**Airflow**

Apache Airflow is an open-source platform designed to programmatically author, schedule, and monitor workflows.

It allows you to define workflows as code, making it easier to manage, schedule, and monitor complex data workflows.

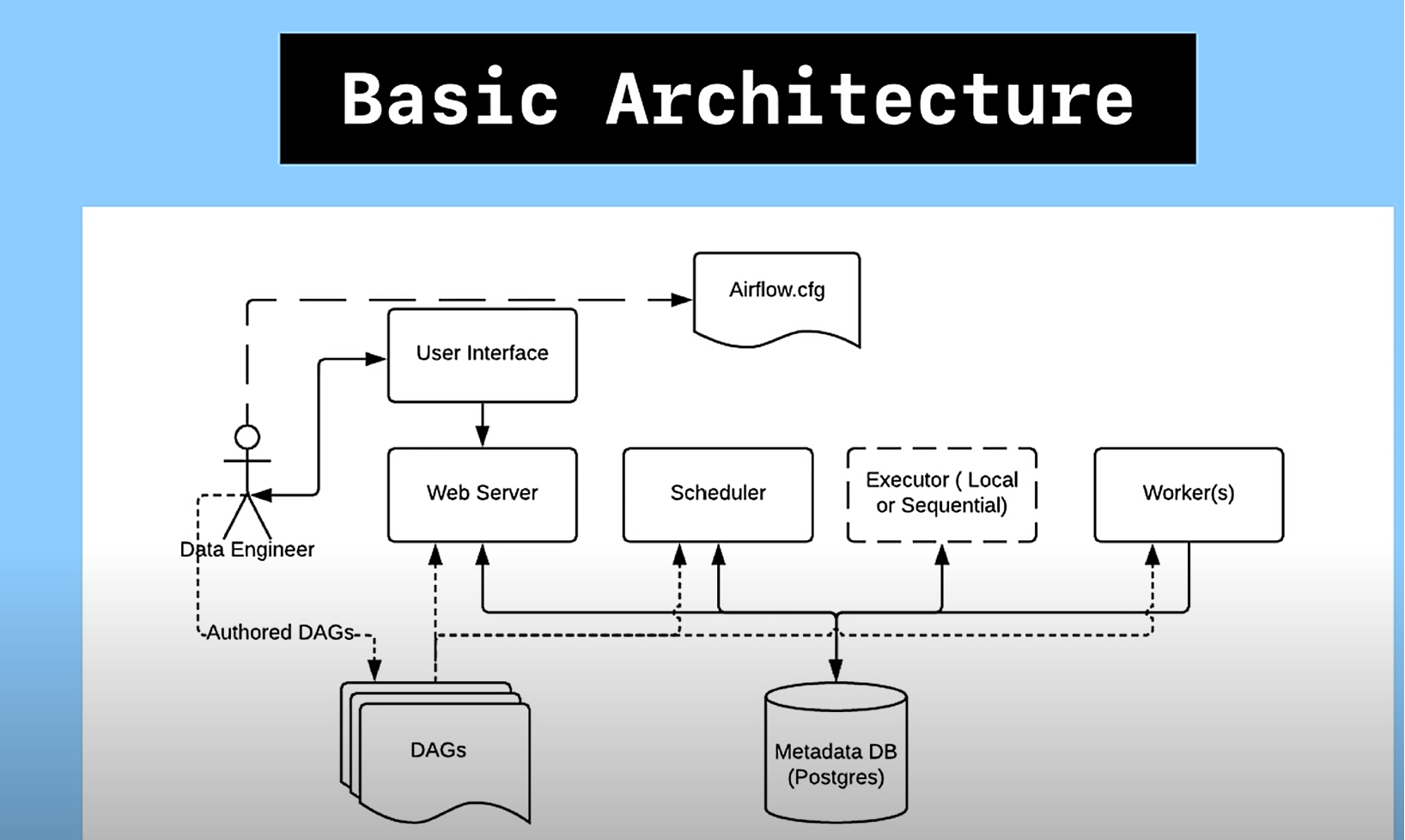
Airflow is particularly useful for orchestrating ETL (Extract, Transform, Load) processes, data pipeline workflows, and other automation tasks.

Key features of Apache Airflow include:

1. **DAGs (Directed Acyclic Graphs):** Workflows in Airflow are defined as Directed Acyclic Graphs, where nodes represent tasks and edges define the order in which tasks should be executed.
2. **Operators:** Tasks within a DAG are implemented as operators. Airflow provides a variety of built-in operators for common tasks (e.g., BashOperator, PythonOperator, SQLOperator), and you can also create custom operators.
3. **Scheduler:** Airflow includes a scheduler that can be configured to run tasks on a specified schedule. It ensures that tasks are executed at the right time and in the correct order.
4. **Web UI:** Airflow comes with a web-based user interface that provides a visual representation of DAGs, task status, and execution history. It allows users to monitor and troubleshoot workflows.
5. **Extensibility:** Airflow is extensible and can be integrated with various external systems and databases. This extensibility is useful for connecting to different data sources, triggering workflows based on events, and more.
6. **Parallel Execution:** Airflow allows for parallel execution of tasks, enabling the concurrent processing of tasks when possible, which can improve the overall efficiency of workflows.
7. **Connections and Hooks:** Airflow provides a way to define external connections (e.g., database connections, API keys) and hooks, allowing for better modularization and reuse of code.
8. **Logging and Monitoring:** Airflow logs task execution details, making it easier to troubleshoot issues. Additionally, it supports integration with external monitoring tools.

**Airflow Architecture:**

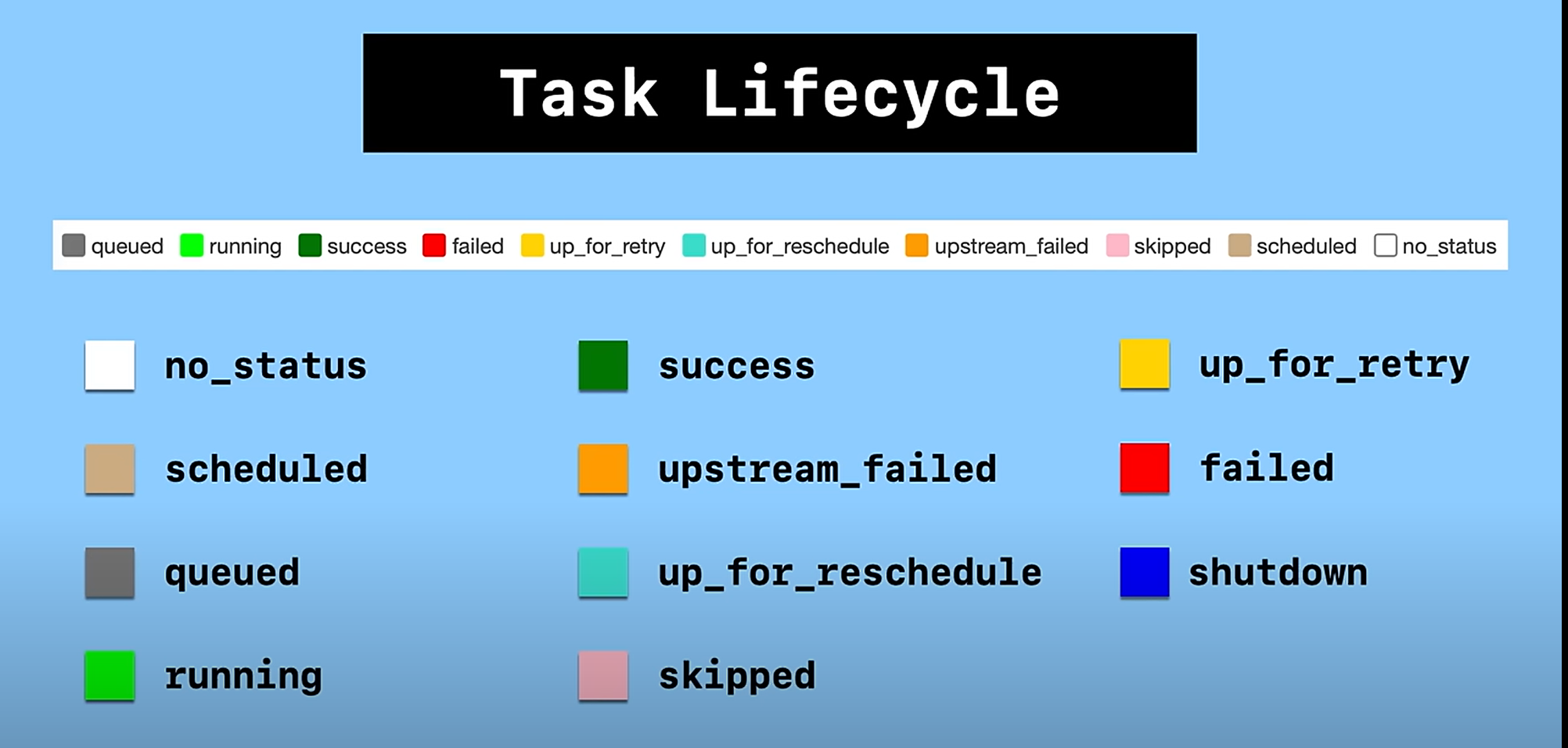
1. **Scheduler**: The scheduler orchestrates the execution of tasks in Airflow. It continuously polls the DAGs for tasks that need to be executed and schedules them based on their dependencies and defined execution times.
2. **Metadata Database**: Airflow uses a relational database to store metadata related to workflows, tasks, task instances, and execution logs. This database is critical for maintaining the state of workflows and tracking their execution history.
3. **Web Interface (UI)**: Airflow comes with a web-based user interface that allows users to visualize and monitor workflows, view execution logs, and manually trigger DAG runs. The UI is built using the Flask web framework and provides a user-friendly interface for interacting with Airflow.
4. **Executor**: The executor is responsible for executing individual tasks within a workflow. Airflow supports different executor types, including the SequentialExecutor (for development and testing), LocalExecutor (for executing tasks in parallel on a single machine), CeleryExecutor (for distributed task execution using Celery), and others.
5. **Scheduler Event Log**: Airflow uses a scheduler event log to track changes to the state of tasks and workflows. This log helps the scheduler make informed decisions about which tasks to execute next and ensures that workflows are executed reliably.
6. **Workers**: In distributed setups, workers are responsible for executing tasks assigned to them by the scheduler. Each worker pulls tasks from the scheduler's queue and executes them according to their dependencies and execution requirements.
7. **DAGs**: Directed Acyclic Graphs (DAGs) are the core abstraction in Airflow. A DAG represents a workflow as a collection of tasks and dependencies between them. Each task in a DAG is defined as a Python function or an instance of an Operator class, which encapsulates a unit of work.
8. **Plugins**: Airflow allows users to extend its functionality through plugins. Plugins can define custom operators, hooks, sensors, and other components that integrate with external systems or provide additional functionality to Airflow.

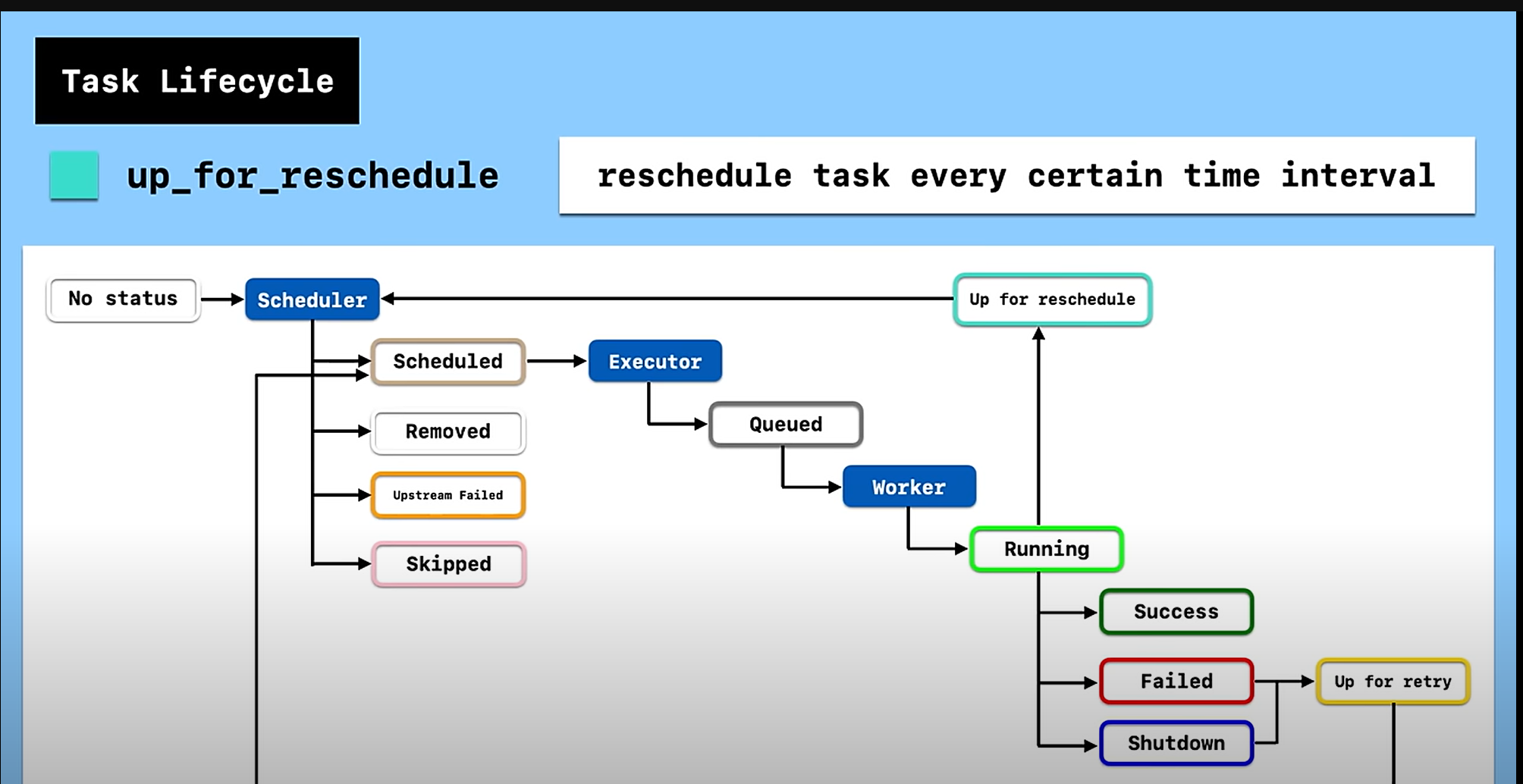


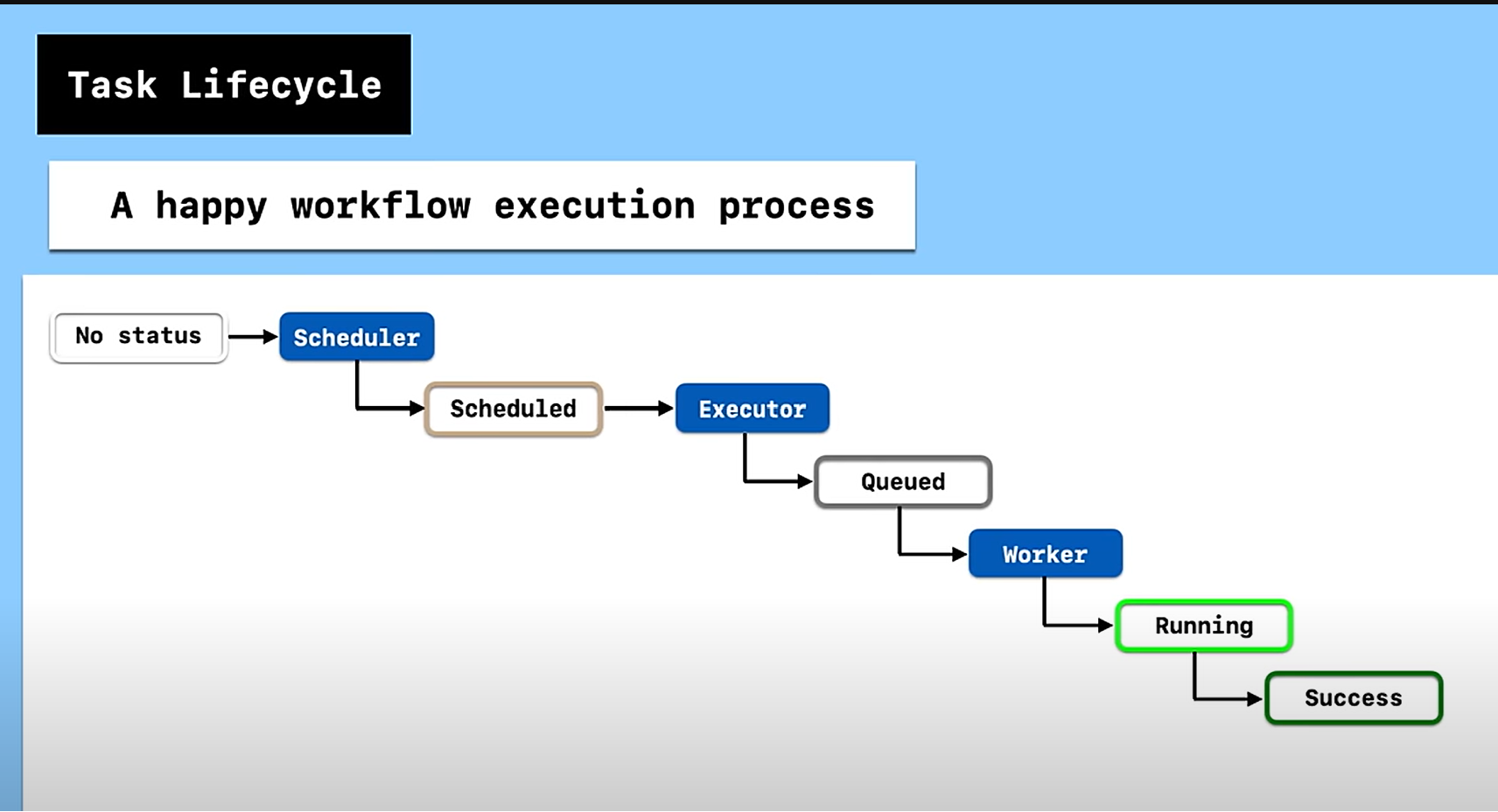
**Airflow execution steps:**

1. **DAG Parsing**: When you start the execution of a DAG, Airflow first parses the Python code defining the DAG. This involves loading the DAG definition file and identifying the tasks, dependencies, and other attributes specified in the DAG definition.
2. **Task Instantiation**: Once the DAG is parsed, Airflow instantiates each task defined in the DAG. Tasks are instantiated as instances of Operator classes or as PythonCallable objects, depending on how they are defined in the DAG definition.
3. **Task Scheduling**: After all tasks are instantiated, Airflow's scheduler determines the order in which tasks should be executed based on their dependencies. Tasks with no upstream dependencies (i.e., those without any tasks depending on them) are considered for execution first.
4. **Task Queuing**: The scheduler places tasks that are ready for execution into a queue. The executor, which is responsible for executing tasks, pulls tasks from this queue based on its configuration (e.g., parallelism settings).
5. **Task Execution**: The executor retrieves tasks from the queue and executes them according to their specifications. This involves running the code associated with each task, which could be a Python function, a shell command, or an interaction with an external system.
6. **Task Status Updates**: As tasks are executed, Airflow updates the status of each task instance in its metadata database. This includes tracking the start time, end time, duration, and status (success, failure, or retry) of each task execution.
7. **Dependency Management**: Airflow ensures that tasks are executed in the correct order based on their dependencies. Tasks with upstream dependencies are only executed after all of their upstream tasks have been completed successfully.
8. **Logging and Monitoring**: Throughout the execution of the DAG, Airflow captures logs and metrics related to task execution. These logs are stored in the metadata database and can be accessed through the Airflow web interface or command-line interface for monitoring and troubleshooting purposes.
9. **Completion and Cleanup**: Once all tasks in the DAG have been executed (or if the execution is aborted due to failure or manual intervention), Airflow marks the DAG run as complete in the metadata database. Any necessary cleanup tasks, such as closing connections or releasing resources, are also performed.

**Task Lifecycle:**







**Executor in airflow:**

1. **Sequential Executor**: The Sequential Executor executes tasks sequentially in a single process. It is primarily used for development and testing purposes because it doesn't support parallel execution of tasks. While simple and easy to set up, it's not suitable for production environments with large workflows or high concurrency requirements.
2. **Local Executor**: The Local Executor allows tasks to run in parallel on a single machine. It uses multiprocessing to execute tasks concurrently, making it suitable for small to medium-sized workflows where parallelism is required but without the complexity of distributed execution. It's a good choice for environments with limited resources or when setting up Airflow for testing purposes.
3. **Celery Executor**: The Celery Executor is used for distributed task execution across multiple worker nodes. It leverages Celery, a distributed task queue, to distribute tasks to worker processes or machines. This executor is suitable for production environments with high concurrency requirements or when deploying Airflow in a distributed architecture. It provides scalability and fault tolerance by allowing tasks to be executed across multiple nodes.
4. **Dask Executor**: The Dask Executor is an experimental executor introduced in Airflow 2.0 that integrates with Dask, a parallel computing library in Python. It allows tasks to be executed using Dask distributed schedulers, enabling scalable and distributed task execution similar to the Celery Executor. The Dask Executor is particularly useful for environments where Dask is already used for parallel computing tasks and workflows.
5. **Kubernetes Executor (KubernetesPodOperator)**: While not technically an executor in the traditional sense, the Kubernetes Executor allows Airflow to launch tasks as Kubernetes Pods. Each task runs within its own containerized environment, providing isolation and scalability. This executor is well-suited for Kubernetes-based deployments, cloud-native environments, or when integrating Airflow with Kubernetes for workload orchestration.

**Docker-compose.yaml file**

* airflow-scheduler - The [scheduler](https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/scheduler.html) monitors all tasks and DAGs, then triggers the task instances once their dependencies are complete.
* airflow-webserver - The webserver is available at http://localhost:8080.
* airflow-worker - The worker that executes the tasks given by the scheduler.
* airflow-triggers - The triggerer runs an event loop for deferrable tasks.
* airflow-init - The initialization service.
* postgres - The database.
* redis - [Redis](https://redis.io/) - is a broker that forwards messages from the scheduler to the worker.

Some directories in the container are mounted, which means that their contents are synchronized between your computer and the container.

* ./dags - you can put your DAG files here.
* ./logs - contains logs from task execution and scheduler.
* ./config - you can add a custom log parser or add airflow\_local\_settings.py to configure cluster policy.
* ./plugins - you can put your [custom plugins](https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/plugins.html) here.

**Default Arguments:**



**Instantiate a DAG:**



1. **'owner'**: Specifies the owner of the DAG, typically used for informational purposes.
2. **'depends\_on\_past'**: Indicates whether the task instances should be triggered based on the success status of the previous run's task instance. Setting it to **True** ensures that the current task instance runs only if the previous one succeeded.
3. **'start\_date'**: Specifies the timestamp when the DAG should start. In this case, it's set to **days\_ago(2)**, which means the DAG will start two days before the current date.
4. **'email'**: Specifies the email addresses to which Airflow should send notifications.
5. **'email\_on\_failure'**: Indicates whether to send email notifications when a task fails.
6. **'email\_on\_retry'**: Indicates whether to send email notifications when a task is retried.
7. **'retries'**: Specifies the number of retries that should be attempted before a task is marked as failed.
8. **'retry\_delay'**: Specifies the delay between retries. In this case, it's set to **timedelta(minutes=5)**, meaning that retries will occur with a delay of 5 minutes between each retry.
9. **'queue'**: Specifies the queue to which the task should be assigned for execution. This is useful when you have multiple queues configured in Airflow and you want to assign specific tasks to particular queues for better resource management.
10. **'pool'**: Specifies the pool to which the task should belong. Pools are used to limit the execution concurrency of tasks. Tasks belonging to the same pool will run concurrently based on the configured pool settings.
11. **'priority\_weight'**: Assigns a priority weight to the task. This is used when there are resource constraints, and tasks with higher priority weights are given precedence in execution.
12. **'end\_date'**: Specifies the timestamp after which the DAG should stop running. If not specified, the DAG will run indefinitely.
13. **'wait\_for\_downstream**': Indicates whether the DAG should wait for all downstream tasks to complete before marking itself as successfully completed. By default, this is set to False.
14. **'dag**': Specifies the DAG object to which the task belongs. This is typically set automatically when defining tasks within a DAG context and doesn't need to be explicitly specified in default\_args.
15. '**sla**': Specifies the Service Level Agreement (SLA) for the task. It represents the maximum allowed duration for the task to complete successfully. If the task exceeds this duration, Airflow will mark it as a SLA miss.
16. **'execution\_timeout'**: Specifies the maximum duration allowed for the task to execute. If the task exceeds this duration, it will be forcefully terminated.
17. **'on\_failure\_callback', 'on\_success\_callback', 'on\_retry\_callback', 'sla\_miss\_callback'**: These are callback functions that can be executed when the task fails, succeeds, retries, or misses its SLA, respectively. You can define custom Python functions to handle these events.
18. **'trigger\_rule'**: Specifies the condition that triggers the execution of the task. For example, 'all\_success' indicates that the task should run if all upstream tasks have succeeded. Other options include 'all\_failed', 'one\_failed', 'none\_failed', etc.

**BashOperator:**

In Apache Airflow, the **BashOperator** is a class used to execute bash commands as tasks within a workflow. It allows you to run arbitrary shell scripts or commands as part of your workflow.

1. **Task Execution**: It executes a single bash command or a series of bash commands as a task within an Airflow DAG (Directed Acyclic Graph). This means you can define a task that runs shell commands, scripts, or any other executable that can be invoked from the command line.
2. **Flexibility**: The **BashOperator** provides flexibility because it allows you to execute virtually any command that you can run from a terminal. This includes running Python scripts, shell scripts, executing system commands, running SQL scripts, etc.

t1 = BashOperator(  
 **task\_id**='print\_date',  
 **bash\_command**='date',  
 **dag**=dag,  
)  
  
t2 = BashOperator(  
 **task\_id**='sleep',  
 **bash\_command**='sleep 5',  
 **retries**=3,  
 **dag**=dag,  
)  
  
t4 = BashOperator(  
 **task\_id**='echo\_hello',  
 **bash\_command**='echo "Hello, Airflow!"',  
 **dag**=dag  
)

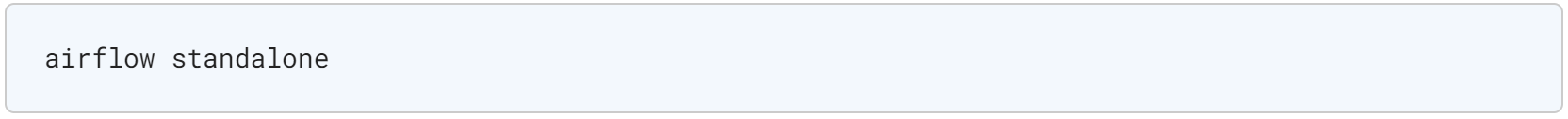
**Python Operator:**

**Data Sharing via Xcoms:**

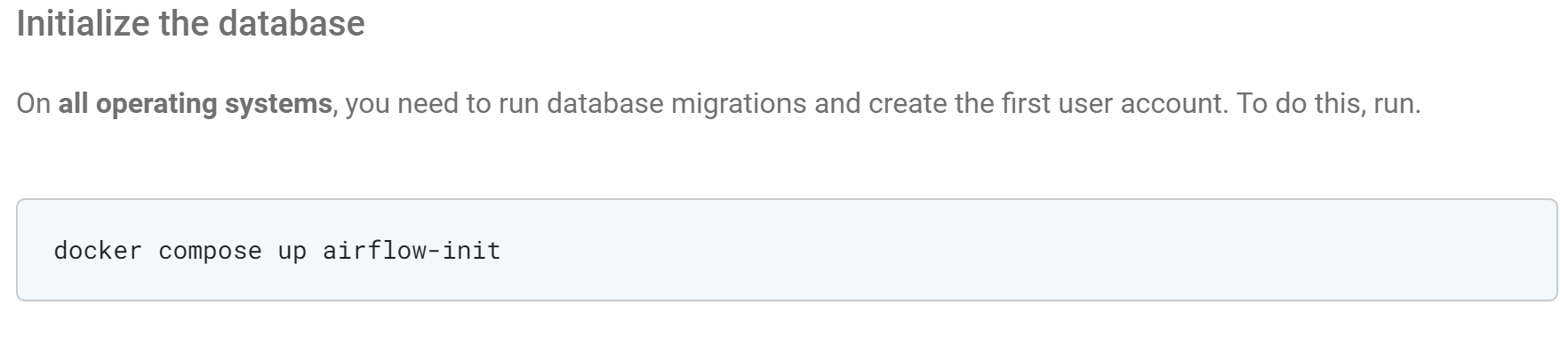
**TaskFlow API**

**Note:**

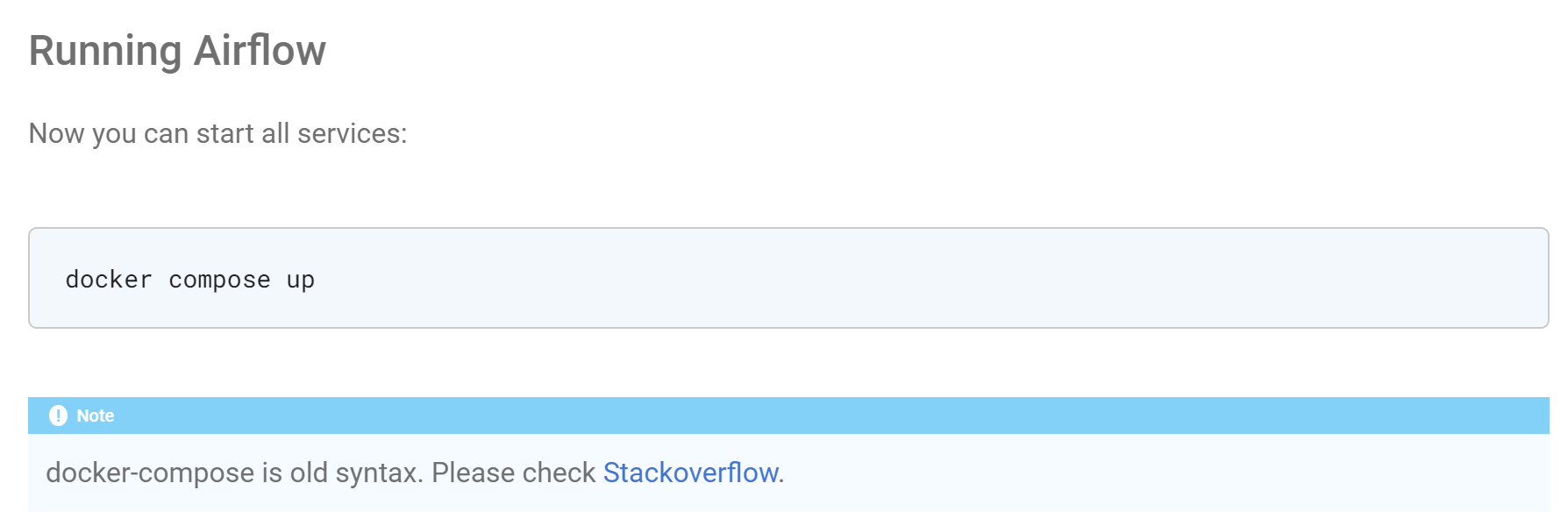
1. ***AIRFLOW\_\_CORE\_\_LOAD\_EXAMPLES: 'true'*** --- this will load the example DAGS in the Web UI
2. The airflow standalone command initializes the database, creates a user, and starts all components.



1. Set the executor using this command : ***AIRFLOW\_\_CORE\_\_EXECUTOR: CeleryExecutor***





****