November 6, 2025

Apache Airflow Workflow Orchestration

What is Airflow?

Definition of Airflow

Apache Airflow is an open-source tool designed to programmatically author, schedule, and monitor workflows as directed acyclic graphs (DAGs).

Purpose and Use Cases

Airflow automates complex data pipelines, enabling reliable execution, scheduling, and monitoring of workflows in data engineering, ETL, machine learning, and DevOps.

Why Airflow for Workflow Orchestration

Airflow offers dynamic pipeline generation, extensibility with custom operators, robust scheduling, and a rich UI for monitoring and managing workflows.

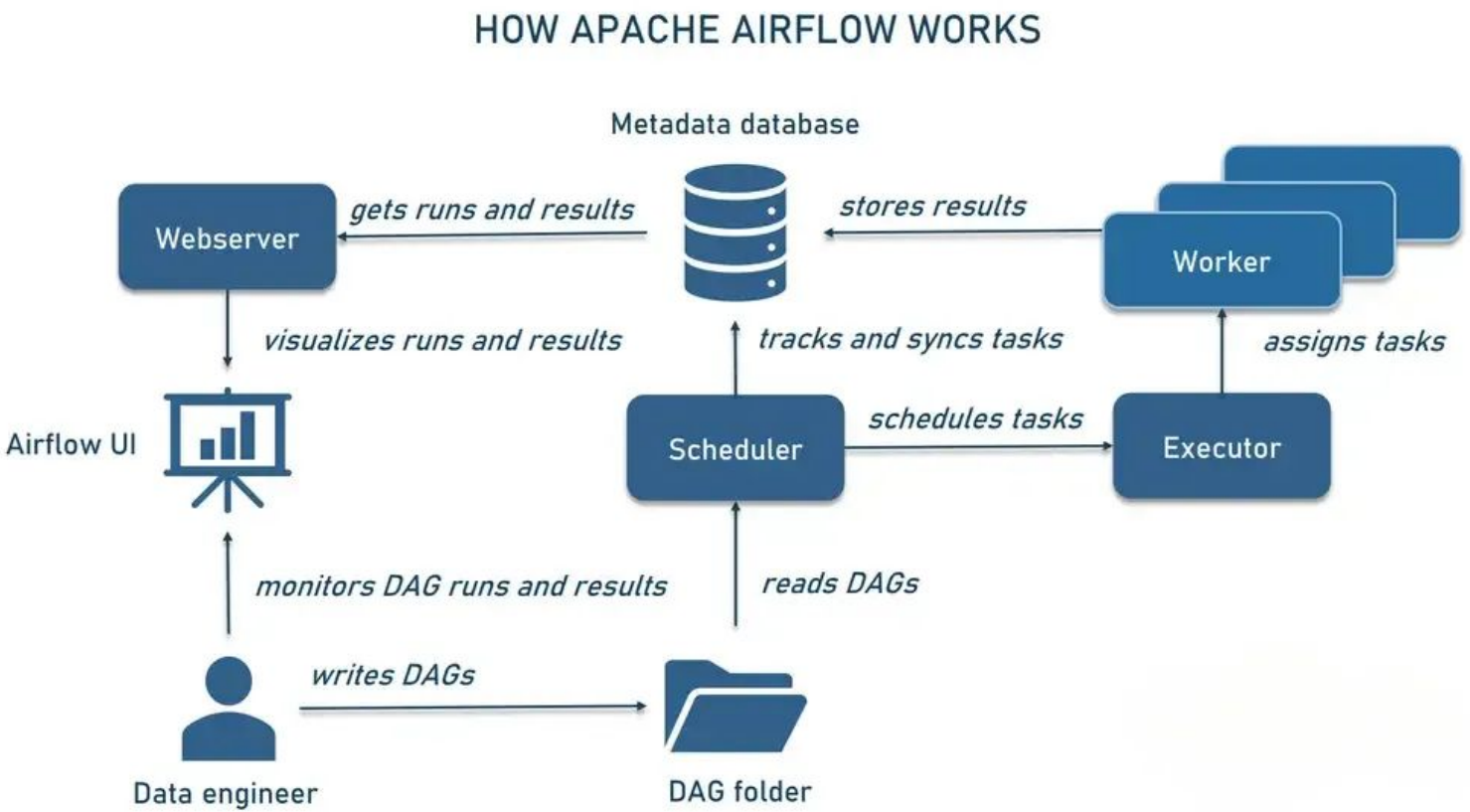
Key Features

Features include DAG-based workflow management, modular architecture, rich user interface, extensive operator support, and integration with many platforms.

Common Use Cases

Used for ETL processes, data warehousing, machine learning workflows, infrastructure automation, and batch job orchestration in various industries.

Airflow Architecture & Components



Airflow Architecture & Components

Scheduler

The Scheduler triggers tasks based on defined DAG schedules, deciding when and what tasks to run next to ensure timely workflow execution.

Executor

The Executor handles task execution by distributing work to workers; it supports different modes like Local, Sequential, Celery, and Kubernetes Executors.

Worker

Workers are the machines or processes that perform the actual task execution assigned by the Executor, running task instances in parallel.

Web Server

The Webserver hosts the Airflow UI, allowing users to monitor DAGs, visualize workflows, trigger tasks, and view logs through a graphical interface.

Metadata Database

Stores all Airflow metadata including DAG definitions, task states, and scheduling information, serving as the single source of truth.

DAGs and Task Lifecycle

DAGs Concept

A Directed Acyclic Graph (DAG) represents a workflow as a collection of tasks with dependencies, ensuring tasks execute in a defined order without cycles.

Task Lifecycle Stages

Tasks transition through states like **queued**, **running**, **success**, **failed**, and **skipped**, reflecting their progress and outcomes during execution.

Relationship Between Tasks and DAGs

Tasks are the nodes within a DAG; the DAG defines how tasks depend on each other and their execution sequence.

DAG Structure Overview

A DAG is defined in Python code, specifying tasks, dependencies, scheduling intervals, and other execution parameters.

How Tasks Are Executed

The scheduler triggers tasks based on dependencies and schedule, the executor runs the tasks, and workers process them in parallel or sequentially.

Airflow Installation & Prerequisites

Prerequisites for Airflow

Ensure **Python 3.7+** is installed along with pip. Additional system dependencies like database drivers and build tools may be required depending on Airflow features.

Installing via pip

Run '**pip install apache-airflow**' with optional extras like 'postgres', 'celery' according to your setup. It offers flexibility for different backend integrations.

Installing via conda

Use '**conda install -c conda-forge airflow**' for an isolated environment. Conda handles dependencies well, making it ideal for data science workflows.

Installing via Docker

Use official Airflow Docker images to avoid complex setups. Docker ensures consistency across environments and simplifies scaling and deployment.

Config File Overview (airflow.cfg)

The **airflow.cfg** file contains key configurations like **executor type, database connection, logging,** and **scheduler settings.** Customize it to optimize Airflow behavior for your environment.

Airflow File System Overview

DAGs Folder

This folder stores all Directed Acyclic Graph (DAG) definitions as Python scripts, which Airflow scans to schedule and execute workflows.

Logs Folder

Contains execution logs for tasks and **DAG runs**, helping users **monitor progress, debug issues**, and **audit workflow** performance.

Plugins

Custom **operators**, **sensors**, **hooks**, and **interfaces** are placed here to extend Airflow's capabilities beyond its built-in functions.

Config Files

Includes the airflow.cfg configuration file that defines system parameters, environment settings, and executor options for Airflow.

Airflow User Interface Overview

DAG Dashboard

Central hub displaying all DAGs with their **status**, **schedule**, and **recent run details** for quick overview and access.

Graph View

Visual representation of DAG tasks and dependencies as a directed acyclic graph, helping users understand execution flow.

Tree View

Hierarchical view showing task status over time, useful for tracking progress and identifying failed or skipped tasks.

Task Monitoring

Real-time updates on individual task execution with logs, duration, and status to facilitate troubleshooting and performance tracking.

Triggering DAGs

Manual and automated options to start DAG runs directly from the UI, enabling control over workflow execution timing.

Common Airflow CLI Commands

airflow db init

Initializes the **Airflow metadata database,** creating all necessary tables and structures for the system to operate.

airflow scheduler

Starts the scheduler process that triggers task execution following the defined DAG schedules and dependencies.

airflow webserver

Launches the Airflow web interface, providing a **dashboard** to **monitor**, **manage**, and **trigger DAGs visually.**

airflow dags list

Lists all DAGs currently available in the Airflow environment, allowing users to see which workflows are defined.

airflow tasks run

Manually triggers the execution of a specific task within a DAG, useful for testing or rerunning tasks on demand.

Types of Airflow Operators

PythonOperator

Executes Python functions or callables within a DAG, enabling integration of complex logic and data processing directly in workflows.

BashOperator

Runs bash commands or shell scripts, allowing system-level command execution as part of the workflow automation.

DummyOperator

Acts as a placeholder or no-op task, useful for structuring DAGs or marking task dependencies without performing actions.

SensorOperator

Waits for a specific condition or external event to be met before proceeding, such as file availability or database updates.

Custom Operators

Users can create their own operators by extending base classes to handle unique or complex tasks tailored to specific workflows.

Airflow Executors Explained

LocalExecutor

Runs tasks in parallel on the same machine using multiprocessing, suitable for moderate workloads and local setups.

SequentialExecutor

Executes tasks one at a time sequentially, ideal for development and testing but not for production use due to limited concurrency.

CeleryExecutor

Distributes task execution across multiple worker nodes using Celery and message brokers, enabling scalable and distributed workflows.

KubernetesExecutor

Launches each task in its own Kubernetes pod, providing dynamic scaling, isolation, and cloud-native orchestration benefits.

Data Passing and Variables in Airflow

What are XComs?

XComs (Cross-communications) are a built-in feature that allows tasks to exchange small pieces of data asynchronously during DAG execution.

Passing Data Between Tasks

Tasks use XComs to push and pull data, enabling downstream tasks to access outputs from upstream tasks within the same DAG run.

Airflow Variables

Variables are **key-value pairs** stored centrally in Airflow's **metadata** database, used for storing configuration or constant values accessible by all tasks.

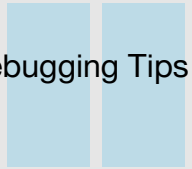
Example with PythonOperator

A PythonOperator can push data to XComs using 'xcom_push' and pull it in downstream tasks with 'xcom_pull', facilitating dynamic data-driven workflows.

Best Practices & Debugging Tips

Best Practices in DAG Design

Design DAGs with clear task dependencies and modular, reusable tasks. Use meaningful task IDs, avoid hardcoding values, and leverage Airflow variables and connections for flexibility.



Debugging Tips

Monitor logs and task states via the Airflow UI. Use verbose logging for detailed error analysis. Test tasks independently and apply retries and alerts for failure handling.

Real-world Case Studies

Examples include automating ETL pipelines, orchestrating machine learning workflows, and managing daily batch jobs, which demonstrate improved reliability and scalability with Airflow.

Thank you