloud / GCP	`BigQueryInsertJobOperator`, `GCSToBigQueryOperator`, `CloudRunExecuteJobOperator`, `CloudFunctionInvokeFunctionOperator`	Submit jobs to GCP services
Database	`PostgresOperator`, `SnowflakeOperator`, `MySQLOperator`	Run SQL or DDL/DML on databases
File Transfer	`SFTPOperator`, `GCSToLocalOperator`, `LocalFilesystemToGCSOperator`	Move files between systems
Messaging / APIs	`HttpOperator`, `SlackAPIPostOperator`, `DiscordWebhookOperator`	Send notifications or call APIs
ML / Big Data	`DataprocSubmitJobOperator`, `VertexAIOperator`, `SageMakerTrainingOperator`	Kick off ML or data jobs

Building a DAG with PythonOperator

Import and definitions of functions

```
from datetime import datetime
from airflow import DAG
from airflow.operators.python import PythonOperator

# 1) Define the Python functions you want to run
def my_task_1_function():
    #####Do something

def my_task_2_function():
    #####Do something alse
```

The DAG definition

```
# 2) Define the DAG
with DAG(
    dag_id="my_simple_dag",           # Unique name for your DAG
    start_date=datetime(2025, 1, 1), # When scheduling starts
    schedule="@daily",                # How often to run (@daily, @hourly, cron, None)
    catchup=False,                    # Skip past runs when first started
    tags=["example"],                 # Optional: for organizing in UI
) as dag:
```

PythonOperator

```
) as dag:

    # 3) Define the PythonOperator
    my_task_1 = PythonOperator(
        task_id="task_1",       # Unique name for this task
        python_callable=my_task_1_function,  # The function to run
    )

    my_task_2 = PythonOperator(
        task_id="task_2",       # Unique name for this task
        python_callable=my_task_2_function,  # The function to run
    )
```

Dependencies

```
#dependencies here:
my_task_1 >> my_task_2
```

```
) as dag:

    ingest = CloudRunExecuteJobOperator(
        task_id="ingest_raw",
        project_id=PROJECT, region=REGION, job_name="weather-ingest-job",
        wait_until_finished=True,
        overrides={"containerOverrides": [{"args": ["--date", "{{ ds }}"]}]},
    )

    transform = DbtCloudRunJobOperator(
        task_id="dbt_transform",
        job_id=12345, account_id=67890, # or use CloudRunExecuteJobOperator to run dbt container
    )

    infer = CloudRunExecuteJobOperator(
        task_id="batch_infer",
        project_id=PROJECT, region=REGION, job_name="weather-infer-job",
        wait_until_finished=True,
        overrides={"containerOverrides": [{"args": ["--date", "{{ ds }}"]}]},
    )

    ingest >> transform >> infer
```

Words of caution

You can technically deploy your code as python operators and make your airflow not only orchestrate but also run the pipeline.

BUT:

its not a standard practice.

Let the services (BQ, Cloud run etc) do the heavy lifting and keep Airflow for what it is, orchestration tool and not a way to execute your code.