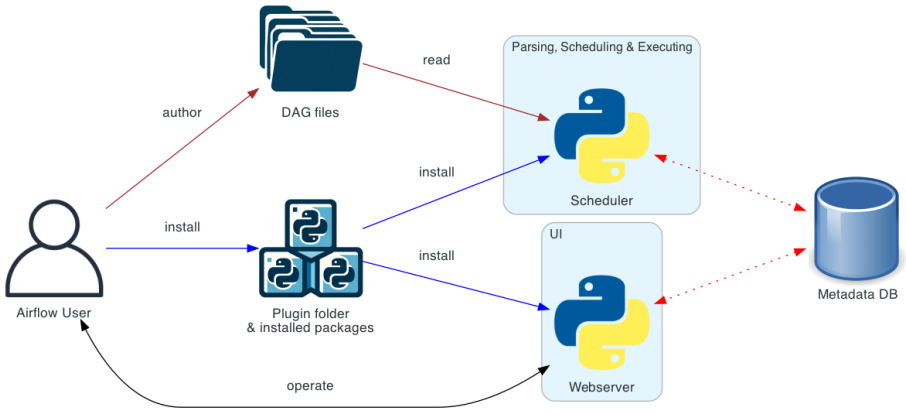
Introduction to Apache Airflow

What is Apache Airflow?

- Airflow is an open-source platform for programmatically authoring, scheduling, and monitoring workflows. It's designed to manage complex data pipelines.

- Key benefit: Enables you to define workflows as code, making them versionable, testable, and collaborative.



Core Concepts Overview:

- Main components: DAGs, Tasks, Operators, Scheduler, and Executor.

- The Scheduler triggers DAGs, which are composed of Tasks, which use Operators to perform work, and the Executor runs those tasks.

Use Cases:

- Data Engineering: ETL pipelines, data warehousing.
- Machine Learning: Training model, deploying model.
- Business Intelligence: Generating reports, dashboards.
- General Automation: Any process that can be defined as a workflow.

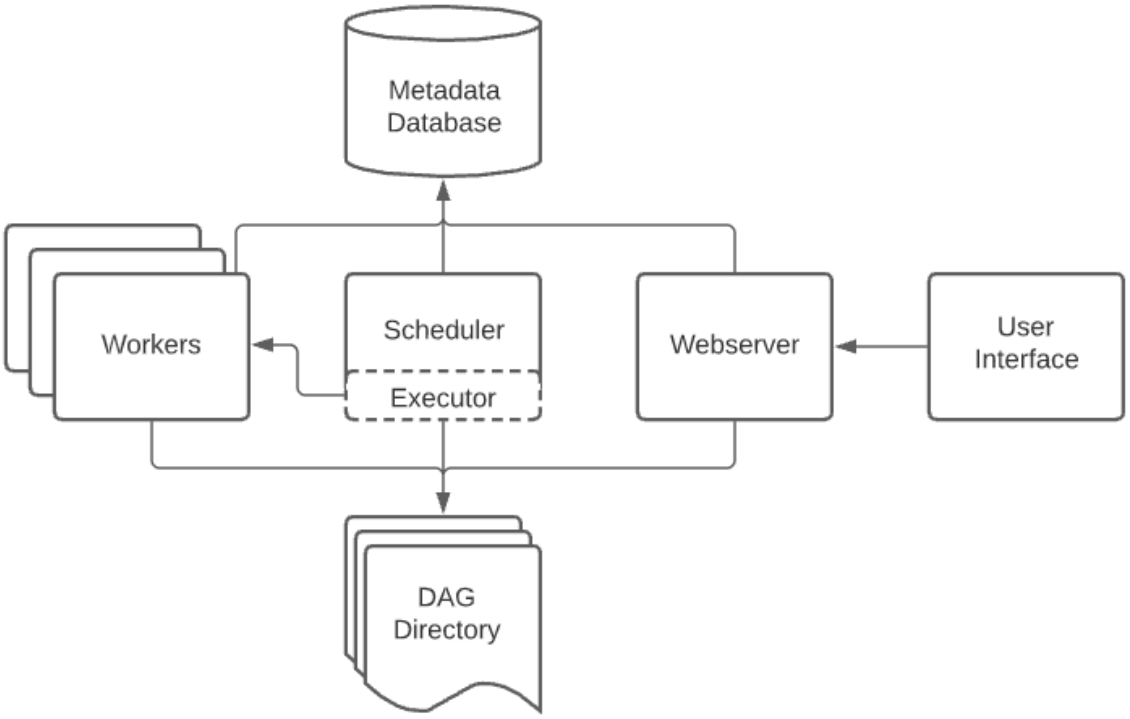
DAGs (Directed Acyclic Graphs)

- A DAG represents a workflow, a collection of tasks you want to run, organized in a way that reflects their dependencies.

- "Directed" means tasks run in a specific order. "Acyclic" means there are no loops (a task cannot depend on itself, directly or indirectly).

DAG Structure:

- DAGs are defined in Python code.




- ```python

  from airflow import DAG

  from airflow.operators.bash import BashOperator

  from datetime import datetime

  with DAG(

    dag_id='my_first_dag',

    start_date=datetime(2023, 1, 1),

    schedule_interval='@daily',

```
    catchup=False
) as dag:
    task1 = BashOperator(
        task_id='print_date',
        bash_command='date'
    )
```

DAG Runs:

- An instance of a DAG execution.
- Each DAG Run has a state (e.g., running, success, failed).
- Airflow automatically manages DAG Runs based on the schedule.

Tasks:

- A Task represents a single unit of work within a DAG.
- It's an instantiation of an Operator.

Task Attributes:

- `task_id`: A unique identifier for the task within the DAG.
- `dag`: The DAG to which the task belongs.
- `depends_on_past`: Whether the task should depend on the successful completion of the previous task instance.
- `retries`: Number of times to retry the task if it fails.

Task Dependencies:

- Tasks can depend on each other, creating a flow of execution.
- Dependencies are defined using `set_upstream()` or `set_downstream()` methods, or using bitshift operators (`>>` and `<<`).
- Example: `task1 >> task2 >> task3` (task1 runs before task2, which runs before task3).

Operators:

- Operators are pre-built or custom components that define what a task *does*. They are the building blocks of workflows.

- They encapsulate the logic to perform a specific action.

Operator Types:

- BashOperator: Executes a bash command.
- PythonOperator: Executes a Python callable.
- EmailOperator: Sends an email.
- PostgresOperator/MySqlOperator/etc.: Executes SQL queries on a database.
- Sensor: Waits for a certain condition to be met.

- Example (PythonOperator):

```
from airflow.operators.python import PythonOperator
def my_python_function():
    print("Hello from Airflow!")
with DAG(
    dag_id='python_operator_example',
    start_date=datetime(2023, 1, 1),
    schedule_interval=None
) as dag:
    python_task = PythonOperator(
        task_id='my_python_task',
        python_callable=my_python_function
    )
```

Scheduler:

- The Scheduler is responsible for triggering DAG runs based on their defined schedules.

- It monitors DAGs and tasks, and submits tasks to the Executor when their dependencies are met.

Scheduling Logic:

- The Scheduler parses DAG files and determines when DAGs should be run.
- It uses the `schedule_interval` defined in the DAG to determine the next execution time.
- `schedule_interval` can be a cron expression (e.g., '0 0 * * *' for daily at midnight) or a predefined value like `@daily`, `@weekly`, `@monthly`.

Role in the Airflow Architecture:

- The Scheduler is a central component that orchestrates the entire workflow.
- It interacts with the metadata database to track DAG and task states.

Executor:

- The Executor is responsible for actually running the tasks. It takes tasks from the Scheduler and executes them.
- Different types of Executors are available, each with its own advantages and disadvantages.

Executor Types:

- SequentialExecutor: Executes tasks one at a time, in the same process as the Scheduler (suitable for testing).
- LocalExecutor: Executes tasks in parallel on the same machine as the Scheduler, using multiple processes.
- CeleryExecutor: Distributes tasks to worker nodes using Celery, enabling distributed execution. Requires a message queue (e.g., RabbitMQ, Redis).
- KubernetesExecutor: Creates a new pod in a Kubernetes cluster for each task, providing excellent isolation and scalability.

Executor Choice Considerations:

- The choice of Executor depends on the scale and complexity of your workflows.
- For small projects, the `LocalExecutor` may be sufficient.
- For large, production environments, the `CeleryExecutor` or `KubernetesExecutor` are recommended.