Apache Airflow Architecture, Spark Integration, and Kubernetes Deployment Guide

# Table of Contents

1. Apache Airflow Architecture and Components

2. Integrating Apache Spark with Airflow

3. Deploying Airflow in Kubernetes using Helm

- Step-by-Step Procedure for Airflow Deployment

- Required Configuration Changes for Spark Integration

# 1. Apache Airflow Architecture and Components

## Overview of Apache Airflow

Apache Airflow is a platform for orchestrating workflows and pipelines. Workflows are represented as Directed Acyclic Graphs (DAGs) and consist of tasks that can be scheduled, monitored, and executed sequentially or in parallel.

## Core Components of Airflow

1. Scheduler

- The Scheduler is responsible for scheduling tasks and DAGs (workflows). It reads DAGs and determines which tasks need to be run based on the defined schedule and their dependencies.

2. Web Server (UI)

- The Web Server provides a user interface to manage DAGs and monitor the status of each task. Users can visualize DAGs, trigger manual task runs, and track progress.

3. Metadata Database

- The Metadata Database (usually PostgreSQL or MySQL) stores the state of DAGs, task instances, and job execution history. This database is critical for tracking task statuses and logs.

The metadata database in Apache Airflow is crucial for storing all the internal states of DAGs, tasks, and workflows. In this section, we'll go over the main tables that are created when using a PostgreSQL database (or any other supported database) as the backend for Airflow.

Each table has a specific role in keeping track of task instances, DAG runs, job schedules, and various configurations. Below is a detailed explanation of the key tables in Airflow’s metadata database.

### 1. ****dag****

This table stores information about the Directed Acyclic Graphs (DAGs) defined in Airflow.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| dag\_id | VARCHAR(250) | Unique identifier for each DAG. |
| root\_dag\_id | VARCHAR(250) | Used for SubDAGs to point to the parent DAG. |
| is\_paused | BOOLEAN | Indicates if the DAG is currently paused. |
| is\_subdag | BOOLEAN | True if the DAG is a SubDAG. |
| is\_active | BOOLEAN | Indicates if the DAG is active. |
| last\_parsed\_time | TIMESTAMP | Timestamp of the last time this DAG was parsed. |

### 2. ****dag\_run****

This table tracks the execution of a DAG for each run (whether triggered by a schedule or manual).

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for the DAG run. |
| dag\_id | VARCHAR(250) | Foreign key referencing the dag table. |
| execution\_date | TIMESTAMP | The execution date of the DAG run. |
| state | VARCHAR(20) | State of the DAG run (e.g., running, success, failed). |
| run\_id | VARCHAR(250) | Unique identifier for each run of the DAG. |
| external\_trigger | BOOLEAN | Indicates if the DAG run was externally triggered. |

### 3. ****task\_instance****

This table records each task's execution details within a DAG.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| task\_id | VARCHAR(250) | The unique identifier for the task within the DAG. |
| dag\_id | VARCHAR(250) | Foreign key referencing the dag table. |
| execution\_date | TIMESTAMP | The execution date of the task instance. |
| start\_date | TIMESTAMP | When the task instance started. |
| end\_date | TIMESTAMP | When the task instance finished. |
| duration | FLOAT | Time taken for the task to complete (in seconds). |
| state | VARCHAR(20) | State of the task instance (e.g., running, success, failed). |

### 4. ****log****

Stores logs related to the execution of tasks and DAGs.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for each log entry. |
| task\_instance\_id | INTEGER | Foreign key referencing the task\_instance table. |
| dag\_id | VARCHAR(250) | Foreign key referencing the dag table. |
| task\_id | VARCHAR(250) | Foreign key referencing the task in the task\_instance table. |
| execution\_date | TIMESTAMP | The execution date of the task being logged. |
| log | TEXT | The log output for the task instance. |

### 5. ****job****

This table keeps track of various jobs in Airflow, such as the Scheduler and the backfill jobs.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for the job. |
| dag\_id | VARCHAR(250) | Foreign key referencing the dag table. |
| job\_type | VARCHAR(30) | Type of job (e.g., scheduler, backfill). |
| start\_date | TIMESTAMP | When the job started. |
| end\_date | TIMESTAMP | When the job ended. |
| state | VARCHAR(20) | The state of the job (e.g., running, success, failed). |

### 6. ****xcom****

This table stores "cross-communication" data, which allows tasks to share small amounts of information during DAG execution.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for the XCom entry. |
| key | VARCHAR(250) | Unique key for the XCom entry. |
| value | BYTEA | The actual value being passed between tasks (serialized). |
| task\_id | VARCHAR(250) | Foreign key referencing the task instance that created the XCom. |
| dag\_id | VARCHAR(250) | Foreign key referencing the DAG. |
| execution\_date | TIMESTAMP | The execution date of the task instance. |

### 7. ****variable****

This table stores global variables that are accessible across all DAGs.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for each variable entry. |
| key | VARCHAR(250) | The unique key identifying the variable. |
| val | TEXT | The value of the variable. |

### 8. ****connection****

This table stores connection information, such as database credentials, APIs, and external services required for DAG execution.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for each connection. |
| conn\_id | VARCHAR(250) | The unique ID for the connection. |
| conn\_type | VARCHAR(500) | Type of connection (e.g., Postgres, MySQL, S3, etc.). |
| host | VARCHAR(500) | The host for the connection. |
| login | VARCHAR(500) | The username for the connection (if applicable). |
| password | VARCHAR(500) | The password for the connection (if applicable). |
| schema | VARCHAR(500) | The database schema for the connection (if applicable). |
| port | INTEGER | The port for the connection (if applicable). |

### 9. ****sla\_miss****

This table records Service Level Agreement (SLA) misses.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for each SLA miss entry. |
| dag\_id | VARCHAR(250) | Foreign key referencing the dag table. |
| task\_id | VARCHAR(250) | Foreign key referencing the task that missed its SLA. |
| execution\_date | TIMESTAMP | The execution date of the task that missed the SLA. |
| timestamp | TIMESTAMP | The time when the SLA was missed. |

### 10. ****pool****

This table manages resource pools, which allow users to limit parallel execution by task type.

| **Column** | **Type** | **Description** |
| --- | --- | --- |
| id | INTEGER | Primary key for the pool entry. |
| pool | VARCHAR(250) | The name of the resource pool. |
| slots | INTEGER | Number of available slots in the pool. |
| description | VARCHAR(500) | Description of the pool’s purpose. |

4. Executor

- The Executor is responsible for running the tasks in the DAGs. There are several types of executors:  
 - LocalExecutor: Runs tasks on the same machine where Airflow is running.  
 - CeleryExecutor: Distributes tasks across multiple worker nodes.  
 - KubernetesExecutor: Dynamically creates Kubernetes Pods to run tasks.

5. Worker Nodes

- Worker nodes (or workers) are responsible for executing tasks. When tasks are ready to be executed, they are dispatched to worker nodes, which can be local processes or distributed workers in a Kubernetes cluster or Celery.

# 2. Integrating Apache Spark with Airflow

## Explanation of Spark Integration

To execute tasks using Apache Spark, you can leverage the SparkSubmitOperator in Airflow. Instead of using Airflow workers to process data, the SparkSubmitOperator submits tasks to an external Spark cluster, which then handles the data processing.

## Required Configuration Changes for Spark Integration

1. Airflow DAG

- The SparkSubmitOperator needs to be used in the DAG to submit jobs to the Spark cluster. Here’s an example of a Spark job defined in a DAG:

from airflow import DAG  
from airflow.providers.apache.spark.operators.spark\_submit import SparkSubmitOperator  
from datetime import datetime  
  
default\_args = {  
 'start\_date': datetime(2023, 9, 1),  
 'catchup': False,  
}  
  
dag = DAG(  
 'spark\_job\_dag',  
 default\_args=default\_args,  
 schedule\_interval='@daily',  
)  
  
spark\_task = SparkSubmitOperator(  
 task\_id='spark\_submit\_job',  
 application='local:///opt/spark/examples/jars/spark-examples\_2.11-2.4.5.jar',  
 conn\_id='spark\_default',  
 verbose=True,  
 conf={  
 'spark.executor.memory': '4g',  
 'spark.executor.instances': '2',  
 'spark.master': 'spark://<your-spark-master-url>',  
 },  
 dag=dag,  
)  
  
spark\_task

2. Airflow Connection

- Set up a connection in Airflow for Spark (spark\_default). This can be done via the Airflow UI under Admin > Connections or in the values.yaml file during deployment.  
  
Example:  
connections:  
 - id: spark\_default  
 type: spark  
 host: spark://<your-spark-master-url>  
 port: 7077

# 3. Deploying Airflow in Kubernetes using Helm

Step-by-Step Procedure for Airflow Deployment

## 1. Add the Apache Airflow Helm Repository

helm repo add apache-airflow https://airflow.apache.org  
helm repo update

## 2. Create a Namespace for Airflow

kubectl create namespace airflow

## 3. Deploy Airflow using Helm

helm install airflow apache-airflow/airflow --namespace airflow

## Required Configuration Changes for Spark Integration

1. Configure the KubernetesExecutor  
  
Modify the values.yaml to use KubernetesExecutor for dynamic scaling of worker nodes in Kubernetes:  
yaml  
executor: KubernetesExecutor  
config:  
 AIRFLOW\_\_CORE\_\_EXECUTOR: KubernetesExecutor  
 AIRFLOW\_\_KUBERNETES\_\_NAMESPACE: "airflow"  
 AIRFLOW\_\_KUBERNETES\_\_WORKER\_CONTAINER\_REPOSITORY: "apache/airflow"  
 AIRFLOW\_\_KUBERNETES\_\_WORKER\_CONTAINER\_TAG: "2.0.0"  
 AIRFLOW\_\_KUBERNETES\_\_WORKER\_SERVICE\_ACCOUNT\_NAME: "airflow-worker"

2. Spark Cluster Connection  
  
Add the Spark cluster connection (spark\_default) to the Helm chart configuration:  
yaml  
connections:  
 - id: spark\_default  
 type: spark  
 host: spark://<your-spark-master-url>  
 port: 7077

3. Deploy Airflow with Custom Configurations  
  
Apply the custom configurations during Helm deployment:  
  
helm upgrade airflow apache-airflow/airflow --namespace airflow -f values.yaml

4. Expose Airflow Web UI  
  
Configure Airflow Web Server as a LoadBalancer in values.yaml to expose the UI externally:  
yaml  
web:  
 service:  
 type: LoadBalancer