**Apache Airflow**

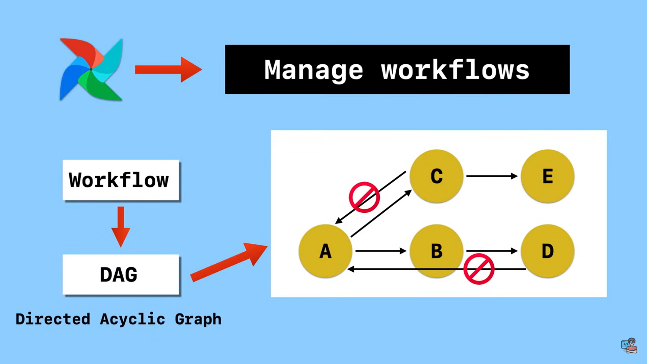
Apache Airflow is an **open-source platform** used to **programmatically authoring, scheduling, and monitoring workflows** or pipelines. First developed by Airbnb, it is now under the **Apache Software Foundation**.

It allows **users** to define **workflows as Directed Acyclic Graphs (DAGs)**, where nodes represent tasks, and edges represent dependencies between tasks

workflows and ETL processes are **closely related**.

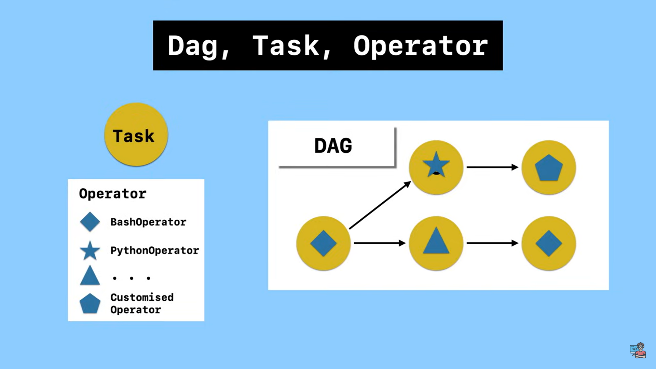
“Orchestrating” refers to the process of **co-ordinating and managing the tasks or cron jobs.**

**Airflow core concepts:**

****

Airflow is managing the **workflows**.

**Workflow** is **sequence of task**.In **airflow ,workflow** is defined as a **DAG**.



**Dag:**

**DAG** is **collection of all the tasks** you want to **run organized way** and **dependencies.**it is **represents as a node** in the DAG graph.and it’s **written in python.**

**Task:**

A task **defines a unit of work** within the DAG and there is a dependency between the task.

**Example:**

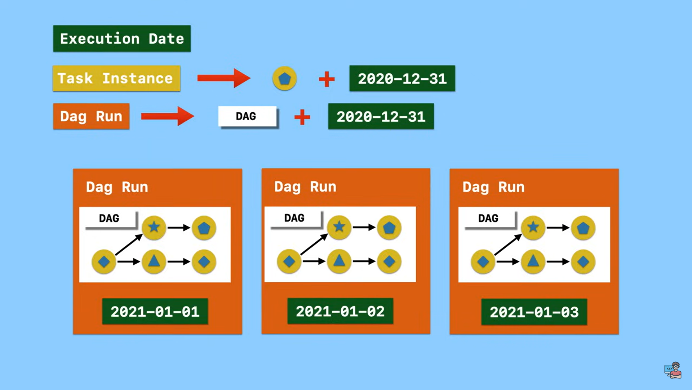
Task C is the downstream of task A

Task C is the upstream of task B

**Operators:**

Operators are **pre-defined functions** predefined Task or classes in Airflow that provide specific functionalities for common tasks.

**Execuion date,task instance,dag run:**

****

**Execution Date:** The **specific date and time** a DAG Run is triggered to execute.

**Task Instance:** An **individual execution of a task** within a DAG Run. It's like a single step within a specific workflow instance, **capturing its status** (success, failure, etc.) for that particular run.

A task instance is a **run of task at a specific point of time** (execute\_date)

capturing its status (success, failure, etc.) for that particular run.

**DAG Run:** An entire execution of a DAG at a specific point in time

**Task lifecycle:**

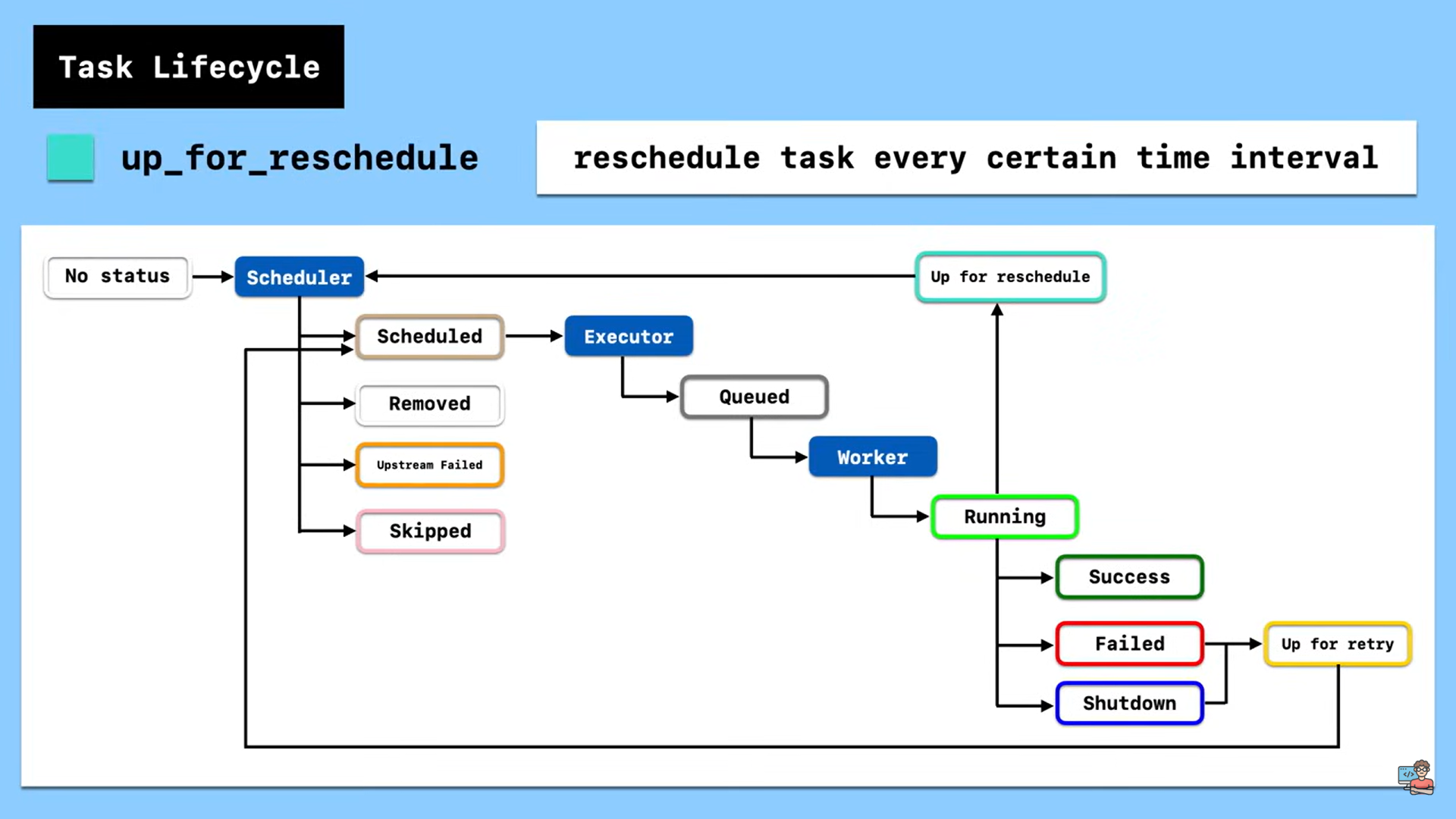
When a **DAG run is triggered** .its **tasks are going to be executed** one after another **according to the dependencies**.

Each task will go through different stages from start to completion.every stage indicates a specific status of the task instances

For Example:

In progress -> status is running

Task finished -> success status



**No\_status :** Task usually **starting with no status**,which means the scheduler create a **empty task instance**.

**Scheduler :** Scheduler determined task instance needs to run.

**Upstream\_failed :** the task **upstream failed**

**Skipped :**  the **task is skipped or removed**

**Executer:** the task is **successfully scheduled means the executer will kicks** to the **task queue**

**Queued :** scheduler sent task to the executer to run on the queue,

**Worker :** after that the worker will execute the task once it is free (free the work computation not fully occupied)at this stage the task **status is running**.

**Running :** worker picked up a task and is now running it

**3 possible stages:**

**Success :** task completed success without error

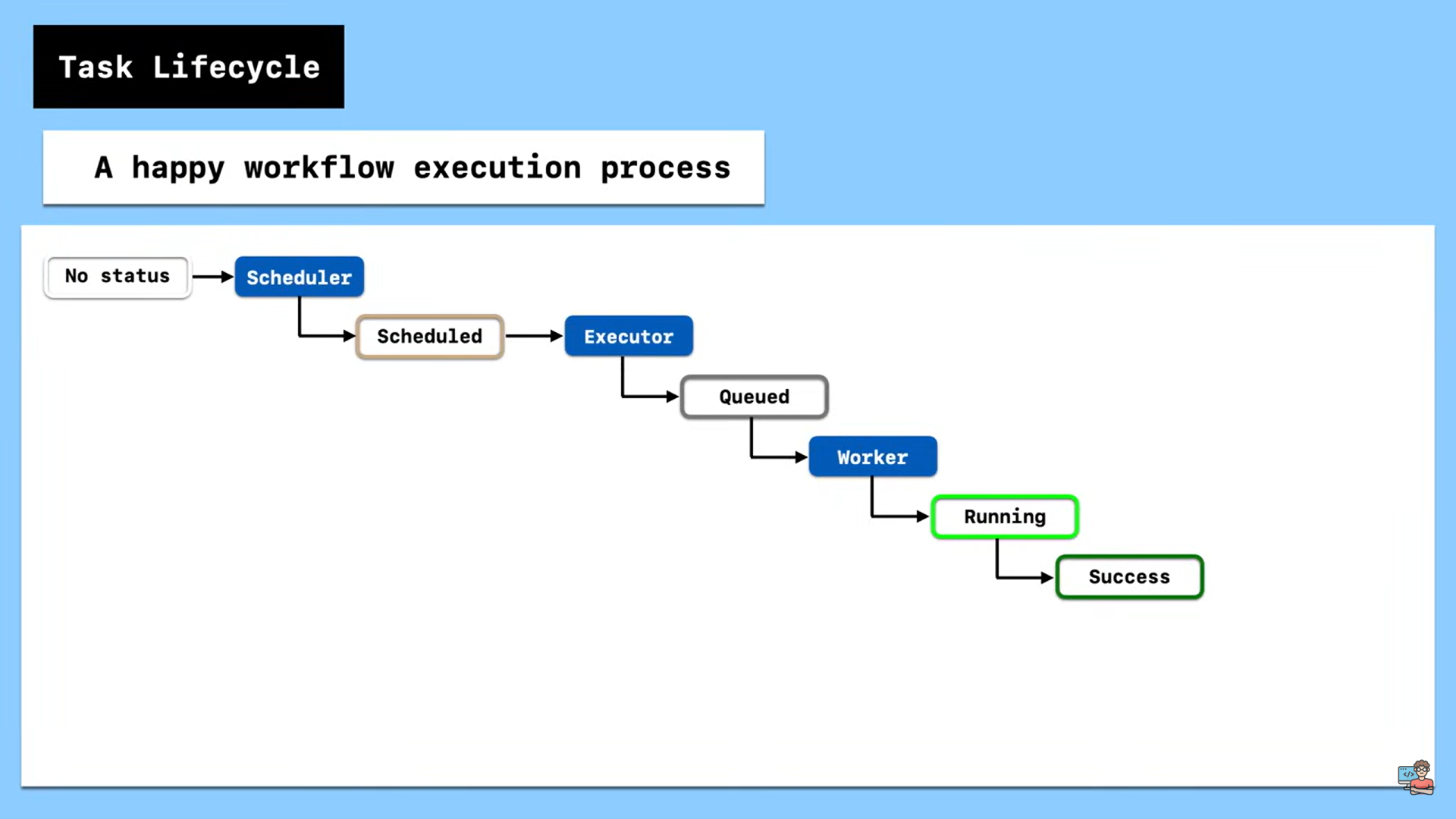
**Faild :** the task failed.

**Shutdown :** the task has been shutdown

**Up\_for\_retry :** rerun to the task

If the task is failed and shutdown the maximum retry is not exceeded the task is forwarded to the **retry stage.**

**A happy workflow process:**



**Usage of apache airflow:**

**Data Pipeline Orchestration**: It manages complex data pipelines that involve multiple steps and **dependencies**. For example, Airflow can automate tasks like:

* Downloading data from various sources.
* Performing data transformations (cleaning, filtering, etc.).
* Loading data into data warehouses or databases.
* Triggering machine learning models or data analysis jobs

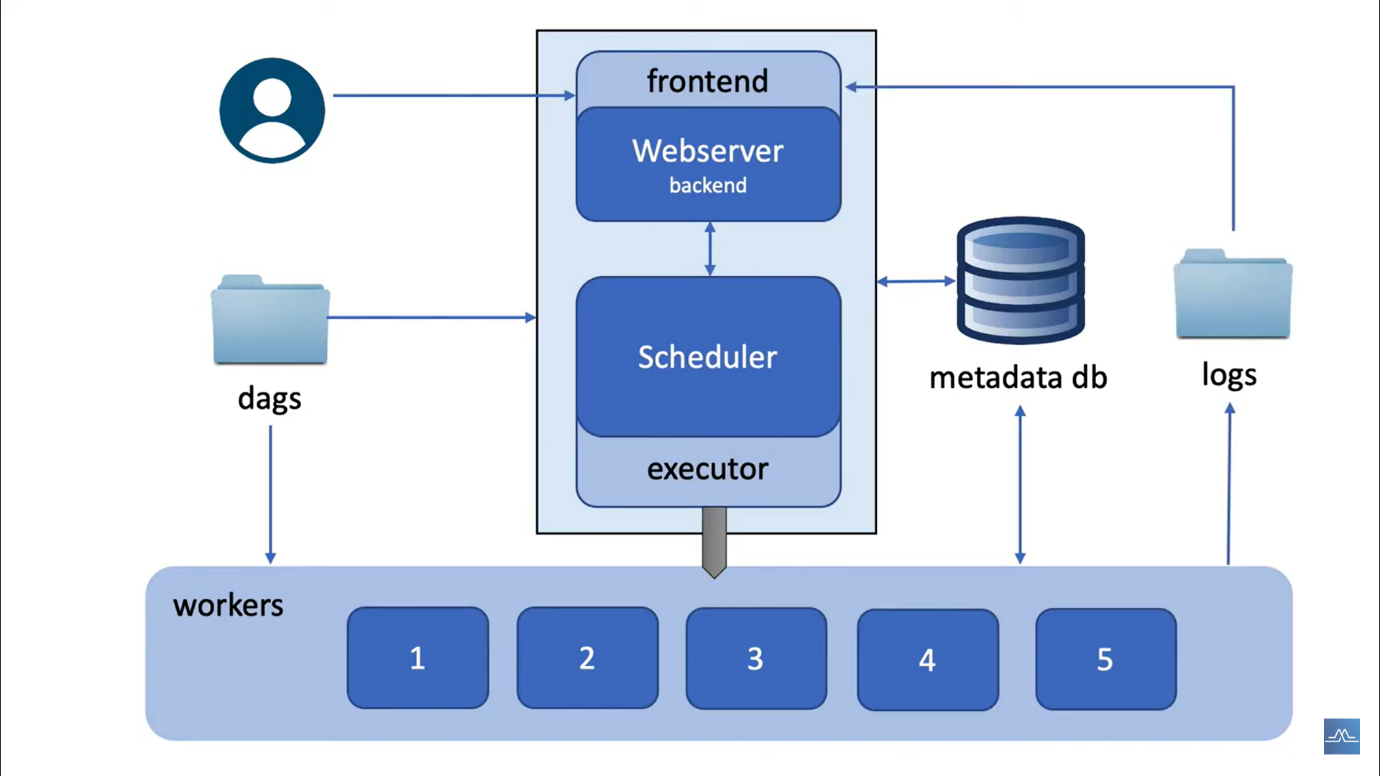
**ETL:**

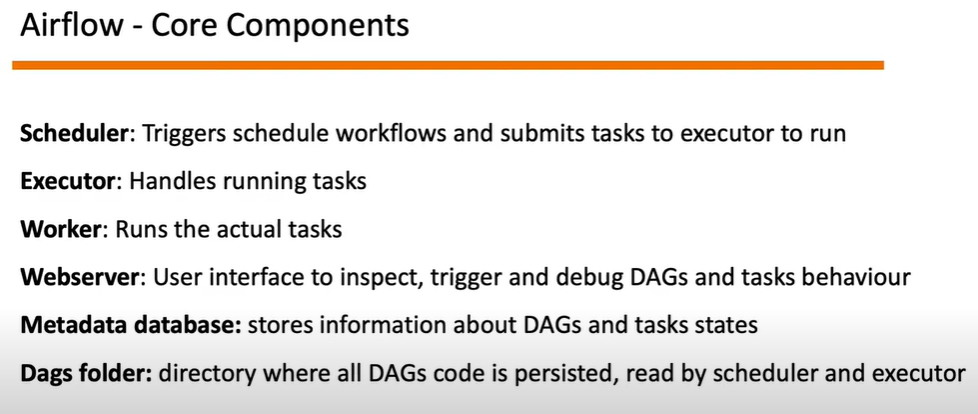
ETL (Extract, Transform, Load) Processes: Airflow **can automate** the **entire ETL process**, ensuring data is extracted from various sources, transformed as needed, and loaded into target/destination location.

**Advantanges of airflow:**

Airflow allows you to define workflows(ETL) using Python

**Architecture of Airflow:**





**how changes to your DAG file will be reflected in the Airflow webserver:**

**volumes:**

**- ${AIRFLOW\_PROJ\_DIR:-.}/dags:/opt/airflow/dags:** This **mounts the local directory** containing your DAG files (usually within your PyCharm project's dags folder) to **the /opt/airflow/dags directory inside the Airflow container**. Any changes you make to your DAG files in the local directory will be automatically reflected in the container's /opt/airflow/dags directory due to this volume mapping.

**Mounts the local directory(dags) to airflow container directory (opt/airflow/dags)**

**How Changes Reflect in Airflow Webserver:**

**File System Changes Detected**: When you modify a DAG file in your local dagsdirectory, the **Docker volume mechanism detects these changes**.

**Updated Files Mounted into Container**: The **updated DAG files** are **automatically copied into the /opt/airflow/dags** directory within the Airflow container.

**Volume:**

In Docker, a volume is a **mechanism** for **persisting data that is generated or used by containers.**

Persistence:

Data stored within a volume is independent of the container's lifecycle. Even if the container is stopped, restarted, or recreated, the data in the volume remains intact. This allows you to maintain application state or manage data that needs to be preserved across container lifecycles.

**DAG:**

**DAG is blueprints for your workflows.**

DAGs represent the **overall workflow** or pipeline in Airflow. A **DAG is a collection of tasks** (nodes) and the **dependencies** (edges) between them. DAGs are defined using Python code and are directed, meaning tasks can only depend on tasks that precede them in the DAG.

A DAG is **acyclic**, meaning it **doesn't have loops** to prevent infinite execution.

**Task:**

Tasks represent **individual units of work** within a **workflow**. **Each task performs a specific action or operation**, such as running a script, executing a SQL query, or interacting with an external system.

**Operators:**

Operators are **pre-defined functions** predefined Task or classes in Airflow that provide specific functionalities for common tasks.

**common operator examples:**

**BashOperator**: Runs Bash shell commands.

**PythonOperator**: Executes a Python function.

Link: <https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/operators.html#operators>

<https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/bash/index.html>

**Executors:**

Executors are **responsible for executing task instances** within a DAG

**LocalExecutor**:

* This **runs tasks directly** on the machine where Airflow is installed, **suitable for small workflows**.
* typically used for development and testing.

**SequentialExecutor**:

Similar to LocalExecutor, but runs tasks in a **separate thread**.

**CeleryExecutor**:

* This uses a **distributed task queue** (Celery) to run tasks on **multiple worker machines**, ideal for **larger workflows**.
* Distributes tasks across a Celery message queue and executes them in parallel on multiple workers.

The executor fetches tasks from the scheduler, executes them, and reports back the results (success, failure, logs).