Airflow

a platform to programmatically author, schedule and monitor workflows.

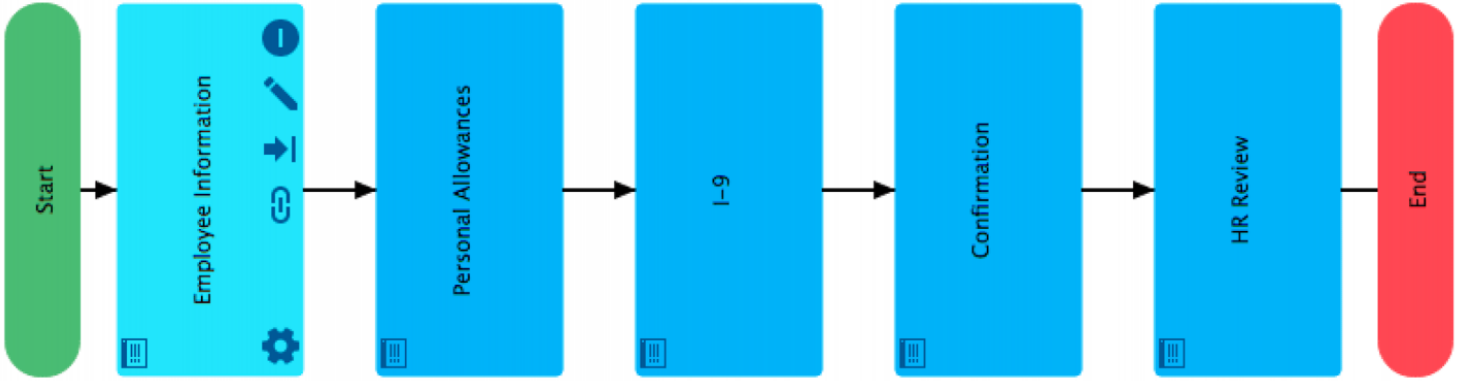
Fact & Problems

We’re facing with **a lots of workflows.**

Sale Orders


Salesperson Creates Order
Manager reviews and approves the order
VP reviews and approves but only if amount > $10K
Send to Client for signature(external to the company)
Forward to Billing for Invoicing
Receive Payment from Client
Fulfill the Sales Order
File documents for Record Keeping.

New Employee Onboarding

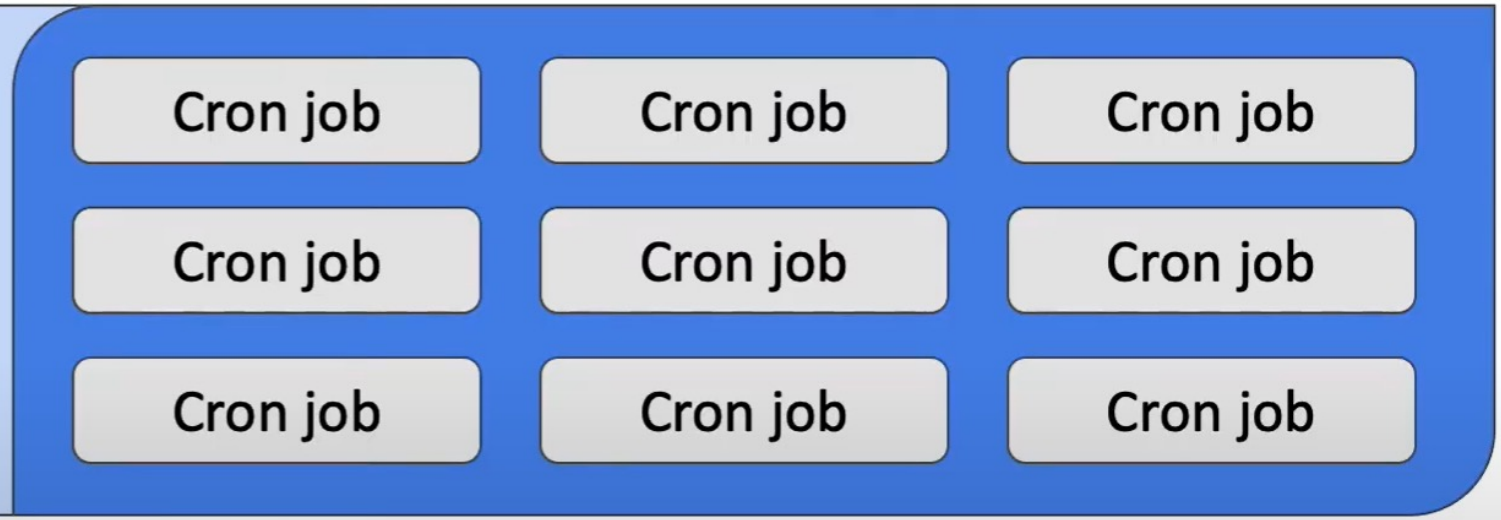


We’re facing with
a lots of disadvantages with current methods.

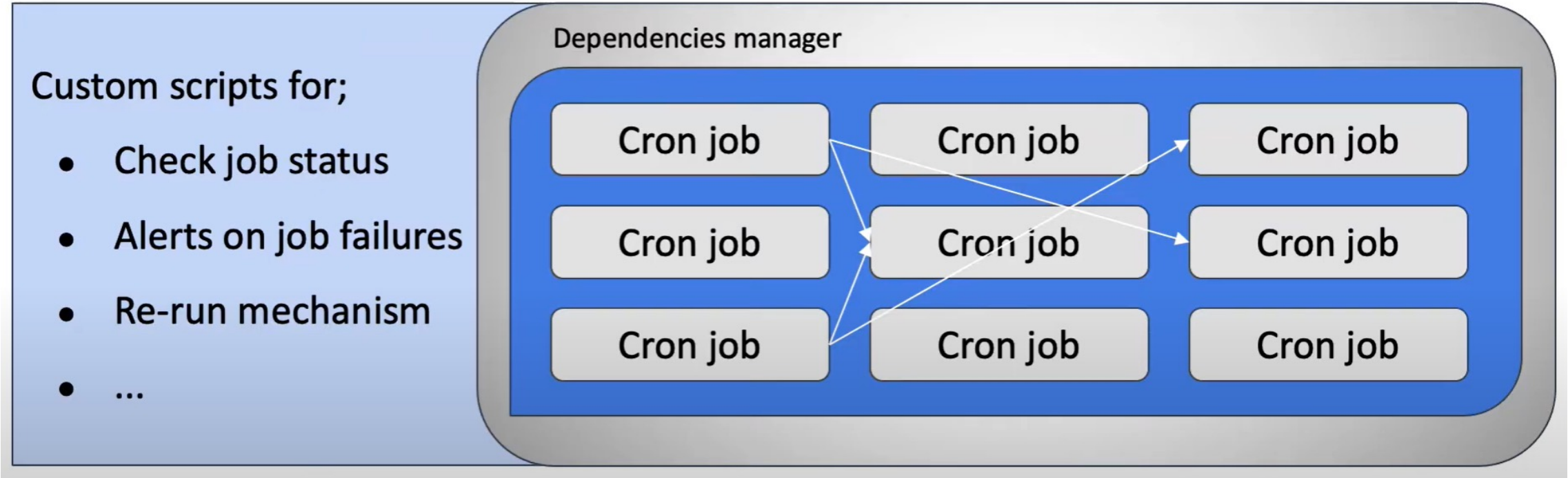
Homemade *orchestration system* to monitor & maintain 100+ cron jobs

Custom scripts for;

- Check job status
- Email on job failures
- Re-run mechanism
- ...



Spend efforts on maintaining both!



Do nothing without saying **Good luck!**

No notification when finish/fail/retry

No status monitoring for running script

No flexible integration mechanism

No dependencies management

No version controller for workflow's scripts

No failure-retry and re-run mechanism

No centralize logging mechanism

No backfill mechanism

No scalability

Airflow
comes into the picture

Go deep in technical perspective

Agenda

- Overview of Airflow
- Airflow Operators
- Airflow Executors
- Demo

Apache Airflow

airbnb

Airflow is **created by Airbnb**

Written in **Python**

Is a **workflow management** system

Open-source, no cost

“Configuration as code” principal

Apache Airflow

airbnb

Airflow is **used for:**

Run ETL pipelines

Data ingestion pipelines

Machine learning pipelines

Predictive data pipelines

General purpose scheduler

We need to know **some concepts.**

DAG
Directed Acyclic Graph

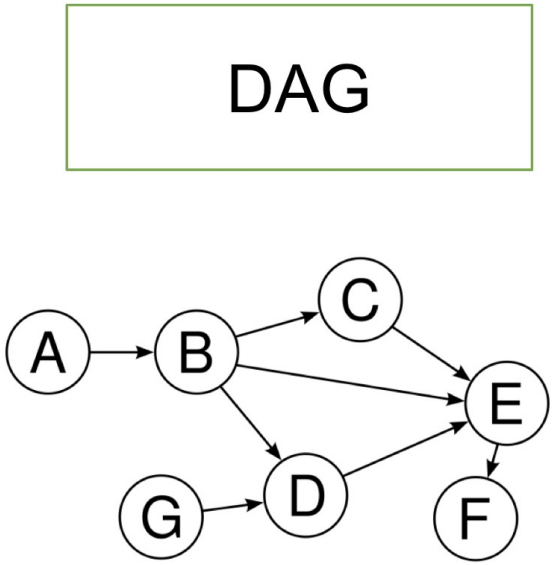


Figure 1

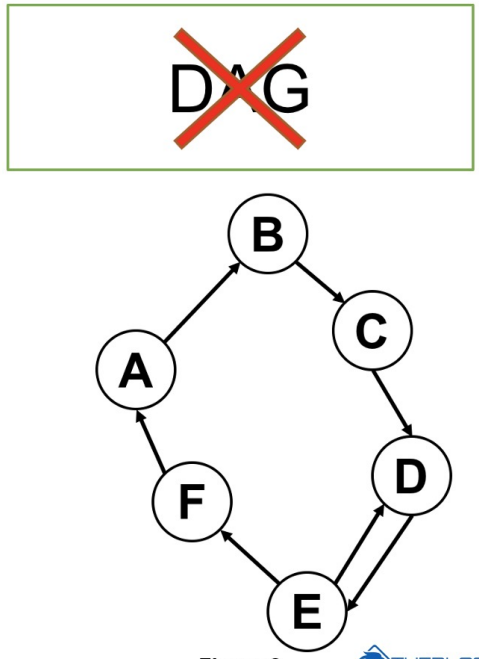
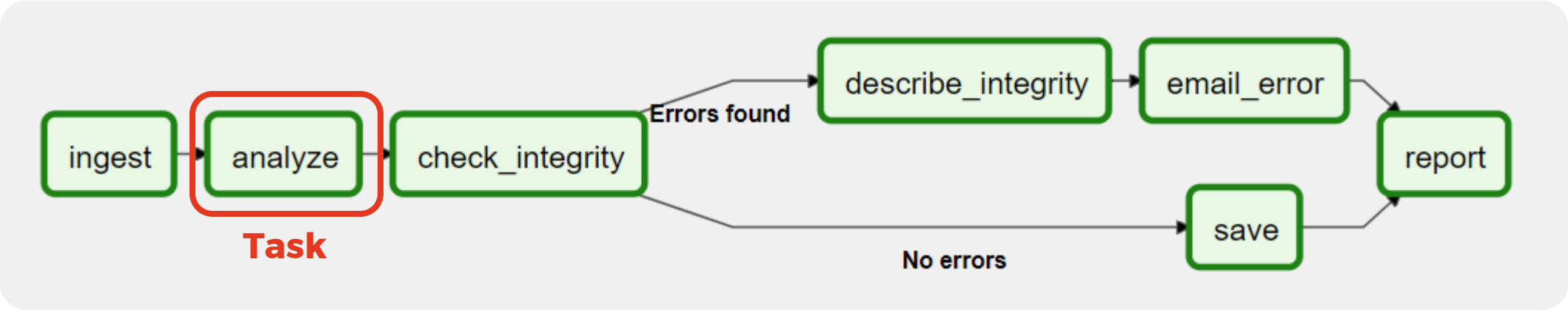


Figure 2

DAG

Directed Acyclic Graph



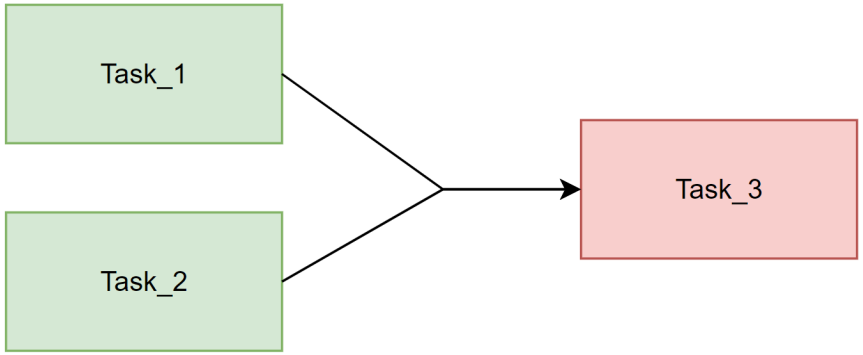
DAG
Directed Acyclic Graph

We need calculate: **(a+b) * (d+c)**

task_1 = a + b

task_2 = d + c

task_3 = task_1 * task_2



We write DAG:

[task_1, task_2] >> task_3

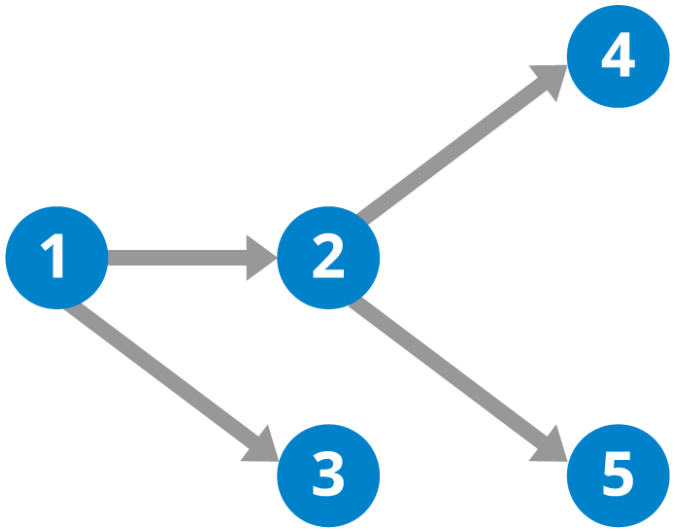
```python
import datetime as dt

from airflow.models import DAG
from airflow.operators.dummy_operator import DummyOperator
from airflow.operators.latest_only_operator import LatestOnlyOperator
from airflow.utils.dates import days_ago
from airflow.utils.trigger_rule import TriggerRule

dag = DAG(
    dag_id='latest_only_with_trigger',
    schedule_interval=dt.timedelta(hours=4),
    start_date=days_ago(2),
    tags=['example']
)

latest_only = LatestOnlyOperator(task_id='latest_only', dag=dag)
task1 = DummyOperator(task_id='task1', dag=dag)
task2 = DummyOperator(task_id='task2', dag=dag)
task3 = DummyOperator(task_id='task3', dag=dag)
task4 = DummyOperator(task_id='task4', dag=dag, trigger_rule=TriggerRule.ALL_DONE)

latest_only >> task1 >> [task3, task4]
task2 >> [task3, task4]
```



A DAG can run **manually** or **automatically**.

A DAG can be **scheduled in the future.**

A DAG can run **periodically**

Three types of **Task**

- Operators
- Sensors
- TaskFlow

Operators

An Operator is conceptually a template for a predefined Task

Builtin

BashOperator - executes a bash command

PythonOperator - calls an arbitrary Python function

EmailOperator - sends an email

Provider Packages

SimpleHttpOperator

MySqlOperator

PostgresOperator

MsSqlOperator

OracleOperator

JdbcOperator

DockerOperator

KubernetesPodOperator

S3FileTransformOperator

PrestoToMySqlOperator

SlackAPIOperator

Sensors

Sensors are a special type of Operator that are designed to do exactly one thing - wait for something to occur

- The **FileSensor**: Waits for a file or folder to land in a filesystem.
- The **S3KeySensor**: Waits for a key to be present in a S3 bucket.
- The **SqlSensor**: Runs a sql statement repeatedly until a criteria is met.
- The **ExternalTaskSensor**: Waits for a different DAG or a task in a different DAG to complete for a specific execution date.
- The **DateTimeSensor**: Waits until the specified datetime (Useful to add some delay to your DAGs)
- The **TimeDeltaSensor**: Waits for a timedelta after the task's execution_date + schedule interval (Looks similar to the previous one no?)

TaskFlow

```python
from airflow.decorators import task
from airflow.operators.email import EmailOperator

@task
def get_ip():
    return my_ip_service.get_main_ip()

@task
def compose_email(external_ip):
    return {
        'subject':f'Server connected from {external_ip}',
        'body': f'Your server executing Airflow is connected from the external IP {external_ip}<br>'
    }

email_info = compose_email(get_ip())

EmailOperator(
    task_id='send_email',
    to='example@example.com',
    subject=email_info['subject'],
    html_content=email_info['body']
)
```

Architect

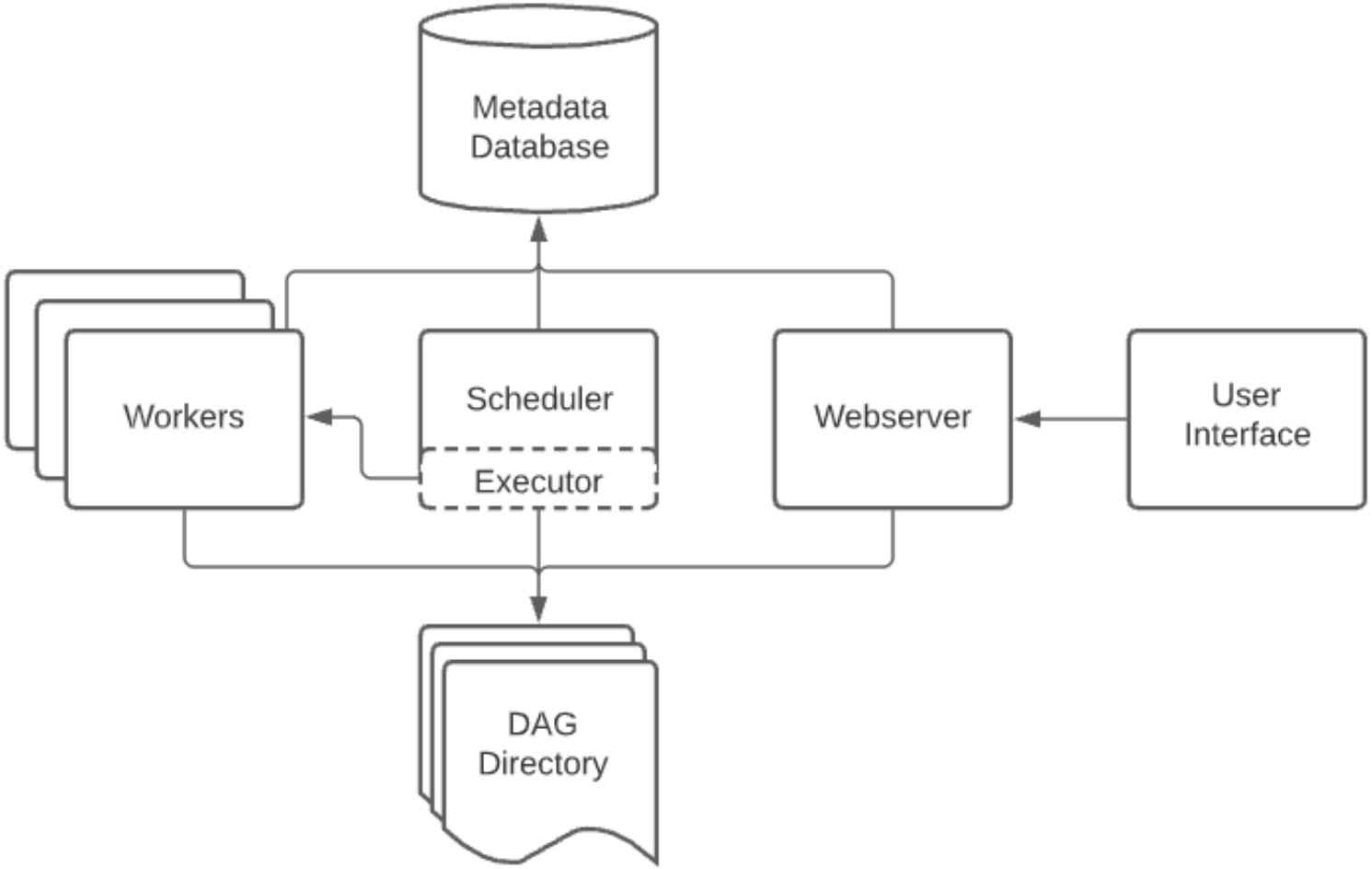
Metadata Database
to store system data

DAG Directory
to store DAG files

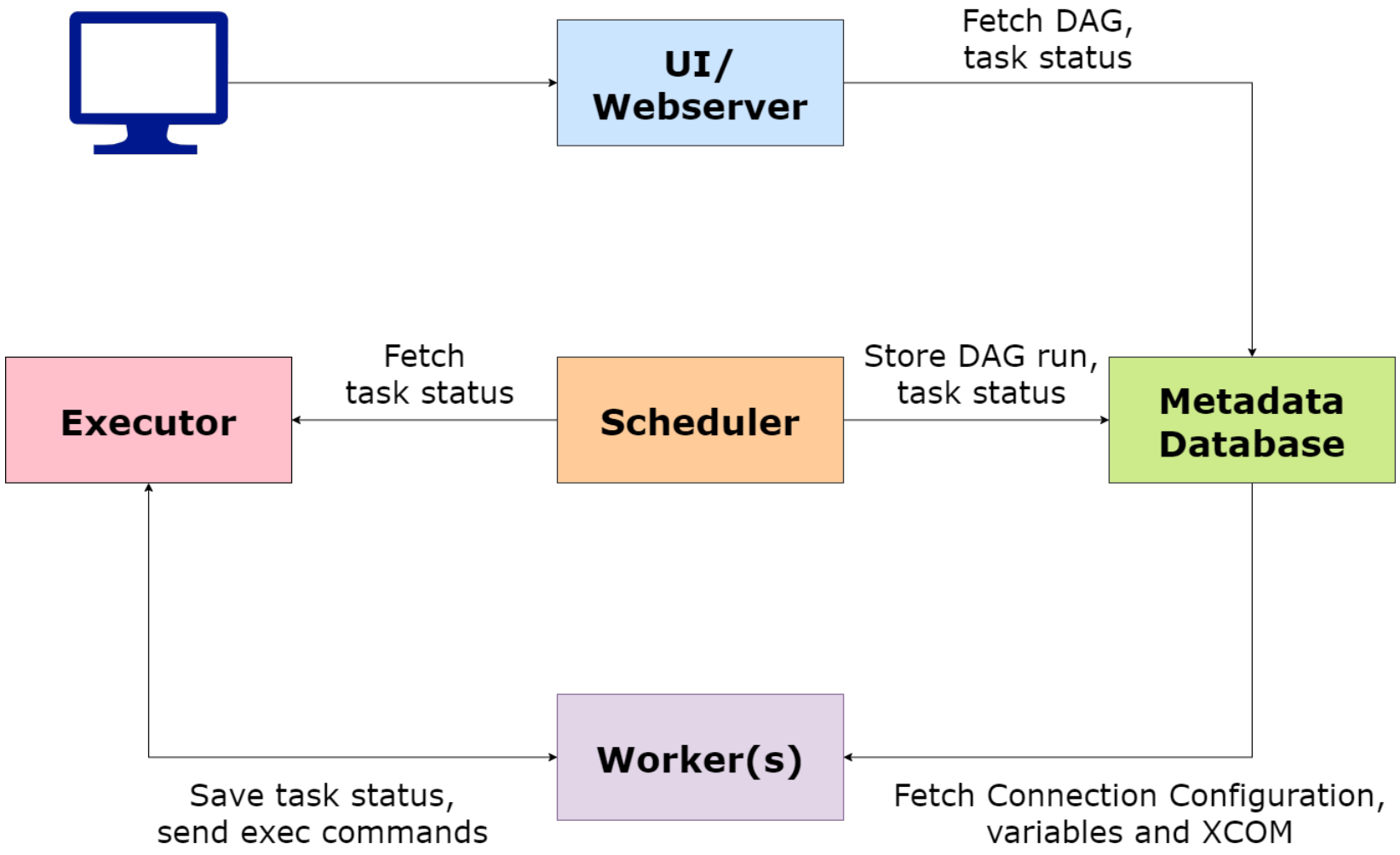
Webserver
to monitor executors

Scheduler
engine to fire executors

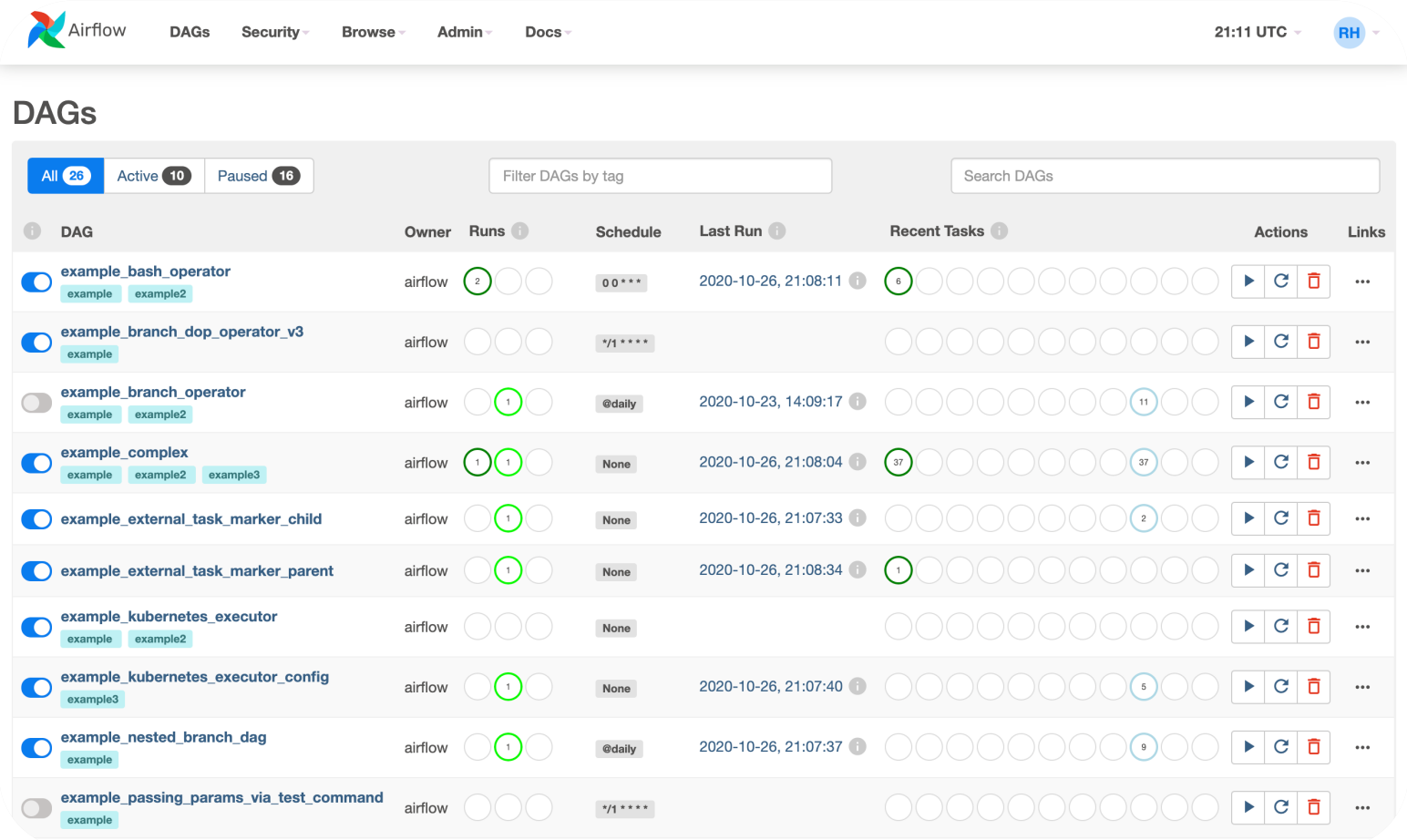
Executor
execute DAGs



Architect


UI/
Webserver
Fetch DAG,
task status
Executor
Fetch
task status
Scheduler
Store DAG run,
task status
Metadata
Database
Worker(s)
Save task status,
send exec commands
Fetch Connection Configuration,
variables and XCOM

Webserver


Airflow
DAGs
Security
Browse
Admin
Docs
21:11 UTC
RH
DAGs
All 26
Active 10
Paused 16
Filter DAGs by tag
Search DAGs
DAG
Owner
Runs
Schedule
Last Run
Recent Tasks
Actions
Links
example_bash_operator
example
example2
airflow
0 0 * * *
2020-10-26, 21:08:11
example_branch_dop_operator_v3
example
airflow
*/1 * * * *
example_branch_operator
example
example2
airflow
@daily
2020-10-23, 14:09:17
example_complex
example
example2
example3
airflow
None
2020-10-26, 21:08:04
example_external_task_marker_child
airflow
None
2020-10-26, 21:07:33
example_external_task_marker_parent
airflow
None
2020-10-26, 21:08:34
example_kubernetes_executor
example
example2
airflow
None
example_kubernetes_executor_config
example3
airflow
None
2020-10-26, 21:07:40
example_nested_branch_dag
example
airflow
@daily
2020-10-26, 21:07:37
example_passing_params_via_test_command
example
airflow
*/1 * * * *

Executors

Local

Debug Executor - debug tool and can be used from IDE

Local Executor - runs tasks by spawning processes in a controlled fashion in different modes.

Sequential Executor - run one task instance at a time (for testing purpose)

Remote

Celery Executor

CeleryKubernetes Executor

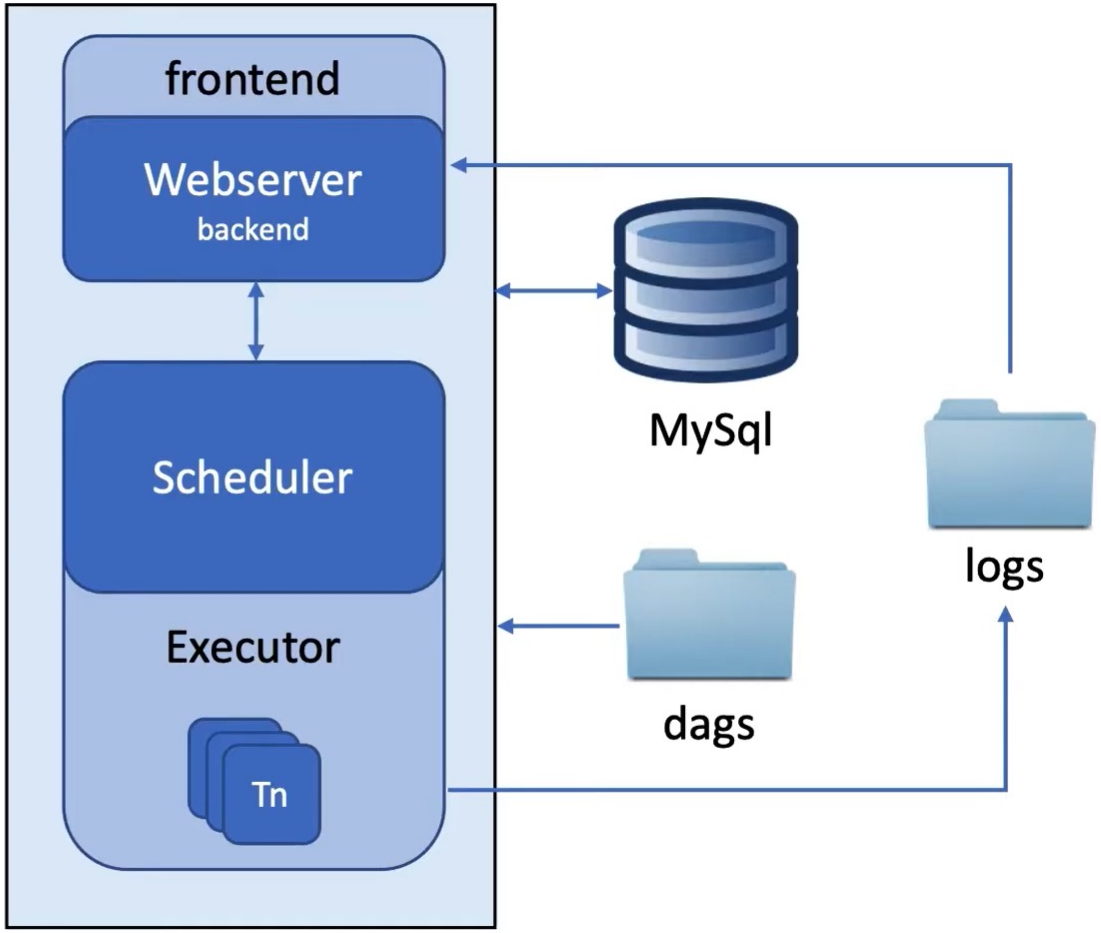
Dask Executor

Kubernetes Executor

LocalKubernetes Executor

Local Executor

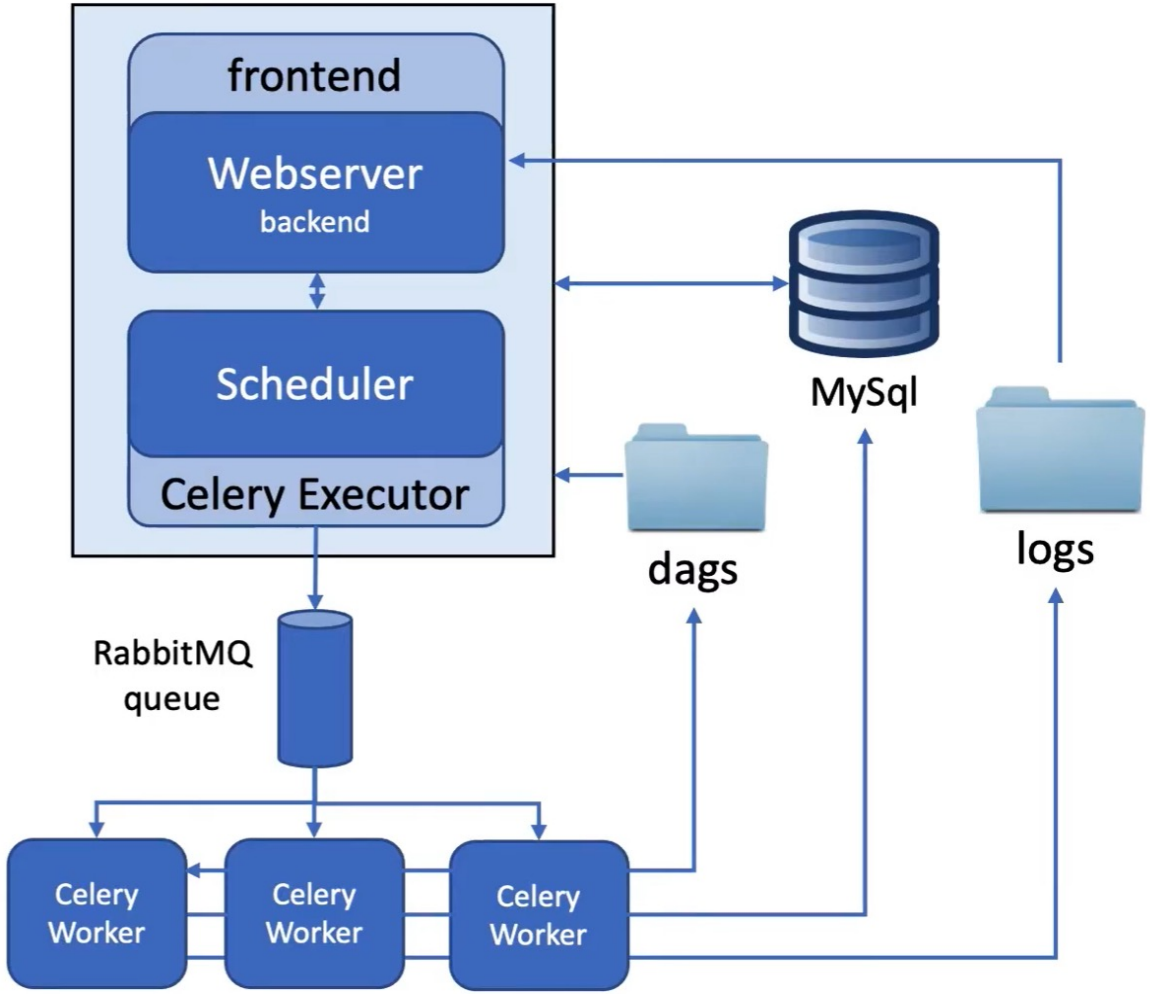
one of the ways you can scale out the number of workers.



- runs within scheduler
- multiple task instance at a time
- Pros:
  - ✓ Easy to Setup
  - ✓ Cheap & Light weight
  - ✓ Can run multiple tasks
- Cons:
  - ✗ Not suitable to scale
  - ✗ Not suitable for production
  - ✗ Single point of failure

Celery Executors

one of the ways you can scale out the number of workers.

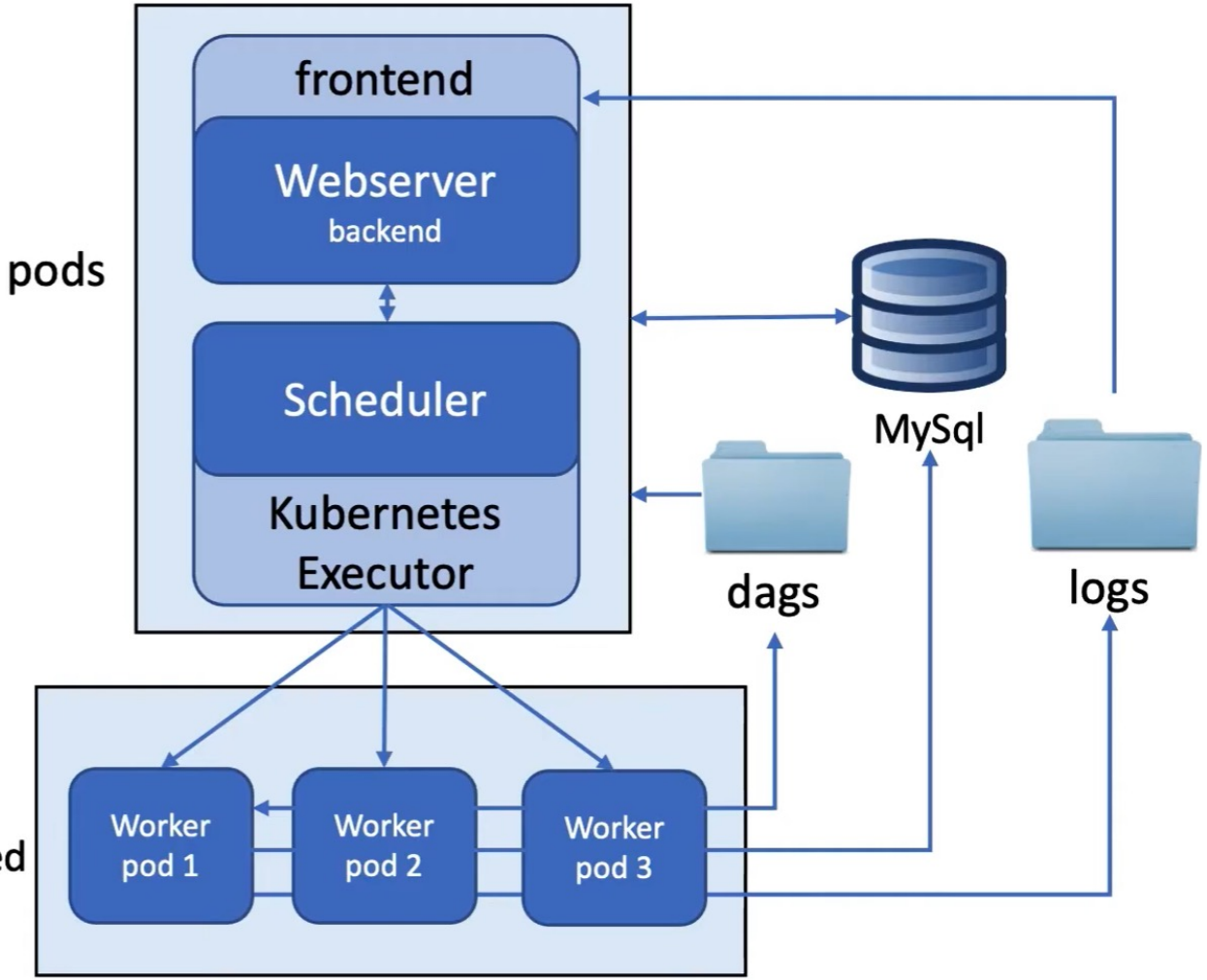


- Runs tasks on dedicated machine
- Distributed Task Queue
- Pros:
  - ✓ Build for horizontal scaling
  - ✓ Fault tolerant
  - ✓ Production ready
- Cons:
  - ✗ Takes time to setup
  - ✗ Resource wastage if no task scheduled
  - ✗ Not cost effective

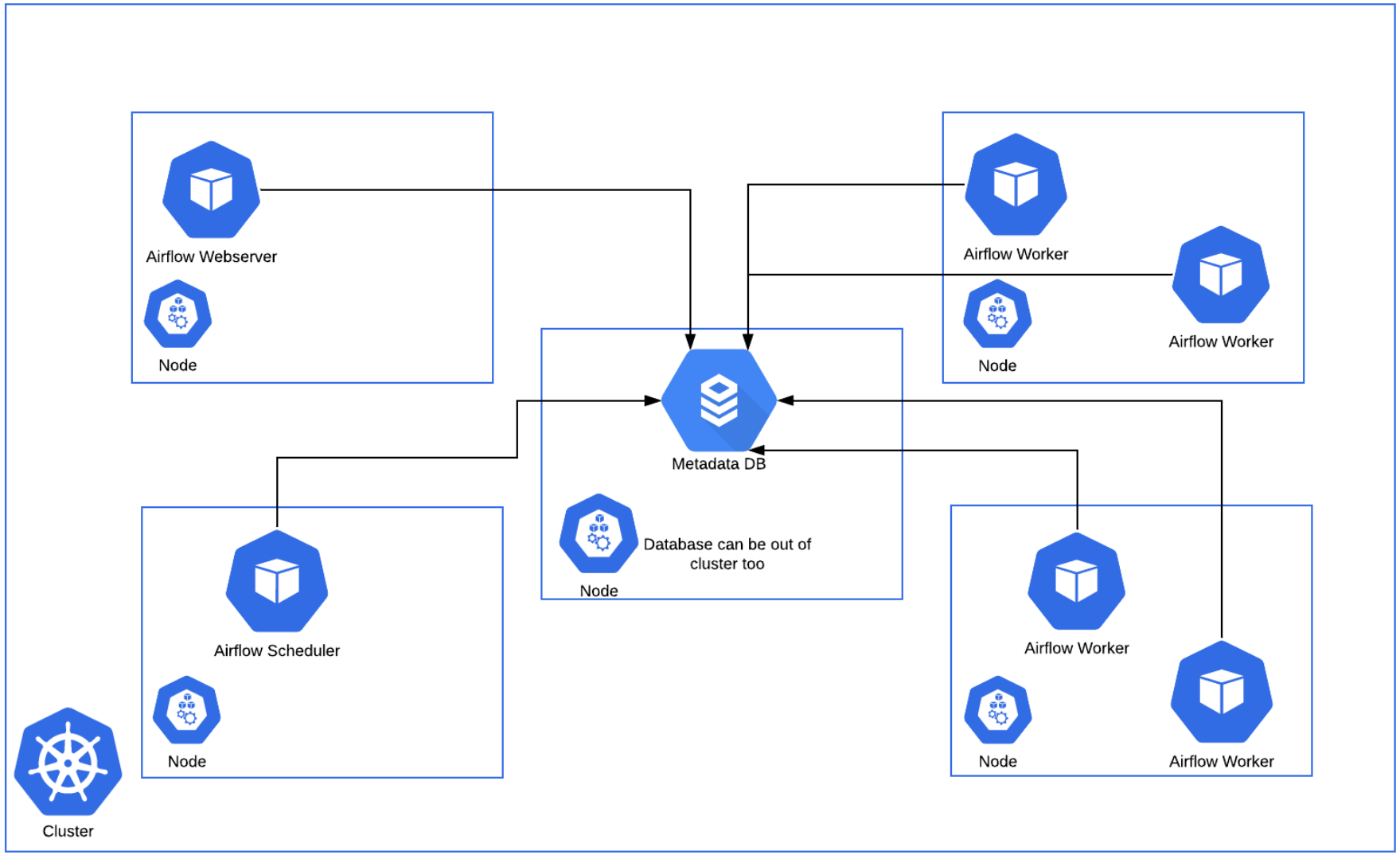
K8sExecutors

The Kubernetes executor runs each task instance in its own pod on a Kubernetes cluster.

- Runs tasks on dedicated pod
- Uses Kubernetes API to manage pods
- Pros:
  - ✓ Can Scale to Zero
  - ✓ Fault tolerant
  - ✓ Can assign resources to individual tasks
  - ✓ Cost & resource effective
- Cons:
  - ✗ Launching Pod for each task takes few seconds
  - ✗ Kubernetes knowledge required



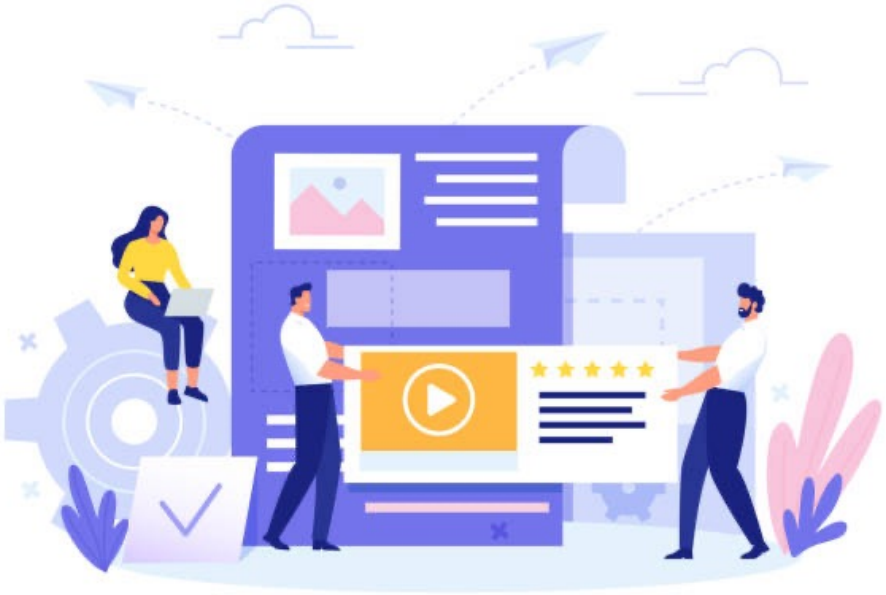
K8sExecutors



K8sExecutors vs K8sPodOperator

K8s Executor – create a Pod with Airflow image (can not change) and then run the task inside it. Task in Pod will save result into DB.

K8s Pod Operator – the task run outside of K8s cluster and then create Pod with any image and run anything inside Pod. After finish, Pod return the results to task, task save result into DB.



Demo

Case #1

Run simple DAG at Local

Case #2

Run complex DAG on local

Case #3

FileSensor

Case #4

ML training pipeline

Thank you
for your attention!