Data Pipeline Orchestration con Apache Airflow

GEOdaysIT 2023 - OSMit

$ whoami

Daniele Santini

Ingegnere informatico

Software developer @ GELLIFY

dsantini.it

GitLab: dsantini

GitHub: Danysan1

Cos'è una pipeline di dati?

Flusso predefinito di elaborazione dati, tipicamente ricorrente

- **Ingestione** dati da una o più fonti
- Sequenza o flusso di task di **elaborazione** / filtraggio / trasformazione / ...
- **Caricamento** dati trasformati su una o più destinazioni
- **Analisi** e/o verifiche e/o invio report sui dati

Può essere Batch, in tempo reale o combinata

Metodi tradizionali e nuove sfide

Tradizionalmente gestite con **script** custom, **cron job**, **Makefile**, ...

Sfide:

- **Visione di insieme** delle pipeline in corso, schedulate e terminate
- Gestione di connessioni, **errori**, **notifiche** e **tentativi**
- Raccolta e analisi dei **log**
- Raccolta di **metriche** su tentativi, durata, successo
- Interazione con nuove fonti/strumenti senza reinventare la ruota

Soluzione: orchestratore di pipeline dati

Automatizza i flussi di elaborazione dei dati con una **gestione centralizzata**:

- Pianifica i processi (**scheduling**)
- Gestisce l'**esecuzione** delle attività
- Coordina le **dipendenze** tra attività
- Gestisce **tentativi**, **errori** e **notifiche**
- Monitora e raccoglie **metriche**

Cosa NON fa un orchestratore di pipeline dati

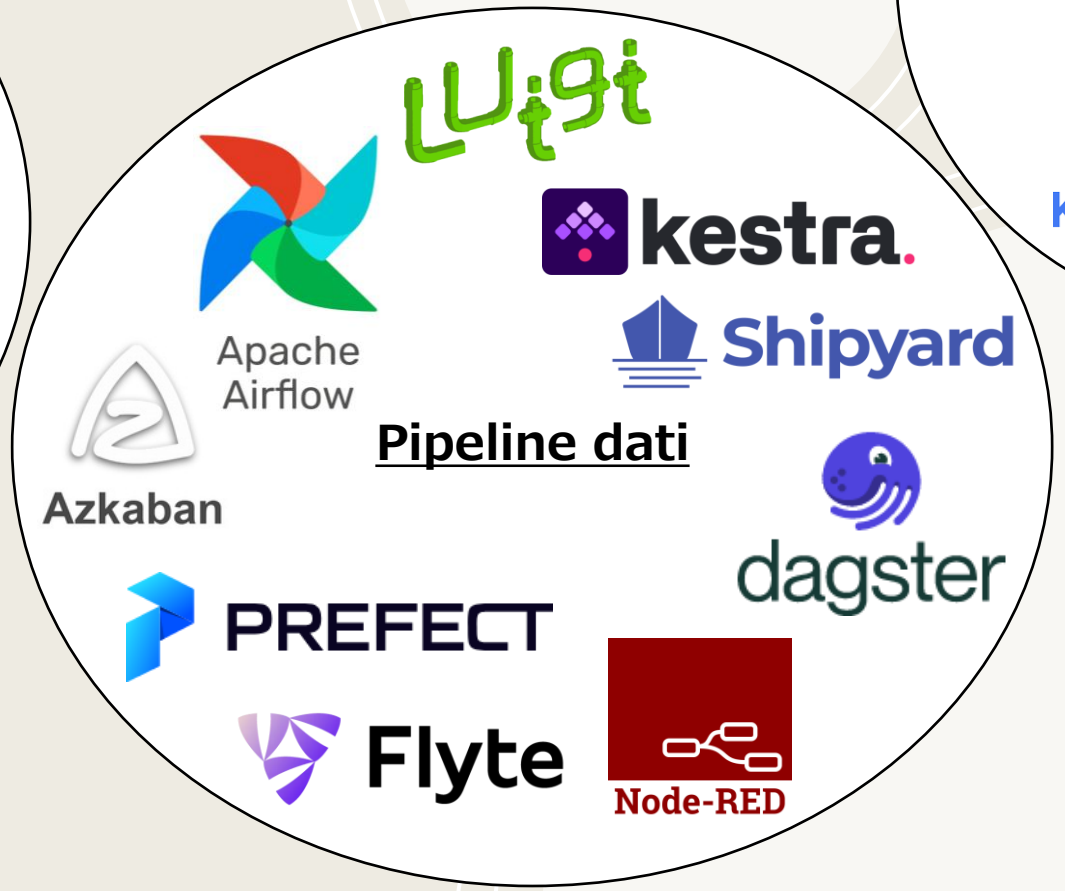
Gestire dati, non **artefatti software** (diverso da CI/CD).

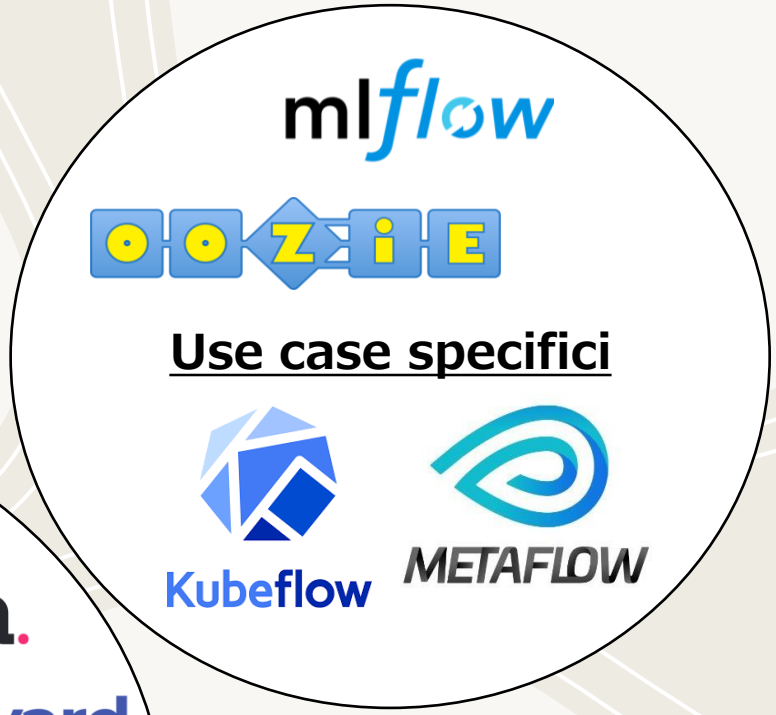
Non è un semplice **message broker** (orchestration, non choreography).

Tipicamente non include i task di basso livello:

- **Persistenza** dei dati da elaborare o elaborati (compito di un DFS / database / data lake / ...)
- **Elaborazione diretta** dei dati (compito di un data processing engine / compute service / ...)

Panoramica degli orchestratori





Apache Airflow

airflow.apache.org

Nato nel 2014 in Airbnb.

Free & Open Source, scritto in Python.

Apache Top-Level Project dal 2019.

Modulare, scalabile, estendibile con plugin, pipeline programmate in **Python** (possibile la generazione dinamica) e parametrizzabili.

Web UI accessibile da browser.

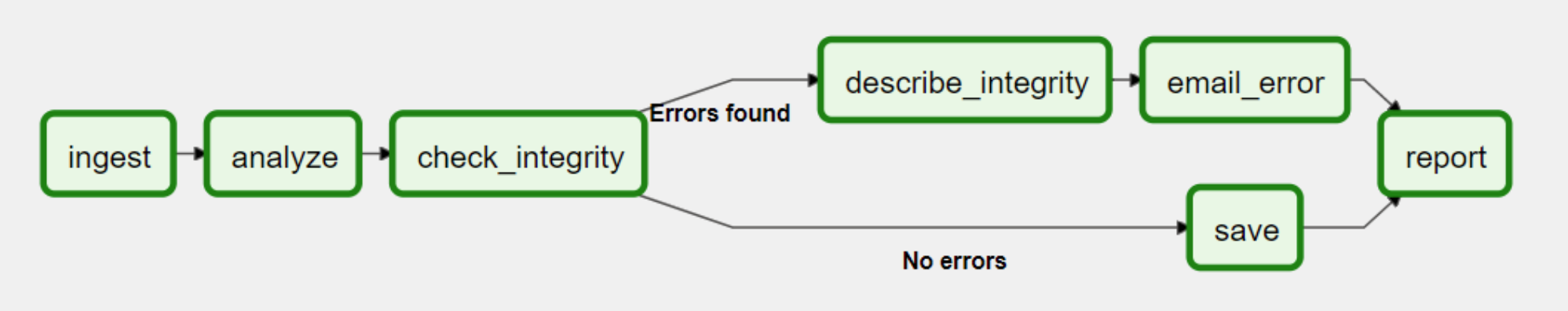
Pensato per **elaborazione in batch**, non in streaming (se non indirettamente).

Apache Airflow

Directed Acyclic Graph (DAG)

Rappresenta una pipeline dati

Ogni nodo («Task») definisce uno step del flusso



Operator

“Modello” di task da libreria

- **Script**: Bash, Python, SQL, ...
- **Interazioni**: Http, FTP, SSH, Cloud (S3, Lambda, EC2, Step functions, ...), Kafka, Spark, Jenkins, ...
- **Notifiche**: Email, Telegram, Slack, ...
- **Container orchestration**: Docker, K8s
- **Flusso**: branching, short-circuiting

Elenco su airflow.apache.org/docs

Sensor

“Modello” di task da libreria per **monitorare**

- Data/ora specifica
- Modifiche al file system
- Modifica/aggiunta oggetti S3 / GCS / ...
- Stream Kafka / DynamoDB / ...
- Messaggi da broker pub-sub
- Polling SQL / HTTP / ...
- ...

Operatori per OSM e GIS

- Strumenti **CLI** come curl, GDAL CLI (ogr2ogr, ...), osm2pgsql, osmium, osmfilter, shp2pgsql, ...: **BashOperator**, **DockerOperator, KubernetesPodOperator**
- **Script Python** con geoPandas, pyosmium, ...: **PythonOperator,** PythonVirtualenvOperator
- **Script** in altri linguaggi: **DockerOperator, KubernetesPodOperator**
- Operatori per Athena / Spark / ... (elenco: airflow.apache.org/docs)
- Operatori custom

Scheduling

Time-based scheduling: utilizzo di «Timetable»:

- Espressioni cron (raw oppure CronTriggerTimetable)
- Elenchi di date+ore (EventsTimetable)

Data-driven («data-aware») **scheduling**:

- «**Dataset**» definito da un URI
- Uno o più task con il dataset in output (opzione «outlet») (es: al termine di una pipeline o in un sensore in ascolto)
- Una o più DAG schedulate all'aggiornamento dell'output

Connection

Gestione centralizzata delle connessioni verso l’esterno (SSH / HTTP / FTP / DB / servizi / ...)

Ogni connessione ha un ID utilizzabile nelle pipeline

Permette di evitare l’hardcoding delle connessioni nelle pipeline

Approccio con context manager

```
import datetime
from airflow import DAG
from airflow.operators.empty import EmptyOperator
with DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
):
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")
    task1 >> task2
```

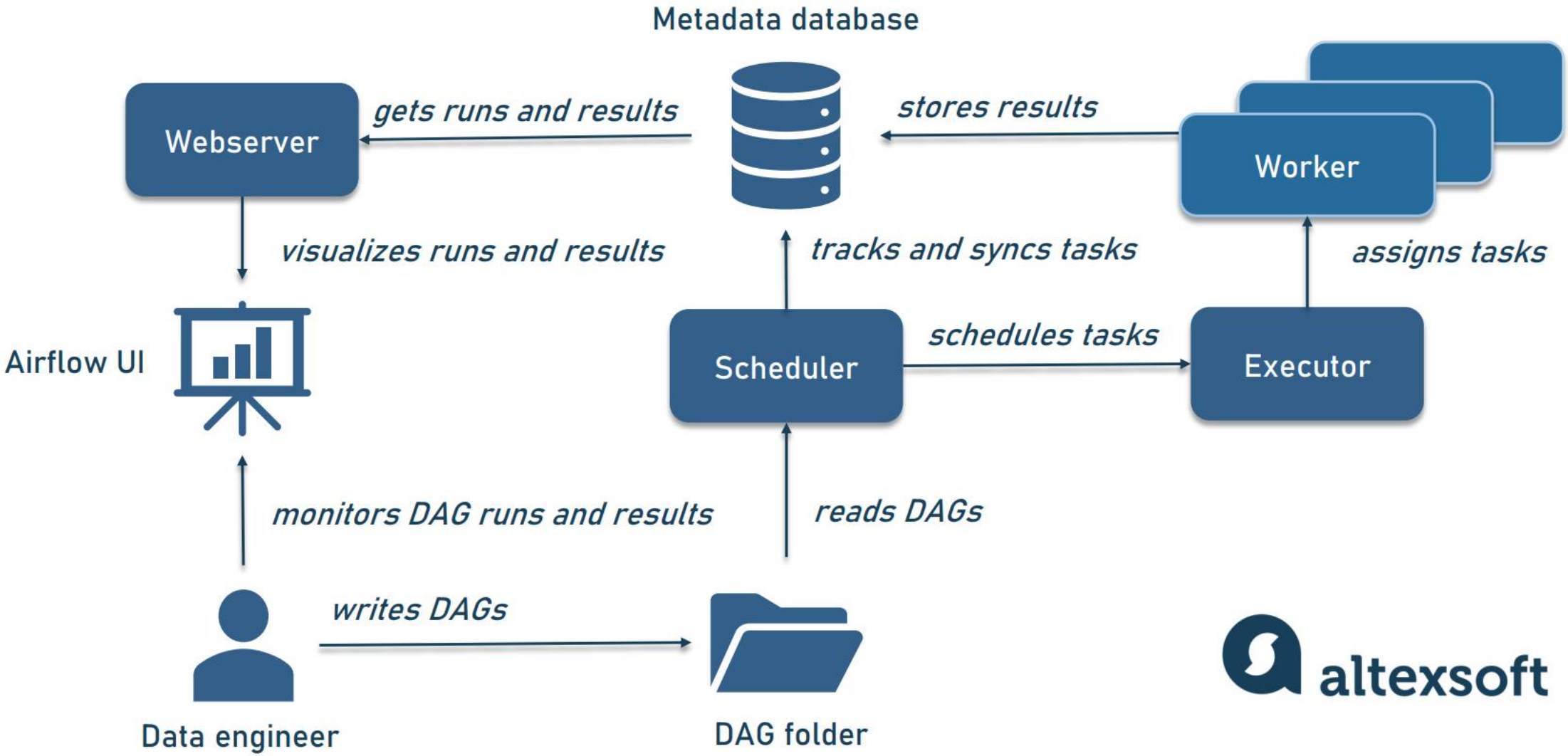
Approccio OOP standard

```
import datetime
from airflow import DAG
from airflow.operators.empty import EmptyOperator
my_dag = DAG(
    dag_id="my_dag_name",
    start_date=datetime.datetime(2021, 1, 1),
    schedule="@daily",
)
task1 = EmptyOperator(task_id="task1", dag=my_dag)
task2 = EmptyOperator(task_id="task2", dag=my_dag)
task1 >> task2
```

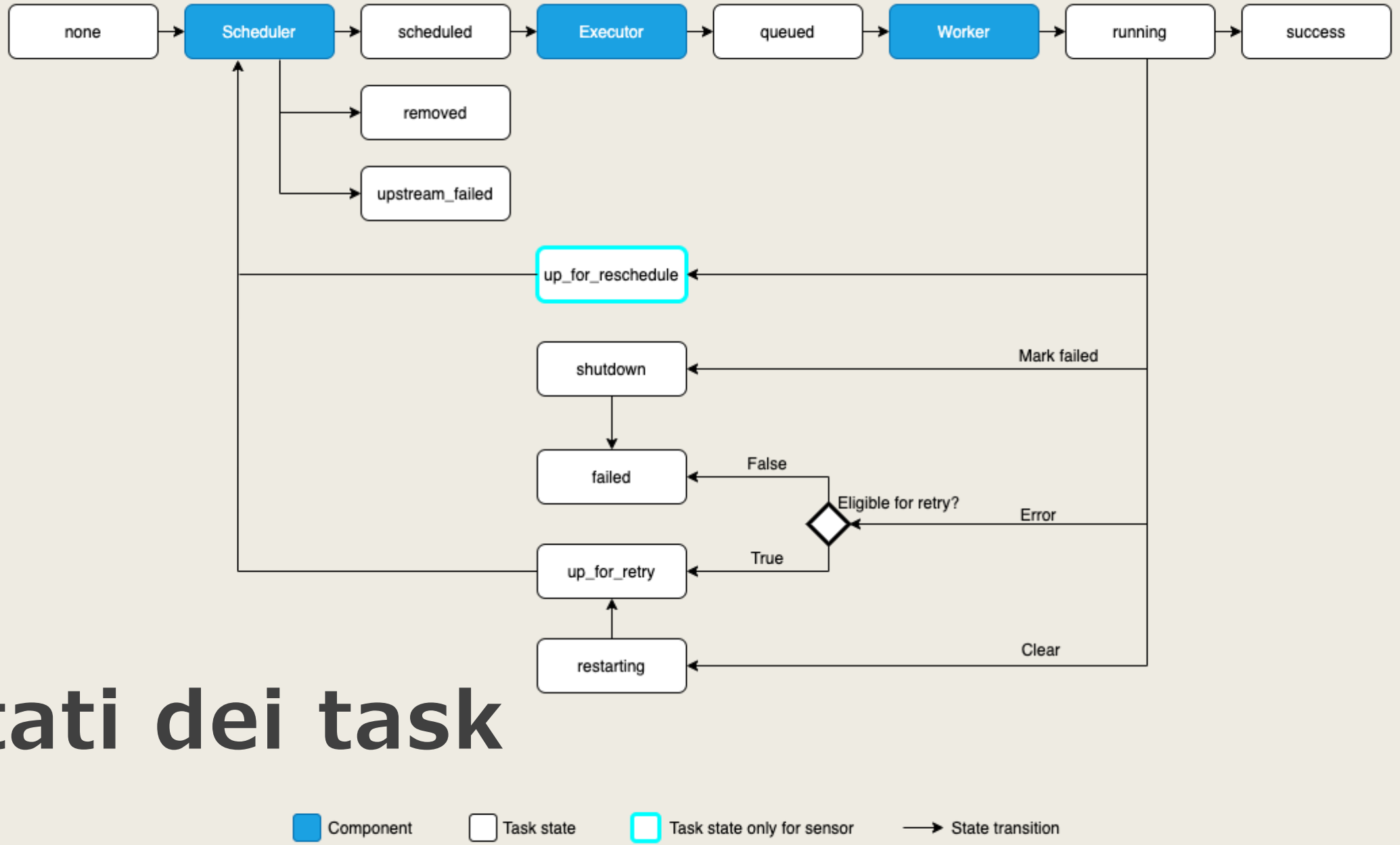
Approccio funzionale (TaskFlow)

```
import datetime
from airflow.decorators import dag
from airflow.operators.empty import EmptyOperator
@dag(start_date=datetime.datetime(2021, 1, 1), schedule="@daily")
def generate_dag():
    task1 = EmptyOperator(task_id="task1")
    task2 = EmptyOperator(task_id="task2")
    task1 >> task2
generate_dag()
```

Architettura



Stati dei task



Executor

Meccanismo di implementazione dell'architettura e esecuzione:

- **Sequential** Executor: CLI, sqlite, no parallelismo

```
$ pip install apache-airflow[postgres]
```

... oppure ...

```
$ pip install apache-airflow apache-airflow-providers-postgres
```

… e poi ...

```
$ airflow standalone
```

Altri Executor

- **Local** Executor: CLI, MySQL/PostgreSQL, singolo worker
- **Celery** Executor: CLI/Docker, MySQL/PostgreSQL, Celery (task queue), RabbitMQ/Redis/..., permette di scalare i worker, template docker-compose.yml disponibile)
- Dask Executor
- **Kubernetes** Executor: MySQL/PostgreSQL (suggerito in produzione, Helm chart disponibile)
- Celery Kubernetes Executor

Servizi managed

- Azure Data Factory - Managed Airflow
- Amazon Web Services - MWAA
- Google Cloud - Cloud Composer
- Astronomer

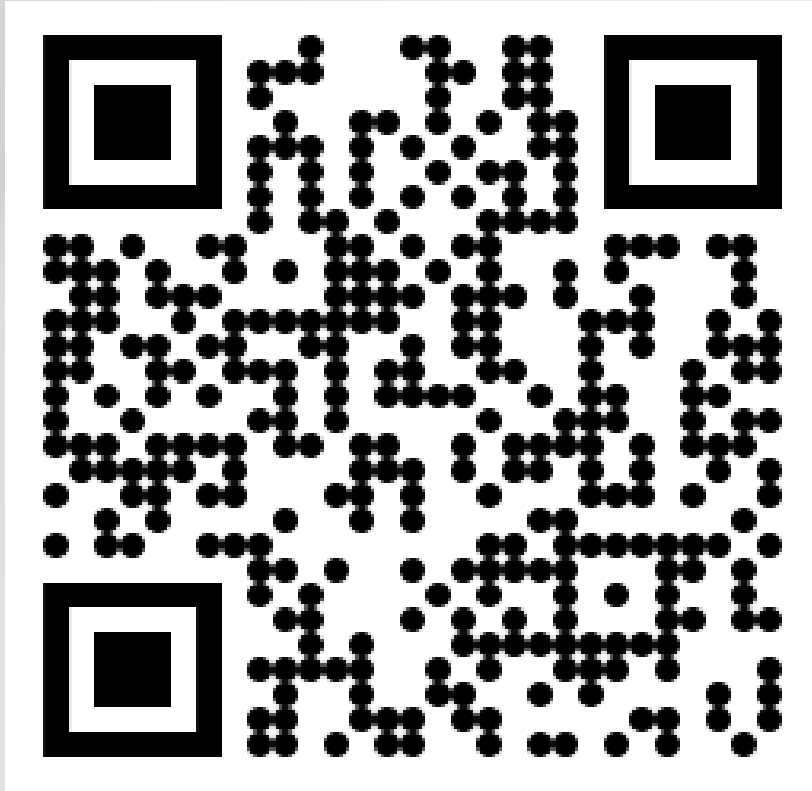
Best practices

Task idempotenti: stesso input => stesso output, senza duplicati (es: UPSERT invece di INSERT).

Preferire gli operator per container a quelli per script e Python per migliorare la scalabilità.

Lazy loading: usare import, risorse bloccanti e variabili solo nella fase di esecuzione piuttosto che nel codice di definizione.

Grazie per l’attenzione!



dsantini.it/airflow.pdf