

***Section A – Basics***

**1.What is Apache Airflow and why is it used?**

**Apache Airflow** is an open-source platform used to create, schedule, and monitor workflows (data pipelines). It allows you to define complex data processing tasks as Directed Acyclic Graphs (DAGs) using Python code.

**Why Airflow is Used:**

Airflow is mainly used for orchestrating and automating data workflows — especially in data engineering and analytics environments.

**2. Define a DAG. What does each part of the acronym stand for?**

A DAG in Apache Airflow stands for Directed Acyclic Graph.

**Definition:**

A DAG is a collection of tasks organized in a way that shows dependencies and execution order — defining *how your workflow runs in Airflow*.

It ensures tasks run in a specific order without creating loops.

**Acronym Breakdown:**

* **D → Directed:**  
  Each edge (arrow) has a direction — showing the flow from one task to another (e.g., Task A → Task B).
* **A → Acyclic:**  
  There are no cycles or loops — a task cannot depend on itself, directly or indirectly.
* **G → Graph:**The entire workflow is represented as a graph of connected tasks.

**3. Explain the difference between a DAG and a Task**.

**DAG (Directed Acyclic Graph):**

A DAG represents the *entire workflow* in Apache Airflow. It defines the structure, schedule, and dependencies between tasks.

* It ensures tasks run in the correct order.
* It has no loops (acyclic).
* It’s defined in a Python script and can run on a schedule (e.g., daily).

**Example:**  
A DAG for a daily ETL pipeline — “Extract → Transform → Load.”

**Task:**

A Task is a single unit of work within a DAG — it performs one specific operation like reading a file, running a query, or sending an email.

* Tasks are created using operators (e.g., PythonOperator, BashOperator).
* Each task represents one step in the overall workflow.

**Example:**  
A task that runs a Python function to extract data from an API.

**4. Why should workflows be “Directed Acyclic Graphs” in Airflow?**

Workflows in Airflow are modeled as Directed Acyclic Graphs (DAGs) to ensure clear execution order, no infinite loops, and dependable scheduling.

**1. Directed — Ensures Clear Flow of Execution**

* Each edge (arrow) points in one direction — from an upstream task to a downstream task.
* This direction defines *which task runs before which*, creating a predictable and logical workflow.

**Example:**extract → transform → load  
Data always flows in one direction.

**2. Acyclic — Prevents Circular Dependencies**

* “Acyclic” means there are no cycles or loops in the graph.
* Without this rule, a task could depend on itself (directly or indirectly), causing infinite loops and failed executions.

**Example:**If Task A depends on Task B, and Task B depends on Task A → it would never complete**.**

**3. Graph — Represents Relationships Between Tasks**

* The workflow is expressed as a graph that visually shows how tasks connect and depend on one another.
* This makes it easier to visualize, debug, and manage complex data pipelines.

***Section B – Core Concepts***

**1. Describe the role of the following Airflow components:**

* **Webserver**
* **Scheduler**
* **Metadata Database**

**Webserver**

* The Webserver provides the Graphical User Interface (GUI) for Apache Airflow.
* It allows users to view, trigger, pause, and monitor DAGs and tasks.
* Through the web UI, you can see task statuses, logs, execution history, and dependencies.
* It continuously fetches metadata from the Airflow database to display up-to-date information.

**Role:**  
To give a **visual dashboard** for managing and monitoring workflows.

**Scheduler**

* The Scheduler is responsible for scheduling and triggering tasks based on DAG definitions and their schedules (e.g., daily, hourly).
* It checks the DAGs regularly, determines which tasks need to run, and sends them to the executor for execution.
* It ensures tasks run in the correct order and time according to dependencies.

**Role:**  
To orchestrate and schedule task execution in the right sequence and timing.

**Metadata Database**

* The Metadata Database stores all the internal information about Airflow’s state.
* It keeps records of:
  + DAGs and task definitions
  + Task run history and statuses (success, failed, running, etc.)
  + User actions, connections, and configurations
* Common databases used: PostgreSQL or MySQL.

**2. What is the purpose of the airflow db init command?**

The command airflow db init (or airflow db initialize) is used to set up and initialize the Airflow metadata database for the first time.

**What It Does:**

When you run airflow db init, Airflow performs the following actions:

1. **Creates the Metadata Database Schema**
   * Sets up all necessary tables, indexes, and relationships in the database (e.g., PostgreSQL, MySQL, or SQLite).
   * These tables store information about DAGs, task instances, users, connections, logs, etc.
2. **Initializes Default Configurations**
   * Inserts default values such as admin roles, connection types, and variable settings.
3. **Prepares Airflow for First Use**
   * Ensures that the Webserver, Scheduler, and other components can read and write data properly.
4. **What is the significance of start\_date and schedule\_interval in a DAG?**

**start\_date:**

* The start\_date defines when a DAG should begin running.
* It marks the first logical execution date of the DAG’s schedule.
* Airflow uses this to determine when to trigger the first run and how to calculate future runs**.**

**Key Points:**

* The first DAG run happens *after* the start\_date.
* It should always be set to a past or current datetime, not a future one.
* Each DAG run is scheduled at the *end* of the interval it represents.

**schedule\_interval:**

* The schedule\_interval defines how often the DAG should run.
* It can be expressed as a cron expression, a preset, or a timedelta.

**4. What does catchup=False do, and when would you use it?**

The catchup parameter in a DAG controls whether Airflow should run all past scheduled DAG runs between the start\_date and the current date.

**What catchup=False Does:**

When you set catchup=False, Airflow will not run any backfilled (past) DAG runs — it will only run the latest scheduled run onward.

**When to Use catchup=False:**

You should use catchup=False when:

* You don’t need historical runs (e.g., real-time or current-only workflows).
* Your DAG processes live or streaming data.
* Backfilling would create unnecessary load or duplicate data.

***Section C – Operators & Execution***

1. **What is an Operator? Give two examples.**

An Operator in Apache Airflow is a template for a single task. It defines what action or work a task will perform in a DAG, such as running a Python function, executing a Bash command, or transferring data.

* Operators do not define dependencies; they just define the type of work.
* Each task in a DAG is an instance of an Operator.

**2. Examples of Operators:**

1. PythonOperator

2. BashOperator

**2. How does Airflow handle task failures and retries?**

Airflow has built-in mechanisms to detect task failures and automatically retry tasks based on settings defined in the DAG or task.

**Task Failures**

* A task fails if it raises an exception or exits with an error during execution.
* Failed tasks are recorded in the metadata database and visible in the Airflow UI.
* Downstream tasks may be paused depending on dependency rules until the failure is resolved.

**Retries**

Airflow allows automatic retries for failed tasks using task parameters:

* retries specifies how many times a failed task should be retried.
* retry\_delay specifies how long Airflow waits before retrying the task.

Together, they allow Airflow to automatically handle transient failures without manual intervention.

**3. What is XCom and how is it useful?**

XCom stands for “Cross-Communication”. It is a feature in Apache Airflow that allows tasks to exchange small pieces of data during a DAG run.

* XComs are stored in the Airflow metadata database.
* They enable inter-task communication without using external storage.

**How XCom Works:**

* Each task can push a value to XCom.
* Downstream tasks can pull that value from XCom.
* This allows dynamic task execution based on outputs from previous tasks.

**Why XCom is Useful:**

1. **Data Sharing Between Tasks:** Pass results from one task to another without writing to files or databases.
2. **Dynamic Workflows:** Downstream tasks can make decisions based on upstream outputs.
3. **Simplifies DAG Design:** Reduces the need for external storage for small messages.

**4.Explain the difference between BashOperator and PythonOperator**

**BashOperator:**

* Purpose: Executes Bash commands or shell scripts.
* Usage: Suitable for running system commands, scripts, or CLI tools.
* Return Values: Can capture standard output (stdout) as XCom if do\_xcom\_push=True**.**

**Example:**

from airflow.operators.bash import BashOperator

bash\_task = BashOperator (

task\_id='list\_files',

bash\_command='ls -l /home/user'

)

**PythonOperator**

* Purpose: Executes Python functions directly.
* Usage: Suitable for running Python logic, data processing, or function calls.
* Return Values: Can return any Python object directly to XCom.

**Example:**

from airflow.operators.python import PythonOperator

def greet():

print("Hello")

python\_task = PythonOperator(

task\_id='greet',

python\_callable=greet

)

***Section D – Real-World Use***

**1. Give one real-world example where Airflow can be used for ETL.**

**Real-World Example: Daily Sales Data ETL Pipeline**

**Scenario:**  
An e-commerce company wants to process daily sales data from multiple sources (website, mobile app, and third-party marketplaces) and load it into a data warehouse for analytics and reporting.

**ETL Workflow Using Airflow:**

1. **Extract (E):**
   * Use Airflow tasks to pull sales data from APIs, CSV files, or databases.
   * Example: Extract yesterday’s sales data from MySQL and an S3 bucket.
2. **Transform (T):**
   * Clean and transform the data using PythonOperator or SparkOperator.
   * Example: Remove invalid records, calculate total\_amount = quantity \* price, and enrich with product info.
3. **Load (L):**
   * Load the transformed data into a data warehouse like Snowflake, BigQuery, or Redshift.
4. **Scheduling & Monitoring:**
   * Airflow DAG is scheduled to run daily at 2 AM.
   * Failures are retried automatically, and logs are monitored in the Web UI.
5. **Why is it recommended to keep DAG scripts lightweight and avoid heavy computations inside them?**

In Airflow, a DAG script defines the workflow structure — tasks, dependencies, schedules — and is parsed frequently by the scheduler and webserver. Heavy computations inside DAG files can cause performance and reliability issues.

**Key Reasons:**

1. **Scheduler Performance:**
   * The scheduler parses all DAG files frequently (every few seconds).
   * Heavy computations can slow down parsing, delaying task scheduling.
2. **Webserver Responsiveness:**
   * The web UI reads DAG definitions to display graphs and task details.
   * Long-running code can make the UI slow or unresponsive.
3. **Maintainability:**
   * DAG scripts should define structure, not execution logic.
   * Keeping computation inside operators (PythonOperator, BashOperator, SparkOperator) improves readability and modularity.
4. **Avoid Unintended Execution:**
   * Code at the top level of the DAG runs every time the DAG is parsed, not just when tasks run.
   * Heavy computation here can be wasted work and increase resource usage.

**3. Why should every DAG have a unique dag\_id ?**

The dag\_id is the unique identifier for a DAG in Airflow. It is used by the scheduler, webserver, and metadata database to track and manage DAG runs.

**Key Reasons for Uniqueness:**

1. **Avoid Conflicts in the Metadata Database:**
   * Airflow stores DAG and task execution history in the metadata database.
   * If two DAGs have the same dag\_id, their task runs and logs may overwrite each other, causing data loss or confusion.
2. **Clear Identification in UI and Logs:**
   * The webserver displays DAGs based on dag\_id.
   * Unique IDs ensure you can accurately monitor, trigger, or troubleshoot a specific DAG.
3. **Scheduler Accuracy:**
   * The scheduler uses dag\_id to determine which DAGs to run and track.
   * Duplicate IDs can lead to scheduling errors or skipped runs.
4. **Maintainability:**
   * Unique dag\_ids make it easier for teams to organize, version, and maintain DAGs in large projects.

**4. How does Airflow ensure workflows run in the correct order?**

Airflow uses DAGs (Directed Acyclic Graphs) and task dependencies to enforce the correct execution sequence of tasks.

**1. Directed Acyclic Graph (DAG)**

* Each DAG defines tasks as nodes and dependencies as directed edges.
* The direction of edges ensures tasks run from upstream to downstream.

**2. Task Dependencies**

* You explicitly define dependencies using operators like >> (set downstream) or << (set upstream).

**Example:**

extract >> transform >> load

* extract runs first, followed by transform, then load.
* Airflow guarantees no task starts before its upstream tasks are completed successfully.

**3. Scheduler Enforcement**

* The Airflow scheduler monitors DAGs and triggers tasks only when:
  1. The scheduled execution time has arrived.
  2. All upstream tasks have succeeded.

**4. Retry & Failure Handling**

* If a task fails, downstream tasks wait until retries succeed or the task is manually cleared.
* This ensures the workflow respects the defined order even in case of failures.