Airflow

Airflow is an orchestrator, not a processing framework, process your gigabytes of data outside of Airflow (i.e. You have a Spark cluster, you use an operator to execute a Spark job, the data is processed in Spark).

// Airflow is an orchestrator allowing you to execute your tasks in the right time, right way at the right order.

**Benefits:**

-- Dynamic - Python

-- Highly scalable - using Kubernetes too

-- UI - Monitor DP, interact tasks

-- Extensible - No need to wait for inbuilt plugin, we can build/ customize our instance

**Core components:**

-- Web Server : Flask Server

-- Scheduler: Deamon incharge of scheduling DP (Heart of Airflow)

-- Metastore: DB usually Postgres, responsible to store metadata, tasks and so on

-- Executor: How tasks are executed - Using Kubernetes tasks, executes

-- Worker: The process/sub-process executes the tasks

An Executor defines how your tasks are execute whereas a worker is a process executing your task

The scheduler schedules your tasks, the web server serves the UI, the database stores the metadata of Airflow.

**Core Component:**

* DAG: The work (tasks), and the order in which work should take place (dependencies), written in Python.
* DAG Run: An instance of a DAG for a particular logical date and time.
* Operator: A class that acts as a template for carrying out some work.
* Task: Defines work by implementing an operator, written in Python.
* Task Instance: An instance of a task - that has been assigned to a DAG and has a state associated with a specific DAG run (i.e for a specific execution\_date).
* execution\_date: The logical date and time for a DAG Run and its Task Instances.

**DAG:**

A DAG is a data pipeline

1->2->3 (flow should not be looped)

**OPERATOR:** - an Operator is a task.

A wrapper around the class to do operations, connecting a database will require an operator that passes data.

3 types:

-- Action Operator: Executing - Functions/commands (Bash Operator/ Python Operator)

-- Transfer Operator: Allow to trasfer data from S->D Presto data to Mysql

-- Sensor Operator: Wait for something to perform -> Example: File sensor waits for some files to be in the folder to perform

**Task/Task Instance:**

-- As soon as a task is triggered in DP, that task becomes a task instance.

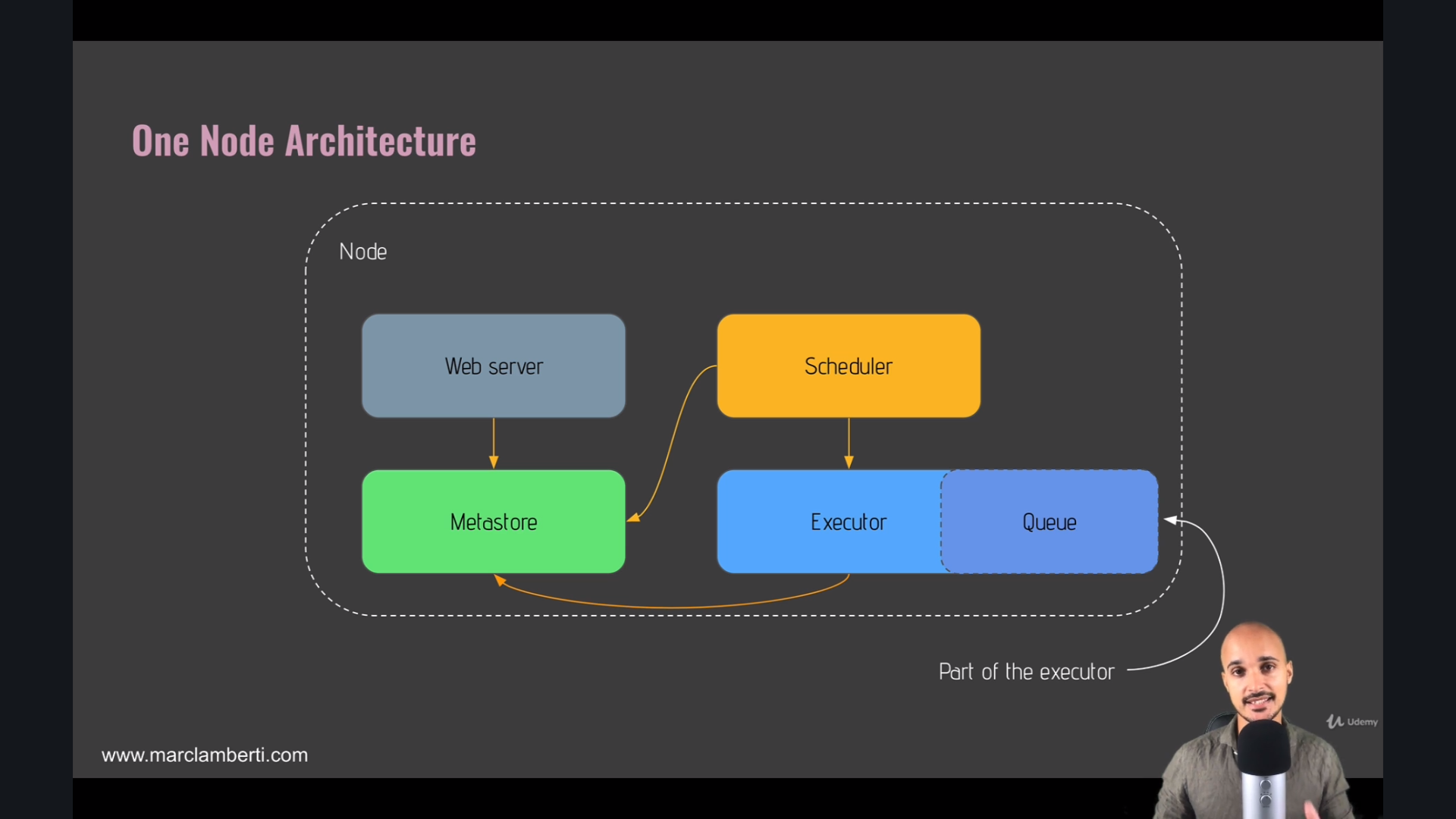
-- A executor also becomes task instance

**WORKFLOW:**

All put together is called a workflow

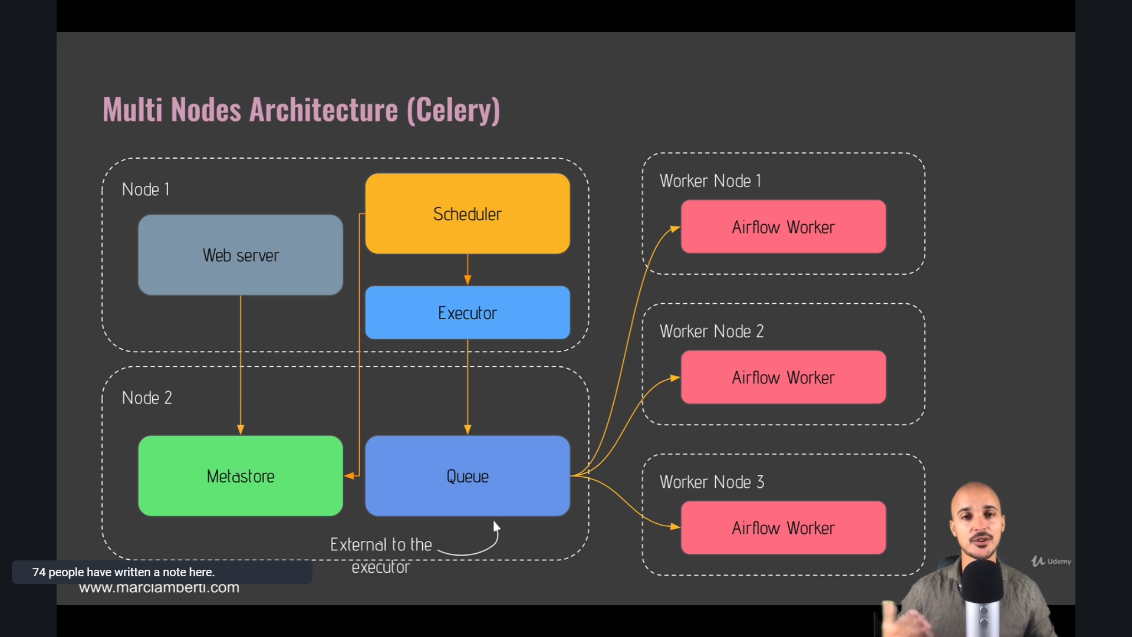
**## ARCHITECTURE**

Single Node:

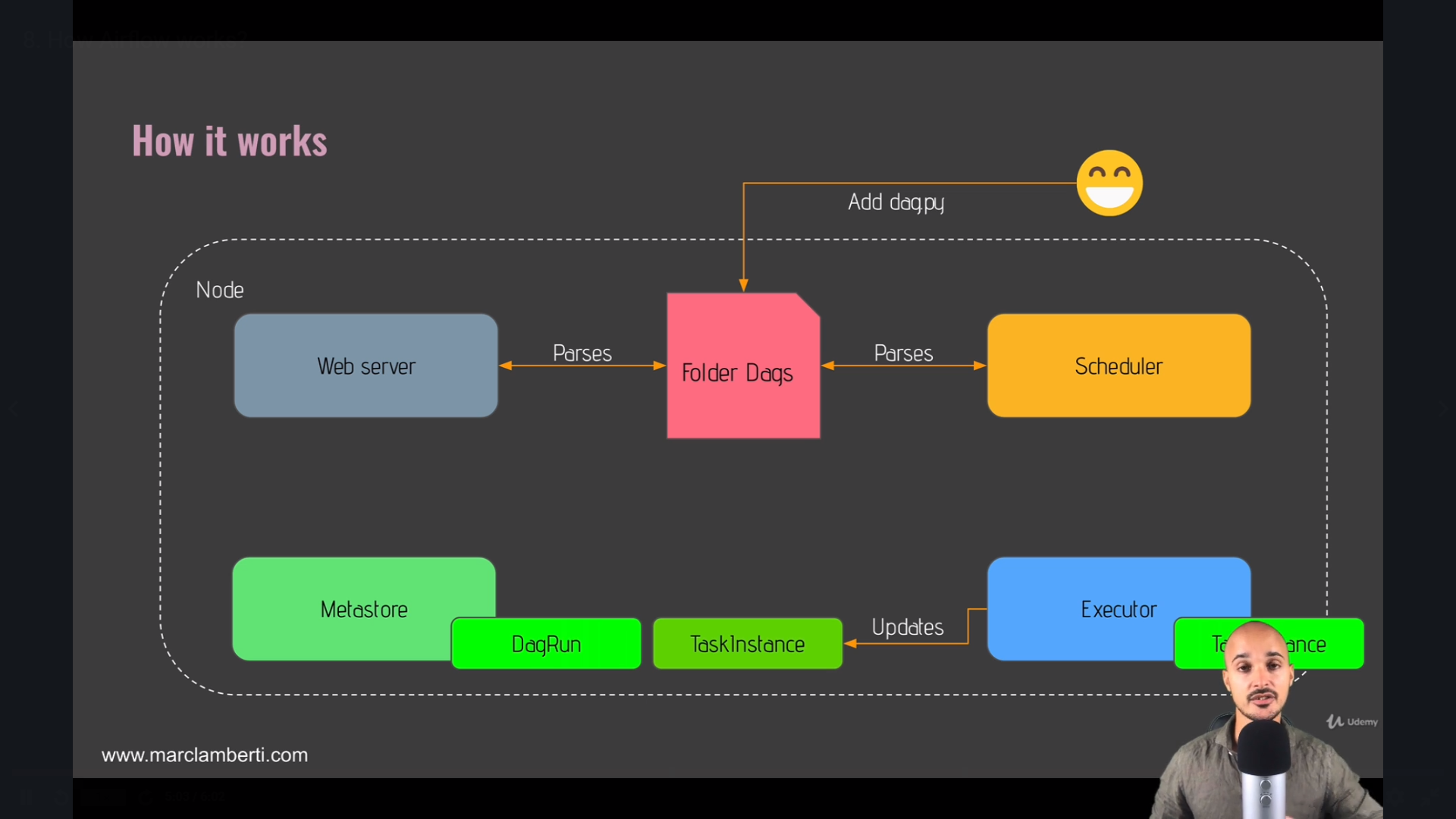


webserver // Scheduler // Metastore // Executor

Multi Node:

* Queue will be in a separate node and not in the Executor itself
* The worker nodes will fetch the tasks from the queue and executes them

New DP – we will add to the **folder dags**



Scheduler and Webserver parses the Folder dags if we put new files/dag files in the Folder Dags.

* DAG Run object is created when ready by the Scheduler (DP is triggered ) It is an initialization of the DAG. It creates a status Running in the metastore.
* When Task is ready for your DP the Task instance object is created
* Task Instance will be sent to the Executor by the Scheduler
* Executor runs the task instance and it keeps updating the status of Task instance Objects to the metastore
* If Successful, the DAG Run Object status is set to Completed
* UI shows the status

airflow db init is the first command to execute to initialise Airflow

If a task fails, check the logs by clicking on the task from the UI and "Logs"

The Gantt view is super useful to sport bottlenecks and tasks are too long to execute

Section 1:

Airflow - Hands On

open Aiflow.ova

1. Visual Studio Code

2. Install all extensions - remote-ssh, git pull, docker, yaml

3. Windows F1 - Type "remote-ssh: Connect to host" then "Add host"

4. Type - "ssh -p 2222 airflow@localhost"

5. choose the .config file

6. Check the added config file in the VS code

7. Again press F1 and choose remote-ssh: Connect to host "localhost"

8. Might promt for "linux", then "yes" to add key

9. Then password : "airflow"

10. Terminal menu click open "New terminal"

airflow@airflowvm:~$ python3 -m venv sandbox

airflow@airflowvm:~$ source sandbox/bin/activate

(sandbox) airflow@airflowvm:~$ pip install wheel

(sandbox) airflow@airflowvm:~$ pip install apache-airflow==2.0.0 --constraint https://gist.githubusercontent.com/marclamberti/742efaef5b2d94f44666b0aec020be7c/raw/5da51f9fe99266562723fdfb3e11d3b6ac727711/constraint.txt

$ airflow db init (to inititalize metadata/db settings)

$ airflow webserver

$ airflow users create -u admin -p admin -f arunraja -l gvk -r Admin -e arunrajagvk.edu@gmail.com

-- localhost:8080 / login : admin pwd: admin

$ airflow scheduler

-- Followed by UI

Creating First Data Pipeline

Steps:

1. Creating table

2. Is\_api\_available

3. extracting\_user from the api

4. Processing\_user

5. Storing\_user

DO's

One operator one task

-- Provider packages : http://airflow.apache.org/docs/apache-airflow-providers/packages-ref.html

-- Do pip install

-- Add connection details to the Admin -- Operators

Step1: ConnID: 'db\_sqlite' (same as in the dag file)

Step2: Connection type: sqlite

Step3: Host: /home/airflow/airflow/airflow.db

Testing a Task:

$ airfow tasks test <dag\_name> <task\_name> <exec date>

airflow tasks test user\_processing creating\_table 2020-01-01

airflow tasks test user\_processing is\_api\_available 2020-01-01

-- Scheduling a dag IMPORTANT:

the catchup=True will trigger all the non executed dags starting from the recent execution\_date

and not the start date mentioned in the dag

If catchup=False it will only trigger for the current execution date

# run your first task instance

airflow tasks run example\_bash\_operator runme\_0 2015-01-01

# run a backfill over 2 days

airflow dags backfill example\_bash\_operator \

--start-date 2015-01-01 \

--end-date 2015-01-02

ALL dates are UTC

we can change the timezone from >> airflow.cfg file "default\_ui\_timezone=utc"

with DAG('user\_processing', schedule\_interval='@daily',

default\_args=default\_args, catchup=False) as dag:

Default Executor for Airflow: SequentialExecutor

airflow@airflowvm:~/airflow$ airflow config get-value core sql\_alchemy\_conn

sqlite:////home/airflow/airflow/airflow.db

airflow@airflowvm:~/airflow$ ls

airflow-webserver.pid airflow.cfg airflow.db dags logs unittests.cfg webserver\_config.py

--Sqlite doesnt allow multiple writes at the same time that's why we cannot use sql lite for parallel

airflow@airflowvm:~/airflow$ airflow config get-value core executor

SequentialExecutor

-- To execute multiple tasks parallel, we have to change the DB to Postgres which supports multiple read and write parallel

-- Change the executor to LocalExecutor - converts tasks to subprocess and runs parallel in machine

Step1:

airflow@airflowvm:~/airflow$ sudo apt update

airflow@airflowvm:~/airflow$ sudo apt install postgresql

-- open postgresql

(sandbox) airflow@airflowvm:~$ sudo -u postgres psql

$$ ALTER TABLE user PASSWORD 'postgres';

\q

Step 2:

-- Change airflow.cfg

sql\_alchemy\_conn = postgresql+psycopg2://postgres:postgres@localhost/postgres

# The executor class that airflow should use. Choices include

# ``SequentialExecutor``, ``LocalExecutor``, ``CeleryExecutor``, ``DaskExecutor``,

# ``KubernetesExecutor``, ``CeleryKubernetesExecutor`` or the

# full import path to the class when using a custom executor.

executor = LocalExecutor

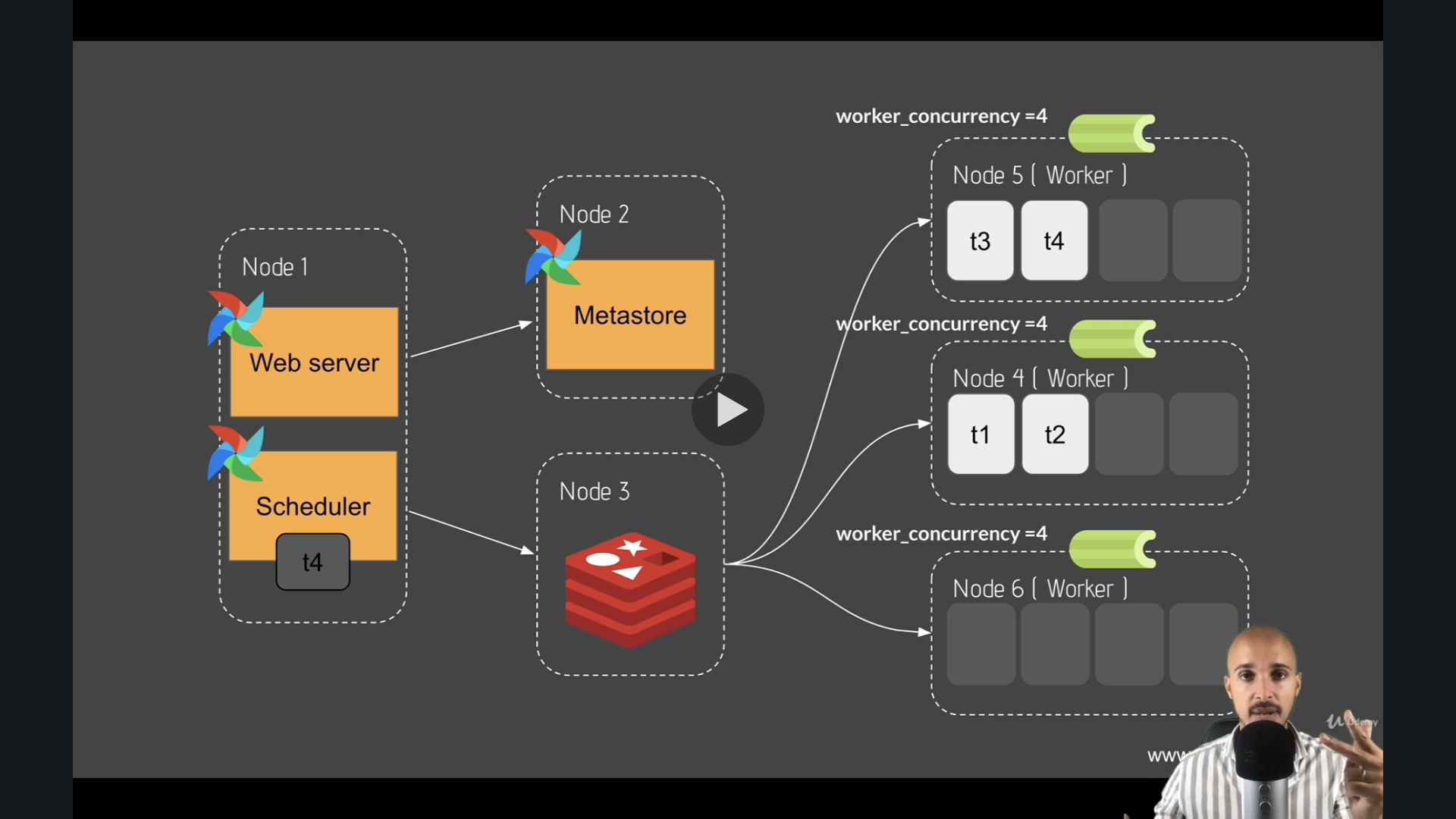
Step 3:

-- Stop airflow scheduler and webserver

-- (sandbox) airflow@airflowvm:~/airflow$ airflow db init

-- Create user :

(sandbox) airflow@airflowvm:~/airflow$ airflow users create -u admin -p admin -r Admin -f admin -ladmin -e admin@airflow.co

**—Airflow in production - scale parallel execution of tasks :**

Scale executors - Multiple machines/nodes

to run multiple tasks parallelly in cluster : CeleryExecutor/KubernetesExecutor play major role.

* STOP airflow scheduler and webserver

(sandbox) airflow@airflowvm:~/airflow$ pip install 'apache-airflow[celery]'

* Sudo apt update
* (sandbox) airflow@airflowvm:~/airflow$ sudo apt install redis-server
* (sandbox) airflow@airflowvm:~/airflow$ sudo nano /etc/redis/redis.conf

Edit the file redis.conf

Supervised no 🡪>>>> supervised system

--save and exit with c-X

Airflow.cfg

executor = CeleryExecutor

broker\_url = redis://localhost:6379/0

result\_backend = db+postgresql://postgres:postgres@localhost/postgres

(sandbox) airflow@airflowvm:~/airflow$ pip install 'apache-airflow[redis]'

// Full command list

174 source sandbox/bin/activate

175 sudo apt install postgresql

176 sudo -u postgres psql

177 pip install 'apache-airflow[postgres]'

178 airflow db check

179 cd airflow/

180 ls

181 airflow db init

182 airflow users create -u admin -p admin -r Admin -f admin –l admin -e admin@airflow.com

183 pip install 'apache-airflow[celery]'

184 sudo apt update

185 sudo apt install redis-server

186 sudo nano /etc/redis/redis.conf

187 sudo systemctl restart redis.service

188 sudo systemctl start redis.service

189 sudo systemctl status redis.service

190 pip install 'apache-airflow[redis]'

* Instantiate celery worker by “airflow celery worker”
* UI using “Airflow celery flower”

PARALLELIZM:

Airflow can be configured to run maximum parallel tasks and concurrency for dags using

---Parallelism = <number>

---dag\_concurrency = <number> /// This will run the maximum per dag

The above will trigger 16 dag runs

If our dags on one date depends on the next one then we need to run one dag run at a time:

---max\_active\_runs\_per\_dag=1

Programmatically setting them in dag.py file is, Add a parameter to the DAG object.

Concurrency:

with DAG ('parallel\_dag', schedule\_interval='@daily',

        default\_args=default\_args, concurrency = 1, catchup=False) as dag:

max\_active\_runs:

with DAG ('parallel\_dag', schedule\_interval='@daily',

        default\_args=default\_args, max\_active\_runs=1, catchup=False) as dag:

SUB DAGS: they are depreciated

from subdags.subdag\_parallel\_dag import subdag\_parallel\_dag

USE – TASKGROUP

To group parallel tasks , we need to use taskgroup

from airflow.utils.task\_group import TaskGroup

* BELOW DAG Object

with TaskGroup('processing\_tasks') as processing\_tasks:

        task2 = BashOperator(

            task\_id='task2',

            bash\_command='sleep 3'

        )

        with TaskGroup('spark\_tasks') as spark\_tasks:

            task3 = BashOperator(

                task\_id='task3',

                bash\_command='sleep 3'

            )

XCOMS:

Best and optimal way to transport message between tasks. But it can handle or storage less data size in metadata. 1 GB for Postgresql

Push and pull data with xcoms

* Instantiating and passing TASK INSTANCE (ti) is important for Xcoms

def \_training\_model(ti):

    accuracy = uniform(0.1, 10.0)

    print(f'model\'s accuracy: {accuracy}')

    ti.xcom\_push(key='model\_accuracy', value=accuracy)

def \_choose\_best\_model(ti):

    print('choose best model')

    accuracies = ti.xcom\_pull(key='model\_accuracy', task\_ids=[

        'processing\_tasks.training\_model\_a',

        'processing\_tasks.training\_model\_b',

        'processing\_tasks.training\_model\_c'

    ])

    print(accuracies)

By default some operators push xcoms value as empty. To avoid that we need to include

do\_xcom\_push=False

in the Operator method of our dag file.

  downloading\_data = BashOperator(

        task\_id='downloading\_data',

        bash\_command='sleep 3',

        do\_xcom\_push=False

    )

### **Trigger Rules**

Though the normal workflow behavior is to trigger tasks when all their directly upstream tasks have succeeded, Airflow allows for more complex dependency settings.

All operators have a trigger\_rule argument which defines the rule by which the generated task get triggered. The default value for trigger\_rule is all\_success and can be defined as “trigger this task when all directly upstream tasks have succeeded”. All other rules described here are based on direct parent tasks and are values that can be passed to any operator while creating tasks:

* all\_success: (default) all parents have succeeded
* all\_failed: all parents are in a failed or upstream\_failed state
* all\_done: all parents are done with their execution
* one\_failed: fires as soon as at least one parent has failed, it does not wait for all parents to be done
* one\_success: fires as soon as at least one parent succeeds, it does not wait for all parents to be done
* none\_failed: all parents have not failed (failed or upstream\_failed) i.e. all parents have succeeded or been skipped
* none\_failed\_or\_skipped: all parents have not failed (failed or upstream\_failed) and at least one parent has succeeded.
* none\_skipped: no parent is in a skipped state, i.e. all parents are in a success, failed, or upstream\_failed state
* dummy: dependencies are just for show, trigger at will

Note that these can be used in conjunction with depends\_on\_past (boolean) that, when set to True, keeps a task from getting triggered if the previous schedule for the task hasn’t succeeded.

One must be aware of the interaction between trigger rules and skipped tasks in schedule level. Skipped tasks will cascade through trigger rules all\_success and all\_failed but not all\_done, one\_failed, one\_success, none\_failed, none\_failed\_or\_skipped, none\_skipped and dummy.

from airflow.utils.trigger\_rule import TriggerRule

join = DummyOperator(task\_id='join', dag=dag, trigger\_rule='none\_failed\_or\_skipped')

## Airflow Docker & Setup:

## docker-compose.yaml

To deploy Airflow on Docker Compose, you should fetch [docker-compose.yaml](https://airflow.apache.org/docs/apache-airflow/stable/docker-compose.yaml).

curl -LfO 'https://airflow.apache.org/docs/apache-airflow/2.0.1/docker-compose.yaml'

This file contains several service definitions:

* airflow-scheduler - The [scheduler](https://airflow.apache.org/docs/apache-airflow/stable/scheduler.html) monitors all tasks and DAGs, then triggers the task instances once their dependencies are complete.
* airflow-webserver - The webserver available at http://localhost:8080.
* airflow-worker - The worker that executes the tasks given by the scheduler.
* airflow-init - The initialization service.
* flower - [The flower app](https://flower.readthedocs.io/en/latest/) for monitoring the environment. It is available at http://localhost:8080.
* postgres - The database.
* redis - [The redis](https://redis.io/) - broker that forwards messages from scheduler to worker.

All these services allow you to run Airflow with [CeleryExecutor](https://airflow.apache.org/docs/apache-airflow/stable/executor/celery.html).

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.
* ./plugins - you can put your [custom plugins](https://airflow.apache.org/docs/apache-airflow/stable/plugins.html) here

Initializing Environment

* Before starting Airflow for the first time, You need to prepare your environment, i.e. create the necessary files, directories and initialize the database.
* On **Linux**, the mounted volumes in container use the native Linux filesystem user/group permissions, so you have to make sure the container and host computer have matching file permissions.
* mkdir ./dags ./logs ./plugins
* echo -e "AIRFLOW\_UID=$(id -u)\nAIRFLOW\_GID=0" > .env
* On **all operating system**, you need to run database migrations and create the first user account. To do it, run.
* docker-compose up airflow-init
* After initialization is complete, you should see a message like below.
* airflow-init\_1 | Upgrades done
* airflow-init\_1 | Admin user airflow created
* airflow-init\_1 | 2.0.1
* start\_airflow-init\_1 exited with code 0
* The account created has the login airflow and the password airflow.
* Running Airflow
* Now you can start all services:
* docker-compose up