

**Apache Airflow Interview Questions**

## **Basic Level:-**

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

[Apache Airflow®](https://github.com/apache/airflow) is an open-source platform for developing, scheduling, and monitoring batch-oriented workflows. Airflow’s extensible Python framework enables you to build workflows connecting with virtually any technology. A web interface helps manage the state of your workflows. Airflow is deployable in many ways, varying from a single process on your laptop to a distributed setup to support even the biggest workflows.

1. **Explain the concept of Directed Acyclic Graphs (DAGs) in Airflow.**

**Directed:** The flow goes in one direction from task to task. In a DAG, tasks are linked by dependencies, which means a task can only proceed once its dependencies are completed.

**Acyclic:** It doesn’t loop or cycle. There are no circular dependencies; a task cannot depend on itself, either directly or indirectly.

**Graph:** A DAG is a collection of nodes (tasks) and edges (dependencies), forming a structure that represents a workflow.

**DAGs are code:** In Airflow, DAGs are defined using Python code. You create a Python script to define the DAG and its structure.

**Tasks as nodes:** Each node in the DAG represents a unit of work or a task (e.g., running a Python function, querying a database, or executing a bash command).

**Dependencies as edges:** The edges between the nodes represent dependencies, meaning one task needs to finish before another can start.

**Scheduling:** DAGs are designed to be triggered on a schedule (e.g., daily, weekly) or based on specific conditions.

**Task dependencies:** You can define task dependencies using methods like:

* + set\_upstream() – Defines that a task should run after another.
  + set\_downstream() – Defines that a task should run before another.
  + >> or << operators – Simplified syntax to declare dependencies (task\_1 >> task\_2).

Example of Airflow DAG:-

from airflow import DAG

from airflow.operators.python\_operator import PythonOperator

from datetime import datetime

# Define the DAG

dag = DAG(

    'example\_dag',

    description='A simple DAG',

    schedule\_interval='0 12 \* \* \*',  # Runs daily at noon

    start\_date=datetime(2023, 1, 1),

    catchup=False

)

# Define a Python task

def my\_task():

    print("Executing task")

# Create tasks

task\_1 = PythonOperator(

    task\_id='task\_1',

    python\_callable=my\_task, ##Python code/function which needs to be executed

    dag=dag

)

task\_2 = PythonOperator(

    task\_id='task\_2',

    python\_callable=my\_task, ##Python code/function which needs to be executed

    dag=dag

)

# Set dependencies

task\_1 >> task\_2  # task\_2 depends on task\_1

### **Example from official airflow website:-**

    from datetime import datetime

    from airflow import DAG

    from airflow.decorators import task

    from airflow.operators.bash import BashOperator

    # A DAG represents a workflow, a collection of tasks

    with DAG(dag\_id="demo", start\_date=datetime(2022, 1, 1), schedule="0 0 \* \* \*") as dag:

        # Tasks are represented as operators

        hello = BashOperator(task\_id="hello", bash\_command="echo hello")

        @task()

        def airflow():

            print("airflow")

        # Set dependencies between tasks

        hello >> airflow()

### **Key Concepts:**

* **Parallelism:** Airflow allows tasks within the same DAG to run in parallel if no dependencies exist between them.
* **Dynamic DAGs:** Because DAGs are defined in Python, you can dynamically generate them based on conditions, inputs, or configurations.
* **Idempotency:** DAGs are designed to be idempotent, meaning re-running the same workflow should yield the same results.

1. **How do you define tasks in Airflow?**

### DAGs vs Tasks:

* **DAGs:** A structure that holds the workflow (tasks and their dependencies).
* **Tasks:** The individual units of work within a DAG.

1. **What are the different types of operators in Airflow?**

#### 1. **PythonOperator**

Used to run a Python function as a task.

    from airflow import DAG

    from airflow.operators.python\_operator import PythonOperator

    from datetime import datetime

    def my\_python\_task():

        print("Executing Python task")

    # Define the DAG

    dag = DAG('example\_dag', start\_date=datetime(2023, 1, 1))

    # Define the task

    task\_1 = PythonOperator(

        task\_id='python\_task',

        python\_callable=my\_python\_task,  # The Python function to run

        dag=dag

    )

#### 2. **BashOperator**

Used to execute Bash commands.

    from airflow.operators.bash\_operator import BashOperator

    task\_2 = BashOperator(

        task\_id='bash\_task',

        bash\_command='echo "Hello from Bash"',

        dag=dag

    )

#### 3. **PostgresOperator**

Used to execute SQL queries on a PostgreSQL database.

    from airflow.operators.postgres\_operator import PostgresOperator

    task\_3 = PostgresOperator(

        task\_id='postgres\_task',

        postgres\_conn\_id='my\_postgres\_connection',

        sql='SELECT \* FROM my\_table;',

        dag=dag

    )

#### **4. EmailOperator**

Used to send emails.

    from airflow.operators.email\_operator import EmailOperator

    task\_4 = EmailOperator(

        task\_id='send\_email',

        to='user@example.com',

        subject='Airflow Email',

        html\_content='<p>This is an email from Airflow</p>',

        dag=dag

    )

#### **5. DummyOperator**

This operator does nothing and is often used as a placeholder or to organize task dependencies.

    from airflow.operators.dummy\_operator import DummyOperator

    start\_task = DummyOperator(

        task\_id='start',

        dag=dag

    )

### **Defining Multiple Tasks**

You can define multiple tasks in the same DAG and set dependencies between them. Here's an example where multiple tasks are created and their order is set:

    # Define more tasks

    task\_2 = PythonOperator(

        task\_id='second\_task',

        python\_callable=my\_python\_task,

        dag=dag

    )

    task\_3 = BashOperator(

        task\_id='third\_task',

        bash\_command='echo "Task 3"',

        dag=dag

    )

    # Set dependencies between tasks

    task\_1 >> task\_2  # task\_2 runs after task\_1

    task\_2 >> task\_3  # task\_3 runs after task\_2

### **Task Dependencies:**

You can define the sequence in which tasks are executed by specifying dependencies:

* task\_1 >> task\_2: task\_2 will only run after task\_1 is complete.
* task\_2.set\_upstream(task\_1): Same as above, but more explicit syntax.
* task\_3.set\_downstream(task\_4): task\_4 will run after task\_3 is complete.

1. **How can you schedule a DAG in Airflow?**

#### **1. Cron Expressions:**

Airflow uses cron expressions to allow flexible scheduling. A cron expression follows the format:

 <minute> <hour> <day\_of\_month> <month> <day\_of\_week>

dag = DAG(

      'daily\_dag',

      schedule\_interval='30 2 \* \* \*',

      start\_date=datetime(2023, 1, 1)

  )

#### **2. Preset Intervals:**

Airflow provides several preset options to simplify common scheduling scenarios. These presets are strings like '@daily', '@hourly', etc.

* **@once**: Run the DAG only once when triggered manually.
* **@hourly**: Run the DAG every hour.
* **@daily**: Run the DAG once a day at midnight (00:00 UTC).
* **@weekly**: Run the DAG once a week at midnight on Sunday.
* **@monthly**: Run the DAG once a month at midnight of the first day of the month.
* **@yearly** (or @annually): Run the DAG once a year at midnight of January 1st.

Example for running daily:

    dag = DAG(

      'daily\_dag',

      schedule\_interval='@daily',

      start\_date=datetime(2023, 1, 1)

  )

#### **3. Time Delta (Relative Time Scheduling):**

If you want the DAG to run at intervals based on when it last ran, you can use the timedelta object from Python’s datetime module to specify relative intervals.

**Example for running every 5 minutes:**

    from datetime import timedelta

    dag = DAG(

        'five\_minute\_dag',

        schedule\_interval=timedelta(minutes=5),

        start\_date=datetime(2023, 1, 1)

    )

#### 4. **No Schedule (Manual Trigger Only):**

You can set a DAG to not run automatically and instead only be triggered manually. In this case, you set schedule\_interval=None.

    dag = DAG(

      'manual\_dag',

      schedule\_interval=None,  # This DAG will only run when triggered manually

      start\_date=datetime(2023, 1, 1)

  )

### **Important Parameters for Scheduling:**

#### **1. start\_date:**

* start\_date specifies when Airflow should start scheduling the DAG.
* It needs to be in the past or the present when the DAG is triggered for the first time. Airflow does not run DAGs retroactively unless catchup is enabled.

#### **2. catchup (Default=True):**

By default, Airflow will "catch up" on all the missed DAG runs between the start\_date and the current date. This behavior can be disabled by setting catchup=False.

## **Intermediate Level:**

1. **How do you monitor and manage workflows in Airflow?**
   * Airflow Web Interface (UI)
   * Airflow Command Line Interface (CLI)
   * Alerts and Notifications
   * SLAs (Service Level Agreements
   * Retries and Failure Handling

|  |  |
| --- | --- |
| **Tasks** | **Key Commands** |
| Start/Stop a DAG: | airflow dags pause <dag\_id> airflow dags unpause <dag\_id> |
| Trigger a DAG manually: | airflow dags trigger <dag\_id> |
| List all DAGs: | airflow dags list |
| List DAG runs: | airflow dags list-runs <dag\_id> |
| List tasks within a DAG: | airflow tasks list <dag\_id> |
| Check the status of a task: | airflow tasks state <dag\_id> <task\_id> <execution\_date> |
| View task logs: | airflow tasks logs <dag\_id> <task\_id> <execution\_date> |
| Mark a task as successful: | airflow tasks mark\_success <dag\_id> <task\_id> <execution\_date> |
|  |  |

**Email Notification:-**

    # Example of Email Notification

dag = DAG(

      'example\_dag',

      default\_args={

          'email': ['your\_email@example.com'],

          'email\_on\_failure': True,

          'email\_on\_retry': False,

      },

      schedule\_interval='@daily',

  )

**SLA:-**

    task\_1 = PythonOperator(

      task\_id='python\_task',

      python\_callable=my\_function,

      sla=timedelta(hours=1),  # SLA: task must complete within 1 hour

      dag=dag

  )

**Retries and failure handling:-**

    default\_args = {

      'retries': 3,  # Number of retries

      'retry\_delay': timedelta(minutes=5),  # Delay between retries

  }

1. **Explain the difference between Airflow Sensors and Operators.**

### Operators are discussed above. Sensor. A **Sensor** is a type of operator that is specifically designed to **wait** for a certain condition to be met before proceeding. Sensors are **stateful** because they keep checking (polling) until their condition is satisfied. They are often used for tasks that depend on external systems or events, such as waiting for a file to appear in a directory, a partition to be added to a database, or an API to return a specific status.

**Types of Sensors**:

* **FileSensor**: Waits for a file to appear in a specific location.
* **ExternalTaskSensor**: Waits for an external task in another DAG to complete.
* **HttpSensor**: Waits for an API response or a certain status from an HTTP request.
* **S3KeySensor**: Waits for a file or object to appear in an S3 bucket.

Example of file sensor:-

    from airflow.sensors.filesystem import FileSensor

    wait\_for\_file = FileSensor(

        task\_id='wait\_for\_file',

        filepath='/path/to/my/file.csv',

        poke\_interval=60,  # Check every 60 seconds

        dag=dag

    )

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Operators** | **Sensors** |
| **Purpose** | Perform an action (e.g., run code, execute a script). | Wait for a condition to be met (e.g., a file to appear, a task to complete). |
| **State** | Stateless (run and complete). | Stateful (can keep running until a condition is met). |
| **Execution Mode** | Executes immediately when dependencies are met. | Keeps checking/polling until the condition is satisfied. |
| **Resource Usage** | Minimal once executed. | Can be resource-intensive if waiting for a long time (unless using reschedule mode). |
| **Action vs. Condition** | Performs an action such as querying, sending, executing, etc. | Checks for an external condition, like a file or data being ready. |
| **Common Use Cases** | Run scripts, query databases, send emails. | Wait for file creation, task completion, API response, etc. |

1. **What are XComs in Airflow, and how do you use them?**
2. from airflow import DAG
3. from airflow.operators.python\_operator import PythonOperator
4. from datetime import datetime
5. import requests
7. # Task 1: Fetch data from an API
8. def fetch\_data(\*\*kwargs):
9. # Simulate an API call (e.g., get user data from a public API)
10. response = requests.get("https://jsonplaceholder.typicode.com/users")
12. if response.status\_code == 200:
13. data = response.json()  # Assume we're getting JSON data
14. # Push the fetched data into XCom using return (default key is 'return\_value')
15. return data
16. else:
17. raise ValueError("Failed to fetch data from API")
19. # Task 2: Process the fetched data (e.g., filter out data)
20. def process\_data(\*\*kwargas):
21. ti = kwargs['ti']
22. # Pull the data from Task 1 (fetch\_data) using XCom using a defalult key which is return value
23. fetched\_data = ti.xcom\_pull(task\_ids='fetch\_task')
25. # Example transformation: Filter users from a specific company
26. filtered\_data = [user for user in fetched\_data if user['company']['name'] == 'Romaguera-Crona']
28. # Push the processed data into XCom for downstream use
29. ti.xcom\_push(key='processed\_data', value=filtered\_data)
31. # Task 3: Save the processed data to a database or file (mocked here)
32. def save\_data(\*\*kwargs):
33. ti = kwargs['ti']
34. # Pull the processed data from Task 2 (process\_data) using a custom XCom key
35. processed\_data = ti.xcom\_pull(task\_ids='process\_task', key='processed\_data')
37. # Mock saving the data (in reality, this could be writing to a database or a file)
38. print(f"Saving the following data: {processed\_data}")
40. # Define the DAG
41. with DAG(
42. dag\_id='xcom\_practical\_example',
43. start\_date=datetime(2023, 1, 1),
44. schedule\_interval='@daily',
45. catchup=False,
46. ) as dag:
48. # Define tasks
49. fetch\_task = PythonOperator(
50. task\_id='fetch\_task',
51. python\_callable=fetch\_data,
52. provide\_context=True
53. )
55. process\_task = PythonOperator(
56. task\_id='process\_task',
57. python\_callable=process\_data,
58. provide\_context=True
59. )
61. save\_task = PythonOperator(
62. task\_id='save\_task',
63. python\_callable=save\_data,
64. provide\_context=True
65. )
67. # Set task dependencies
68. fetch\_task >> process\_task >> save\_task
    1. **How do you handle dependencies between tasks in a DAG?**

### **1. Using >> and << Operators (Preferred Method)**

The most common and intuitive way to define dependencies is by using the >> (set downstream) and << (set upstream) operators.

#### **Example:**

task1 >> task2  # task2 will execute after task1

task2 << task3  # task3 will execute before task2

This is equivalent to:

task1.set\_downstream(task2)

task3.set\_upstream(task2)

#### **Chain multiple tasks:**

#### You can easily chain multiple tasks with the >> operator.

task1 >> task2 >> task3  # task2 follows task1, task3 follows task2

### 2**. Using set\_downstream and set\_upstream Methods**

You can manually set dependencies using set\_downstream() and set\_upstream() methods.

#### Example:

task1.set\_downstream(task2)  # task2 depends on task1

task2.set\_upstream(task3)    # task2 depends on task3

### **3. Handling Multiple Dependencies**

You can create complex dependencies between tasks using multiple downstream or upstream tasks.

#### Example:

task1 >> [task2, task3]  # task2 and task3 both depend on task1

[task2, task3] >> task4  # task4 depends on both task2 and task3

* 1. **Explain the process of scaling Airflow for large-scale workflows.**

### **Scaling Strategies:**

1. **Use the Celery or Kubernetes Executor** to distribute tasks across multiple worker nodes or pods.
2. **Run multiple schedulers** for better load balancing and high availability.
3. **Deploy multiple web servers** behind a load balancer to handle increased UI traffic.
4. **Scale the metadata database** with read replicas and vertical scaling.
5. **Optimize DAG and task concurrency settings** to allow more parallel execution.
6. **Use distributed log storage** to manage task logs at scale.
7. **Set up comprehensive monitoring and alerting** for proactive issue detection.
8. **Implement high availability** for key components like the scheduler and database.
9. **Leverage containerization and IaC** for easy deployment and scaling.

## **Advanced Level: Out of scope**

1. **How do you implement retry logic and error handling in Airflow tasks?**

Retries and failure handling:-

    default\_args = {

      'retries': 3,  # Number of retries

    'retry\_delay': timedelta(minutes=5),  # Delay between retries

  }

1. **Describe how you would set up and manage Airflow in a production environment.**
2. **How can you customize and extend Airflow with plugins?**
3. **Explain the process of dynamically generating DAGs in Airflow.**
4. Discuss best practices for optimizing Airflow performance and resource utilization.
5. How do you manage and secure sensitive data within Airflow workflows?