Apache Airflow

APACHE AIRFLOW

Orchestration de Workflows de Données

01	INTRODUCTION	Introduction et Contexte
2	CONCEPTS FONDAMENTAUX	DAG , Tasks , Operators, Dependencies, Scheduler et Sensors
03	ARCHITECTURE ET COMPOSANTS	Architecture globale d'Airflow et Composants principaux
04	DÉMO	Déploiement Apache Airflow avec docker compose
05	CONCLUSION	Bonnes pratiques

Introduction

01

COMPLEXITÉ CROISSANTE

Pipelines de données de plus en plus nombreux et interdépendants

2

PLANIFICATION DIFFICILE

Gestion des dépendances et des horaires d'exécution complexe

03

SOURCES MULTIPLES

Intégration de données provenant de systèmes hétérogènes

04

GESTION DES ÉCHECS

Monitoring, retry, alertes et récupération après erreur

05

QUESTIONS

Comment orchestrer efficacement tous ces workflows ?

Apache Airflow

Plateforme open-source d'orchestration de workflows permettant de programmer, planifier et monitorer des pipelines de données complexes

01

PLANIFICATION

Scheduling automatique des tâches

2

ORCHESTRATION ET MONITORING

Gestion des dépendances entre tâches, Visualisation et suivi en temps réel

03

PRINCIPE CLÉ

Les workflows sont définis en Python sous forme de DAGs (Directed Acyclic Graphs)

Airflow et historique

01

2014

Création chez Airbnb par Maxime Beauchemin
Besoin d'orchestrer les pipelines de données croissants

2

2016

Open-source et rejoint l'Apache Incubator
Adoption rapide par la communauté data

03

2019

Projet Top-Level Apache
Maturité et reconnaissance officielle

04

AUJOURD'HUI

Leader de l'orchestration de workflows
Utilisé par des milliers d'entreprises dans le monde

Gestion intelligente des échecs avec stratégies de retry configurables

RETRY AUTOMATIQUE

SCALABILITÉ ET EXTENSIBILITÉ

Scalabilité
Architecture distribuée supportant des milliers de tâches.
Écosystème riche de providers et création d'opérateurs custom

GESTION DES DÉPENDANCES

Ordre d'exécution automatique basé sur les relations entre tâches

INTERFACE INTUITIVE

UI web complète pour monitoring, logs et débogage

DÉFINITION AS CODE

Workflows versionnés en Python, testables et maintenables

Concepts fondamentaux

Concepts fondamentaux

DAG - Directed Acyclic Graph

LE CŒUR D'AIRFLOW	DIRECTED	ACYCLIC	GRAPH	UN DAG
un graphe orienté acyclique représentant un workflow	Les tâches s'exécutent dans un ordre précis, défini par des flèches	Pas de boucles : impossible de revenir à une tâche précédente	Pas de boucles : impossible de revenir à une tâche précédente	Un workflow complet (exemple : pipeline ETL quotidien)

Concepts fondamentaux

DAG : Définition d'un DAG en Python

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

# Définition du DAG
with DAG(
    dag_id='exemple_etl',
    start_date=datetime(2024, 1, 1),
    schedule='@daily',  # Exécution quotidienne
    catchup=False,
    tags=['example', 'etl']
) as dag:

    # Les tâches seront définies ici
    pass
```

dag_id

Identifiant unique du workflow

start_date

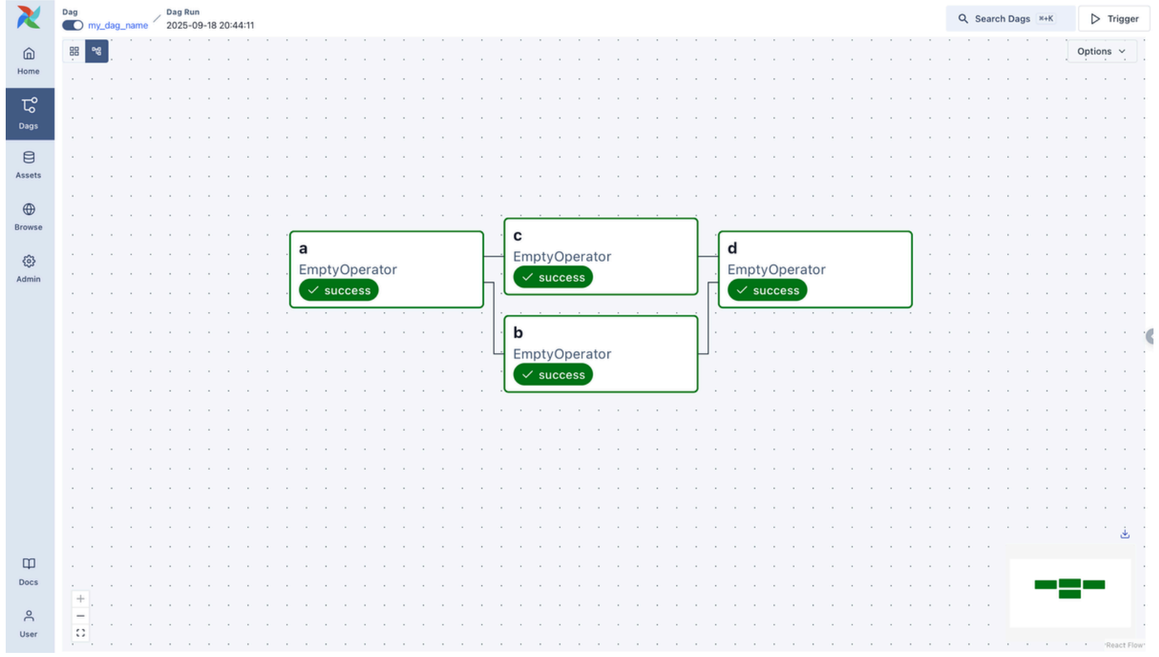
Date de début d'exécution

schedule

Fréquence d'exécution (cron ou preset)

tags

Étiquettes pour organiser les DAGs



Concepts fondamentaux

Tasks - Les unités de travail

> Chaque nœud du DAG représente une tâche à exécuter

> 💡 **Task = Action concrète** (ex: requête SQL, appel API, script Python)

Caractéristiques

- Unité atomique d'exécution
- Possède un état (success, failed...)
- Peut être retentée en cas d'échec
- Génère des logs pour le débogage

États possibles

🔵 **queued** - En attente

🔵 **running** - En cours

🟢 **success** - Réussie

🔴 **failed** - Échouée

🟠 **skipped** - Ignorée

Concepts fondamentaux

Operators - Les types de tâches

Templates prédéfinis pour créer des tâches spécifiques

PythonOperator

Exécute une fonction Python

```
PythonOperator(
  task_id='...',
  python_callable=ma_fonction
)
```

BashOperator

Exécute une commande bash

```
BashOperator(
  task_id='...',
  bash_command='echo hello'
)
```

PostgresOperator

Exécute une requête SQL

```
PostgresOperator(
  task_id='...',
  sql='SELECT * FROM...'
)
```

EmailOperator

Envoie un email

```
EmailOperator(
  task_id='...',
  to='user@example.com'
)
```

HttpOperator

Appel API HTTP/REST

```
HttpOperator(
  task_id='...',
  endpoint='/api/data'
)
```

Et bien plus...

S3, GCS, Spark, Docker, Kubernetes...

+ Création d'operators custom

Concepts fondamentaux

Operators: Création de tâches avec des Operators

```python
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator

def extract_data():
    print("Extraction des données...")
    return "data_extracted"

def transform_data():
    print("Transformation des données...")
    return "data_transformed"

# Création des tâches
extract = PythonOperator(
    task_id='extract',
    python_callable=extract_data
)

transform = PythonOperator(
    task_id='transform',
    python_callable=transform_data
)

load = BashOperator(
    task_id='load',
    bash_command='echo "Loading data to database"'
)
```

💡 **Chaque Operator crée une Task dans le DAG avec un task_id unique**

Concepts fondamentaux

Dependencies - Définir l'ordre d'exécution

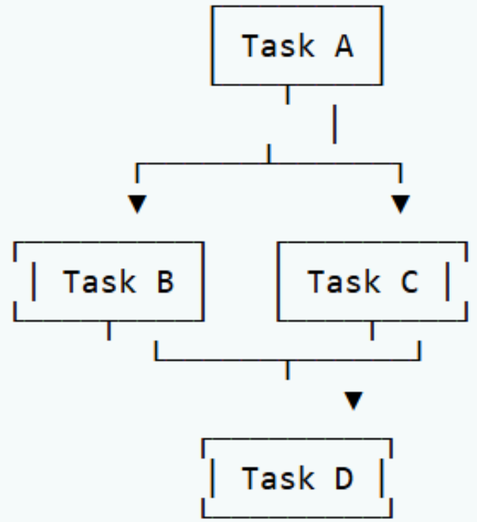
Créer les relations entre les tâches

Syntaxe avec >>

```
# task1 avant task2
task1 >> task2

# Chaîne de tâches
task1 >> task2 >> task3

# task1 avant task2 ET task3
task1 >> [task2, task3]
```

Syntaxe avec <<

```
# task2 après task1
task2 << task1

# task3 après task1 ET task2
task3 << [task1, task2]

# Équivalent à >>
[task2, task3] << task1
```



Code : task_a >> [task_b, task_c] >> task_d

Concepts fondamentaux

Scheduler - Le chef d'orchestre

Composant qui déclenche et surveille les DAGs

Rôles du Scheduler

- Parse les fichiers Python pour trouver les DAGs
- Déclenche les DAGs selon leur schedule
- Soumet les tâches prêtes à l'Executor
- Surveille l'état des tâches en cours
- Gère les retry en cas d'échec

Types de Schedule

@daily - Tous les jours à minuit

@hourly - Toutes les heures

@weekly - Tous les dimanches

'0 8 * * *' - Expression cron (8h)

None - Déclenchement manuel

💡 **Le Scheduler peut être répliqué pour la haute disponibilité (HA)**

Concepts fondamentaux

Sensors - Attendre des conditions

> Tâches spéciales qui attendent qu'une condition soit remplie

FileSensor

Attend qu'un fichier existe

```
FileSensor(
  task_id='wait_file',
  filepath='/data/file.csv'
)
```

HttpSensor

Attend qu'une API réponde

```
HttpSensor(
  task_id='wait_api',
  endpoint='/health'
)
```

SqlSensor

Attend qu'une requête retourne un résultat

```
SqlSensor(
  task_id='wait_data',
  sql='SELECT COUNT(*)'
)
```

ExternalTaskSensor

Attend qu'un autre DAG se termine

```
ExternalTaskSensor(
  task_id='wait',
  external_dag_id='...'
)
```

poke_interval	timeout	mode
Temps entre vérifications	Temps max d'attente	poke / reschedule

> ⚠️ **Les sensors vérifient périodiquement et peuvent timeout**

Concepts fondamentaux

DAG complet : Pipeline ETL

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator
from airflow.sensors.filesystem import FileSensor
from datetime import datetime, timedelta

default_args = {
    'owner': 'data_team',
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
}

with DAG(
    dag_id='etl_pipeline',
    default_args=default_args,
    start_date=datetime(2024, 1, 1),
    schedule='@daily',
    catchup=False,
) as dag:

    wait_for_file = FileSensor(
        task_id='wait_for_file',
        filepath='/data/input.csv',
        poke_interval=30
    )

    extract = PythonOperator(
        task_id='extract',
        python_callable=lambda: print("Extract")
    )

    transform = PythonOperator(
        task_id='transform',
        python_callable=lambda: print("Transform")
    )

    load = BashOperator(
        task_id='load',
        bash_command='echo "Load to DB"'
    )

    notify = PythonOperator(
        task_id='notify',
        python_callable=lambda: print("Notify")
    )

    # Dépendances
    wait_for_file >> extract >> transform >> load >> notify
```

Points clés

- 5 tâches différentes
- Sensor au début
- Configuration retry (3x, 5 min)
- Exécution quotidienne

Flux d'exécution

1. Attendre le fichier
2. Extraire les données
3. Transformer
4. Charger en base
5. Notifier

Concepts fondamentaux

Récapitulatif des Concepts

Operators

Templates pour créer des tâches

Dependencies

Ordre d'exécution avec >> et <<

Scheduler

Chef d'orchestre qui déclenche tout

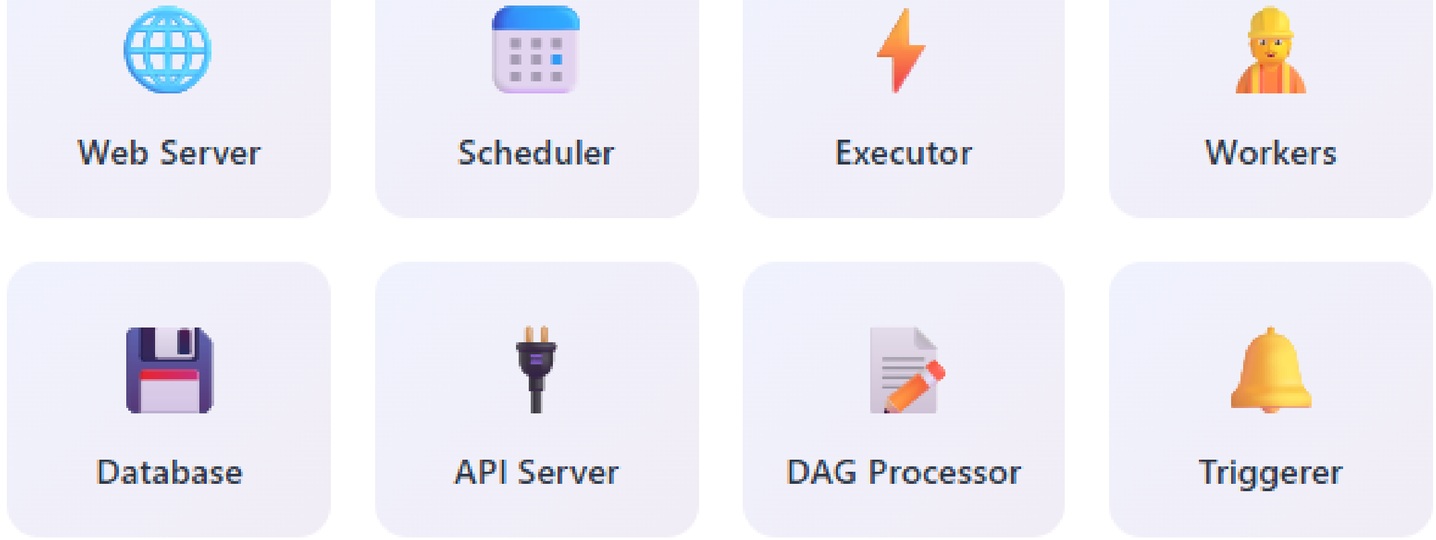
Sensors

Tâches qui attendent des conditions

🚀 **Tous ces concepts se combinent pour créer des workflows puissants et maintenables !**

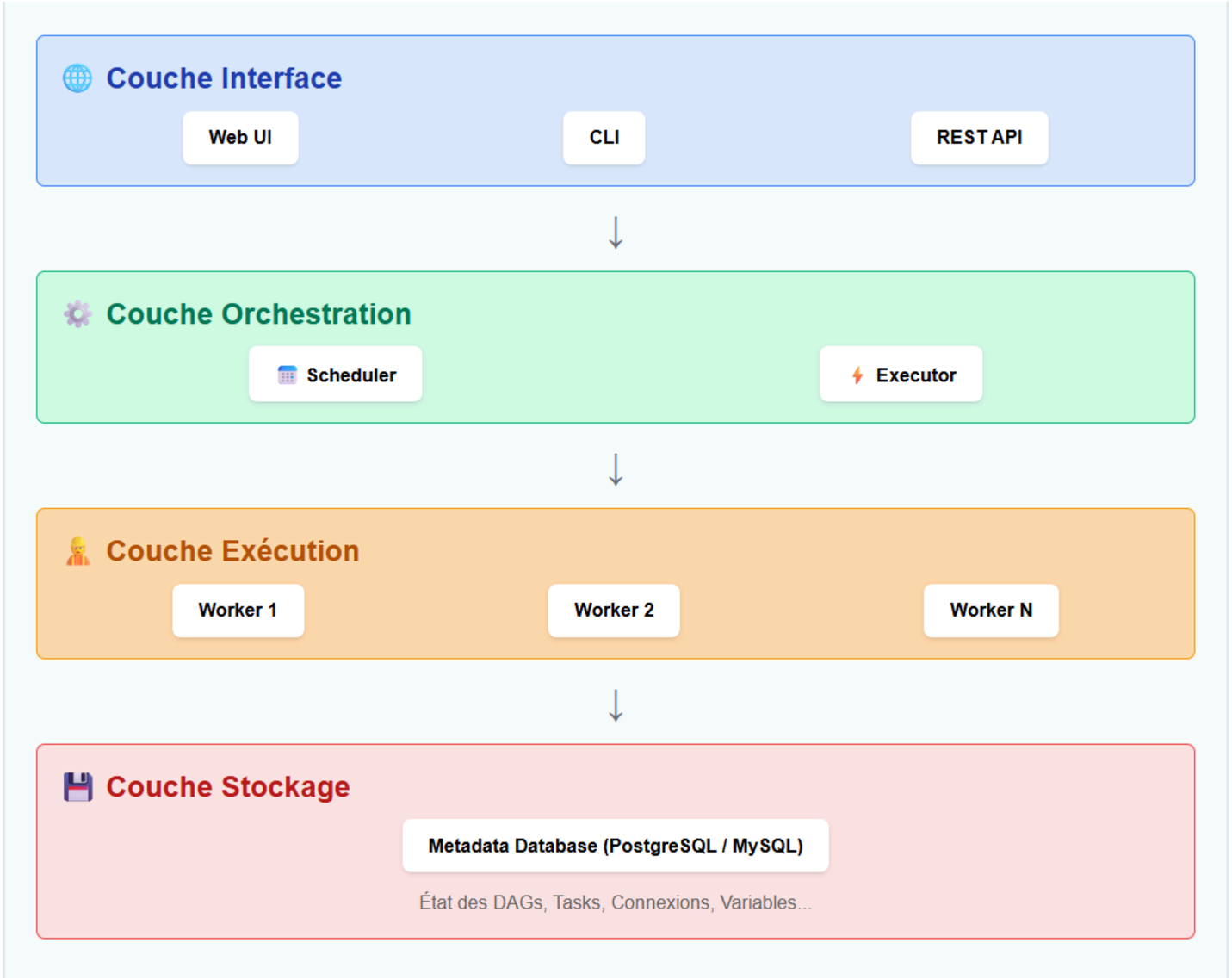
Architecture d'Airflow

Les composants airflow



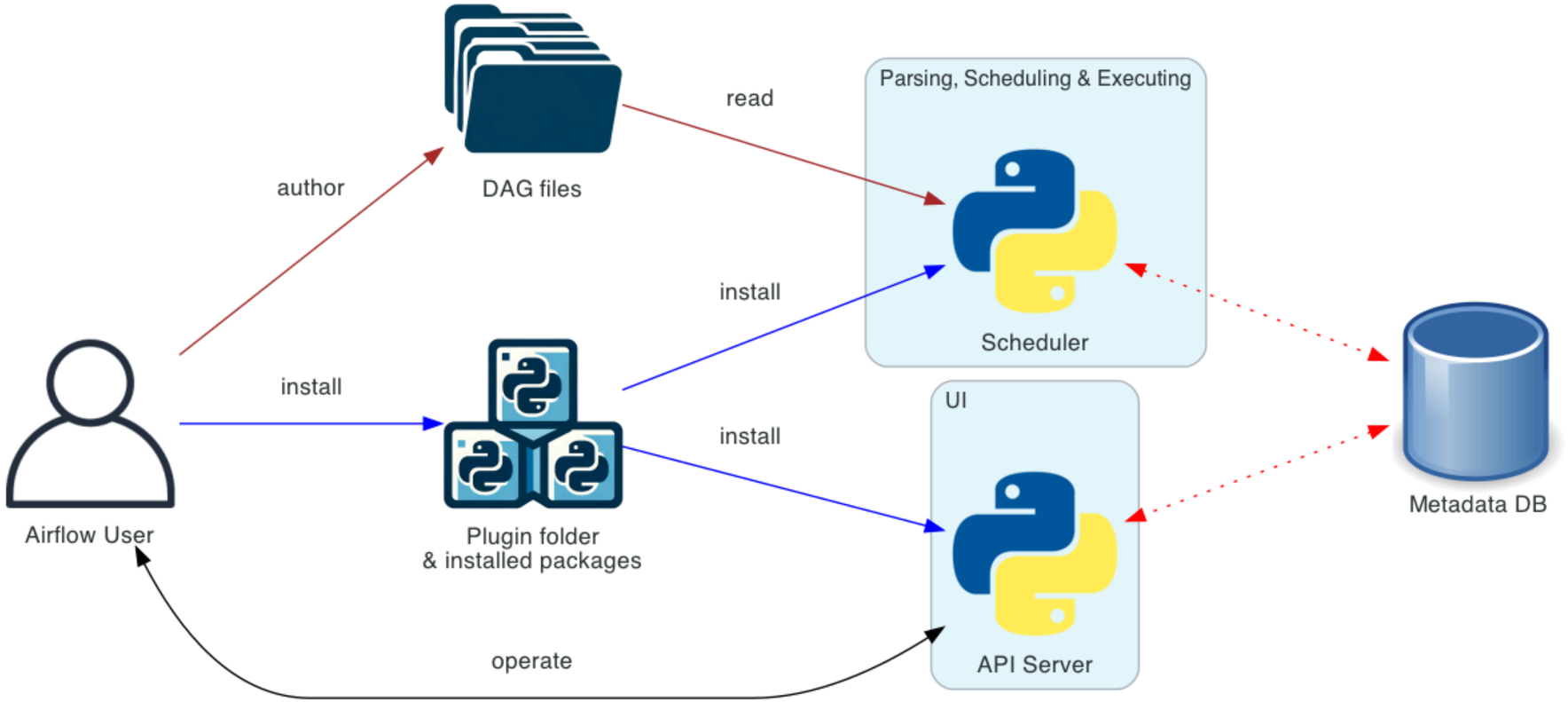
Architecture d'Airflow

Vue d'ensemble de l'Architecture



Architecture d'Airflow

Vue d'ensemble de l'Architecture



https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html

Architecture d'Airflow

Web Server - Interface utilisateur

Fonctionnalités

- 📊 Visualisation des DAGs et leur état
- 🔍 Monitoring en temps réel
- 📝 Consultation des logs
- ▶️ Déclenchement manuel de DAGs
- ⚙️ Gestion connexions et variables
- 👥 Gestion utilisateurs et permissions

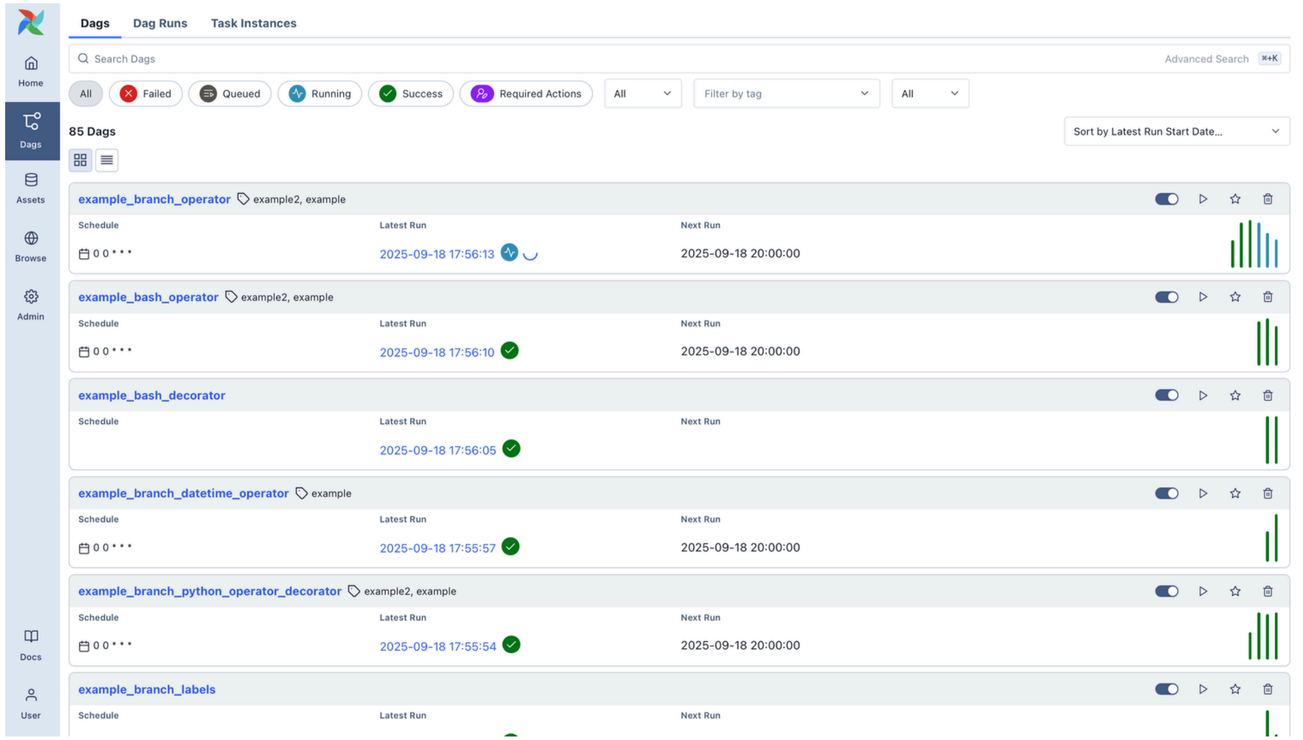
Vues principales

DAGs View
Liste de tous les DAGs disponibles

Graph View
Visualisation graphique du workflow

Tree View
Historique des exécutions en arbre

Gantt Chart
Timeline d'exécution des tâches



Technologie : FastApi (Python) avec base de données pour l'authentification

Architecture d'Airflow

Scheduler - Le cerveau d'Airflow

Responsabilités

- **1.** Parse les DAGs : Lit les fichiers Python du dossier dags/
- **2.** Déclenche les DAGs : Selon leur schedule défini
- **3.** Soumet les tâches : Envoie les tâches prêtes à l'Executor
- **4.** Monitore l'état : Suit l'exécution et met à jour la DB
- **5.** Gère les retry : Relance automatiquement en cas d'échec

Cycle de vie

1. **Parse DAGs**
   Toutes les X secondes
2. **Crée DAG Runs**
   Selon le schedule
3. **Identifie tasks prêtes**
   Dépendances satisfaites
4. **Envoie à l'Executor**
   Pour exécution

Technologie : FastApi (Python) avec base de données pour l'authentification

Architecture d'Airflow

Executor - Gestionnaire d'exécution.Détermine **COMMENT** les tâches sont exécutées

SequentialExecutor

Usage : Dev/Test uniquement

Mode : Une tâche à la fois

Performance : ⭐

⚠️ **Ne pas utiliser en production**

LocalExecutor

Usage : Petites installations

Mode : Multiprocessing local

Performance : ⭐⭐⭐

✓ **Bon pour démarrer**

CeleryExecutor

Usage : Production distribuée

Mode : Workers distribués

Performance :

⭐⭐⭐⭐⭐

✓ **Scalabilité horizontale**

KubernetesExecutor

Usage : Environnements cloud-native

Mode : Pod K8s par tâche

Performance :

⭐⭐⭐⭐⭐

✓ **Isolation maximale**

Architecture d'Airflow

Executor - Comparaison des Executors

Executor	Parallélisme	Scalabilité	Complexité	Usage
Sequential	❌	❌	Faible	Dev uniquement
Local	✅	⚠️ Limitée	Moyenne	Petits projets
Celery	✅	✅	Élevée	Production
Kubernetes	✅	✅	Très élevée	Cloud-native

> 💡 **Conseil :** Commencez avec LocalExecutor, puis passez à Celery quand vos besoins augmentent

Architecture d'Airflow

Workers - Exécution des tâches.Processus qui exécutent réellement les tâches

Caractéristiques

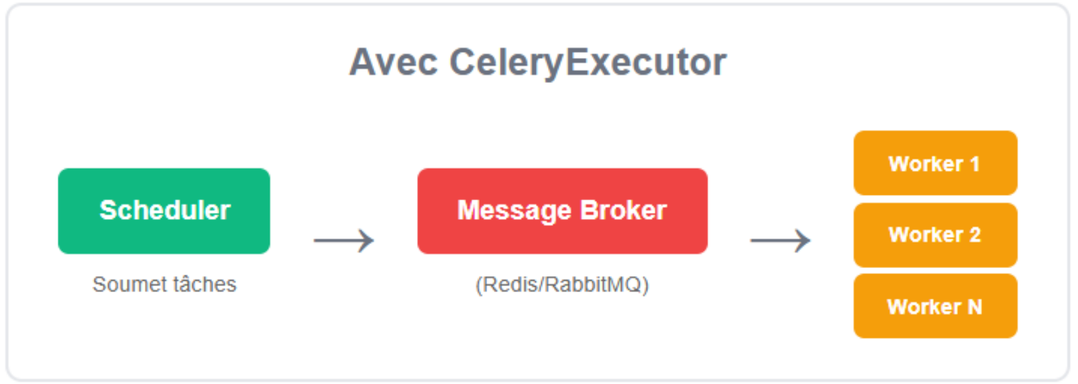
- Reçoivent les tâches de l'Executor
- Exécutent le code des operators
- Rapportent l'état d'exécution
- Génèrent les logs
- Peuvent être sur différentes machines

Architecture distribuée

Machine 1
Workers 1-5 (ETL tasks)

Machine 2
Workers 6-10 (ML tasks)

Machine 3
Workers 11-15 (API tasks)



Architecture d'Airflow

Metadata Database - La mémoire.Stocke tout l'état et l'historique d'Airflow

- 📋 **DAGs :** Définitions et configurations
- 🔄 **DAG Runs :** Instances d'exécution
- ✓ **Task Instances :** État de chaque tâche
- 🔗 **Connexions :** Credentials DB, APIs...
- ⚙️ **Variables :** Configuration globale
- 👥 **Users & Permissions :** Sécurité
- 📊 **Logs metadata :** Références aux logs

Bases supportées

PostgreSQL
✅ Recommandé pour la production

MySQL
✅ Alternative viable

SQLite
⚠️ Dev uniquement, pas de parallélisme

Conseil : PostgreSQL est le choix le plus courant et stable

Importance critique

La base de données est le cœur du système. Tous les composants (Scheduler, Web Server, Workers) lisent et écrivent dans cette DB. **Sa disponibilité et ses performances sont essentielles.**

Architecture d'Airflow

API Server

Rôle Principal

Interface REST API permettant l'interaction programmatique avec Airflow sans passer par l'interface web.

Fonctionnalités

- Endpoints RESTful pour toutes les opérations Airflow
- Déclenchement de DAGs via appels HTTP
- Consultation de l'état des tâches et DAG runs
- Gestion des connexions et variables via API
- Intégration avec des outils CI/CD
- Automatisation et orchestration externe

Sécurité

Support de l'authentification (Basic Auth, JWT, OAuth) et autorisation basée sur les rôles pour un accès sécurisé.

Use Cases

Idéal pour l'intégration avec des systèmes externes, les pipelines CI/CD, et l'automatisation d'opérations Airflow.

Architecture d'Airflow

DAG Processor

▶ Rôle Principal

Processus dédié à l'analyse et au traitement des fichiers DAG Python pour extraire les définitions de workflows.

▶ Fonctionnalités

- Parsing des fichiers Python dans le dossier dags/
- Extraction de la structure des DAGs et leurs tâches
- Validation de la syntaxe et de la cohérence des DAGs
- Détection des erreurs de configuration
- Mise à jour de la représentation des DAGs dans la base
- Support du parsing parallèle pour améliorer les performances

▶ Avantages (Airflow 2.0+)

Séparation du parsing DAG du Scheduler pour améliorer la stabilité et les performances. Réduit la charge du Scheduler et permet un traitement plus rapide des changements de DAG.

▶ Configuration

Peut être configuré pour s'exécuter en tant que processus séparé ou intégré au Scheduler selon les besoins de performance.

Architecture d'Airflow

Triggerer

Rôle Principal

Composant asynchrone qui gère les tâches nécessitant des attentes longues sans bloquer les workers (Airflow 2.2+).

Fonctionnalités

- Gestion des opérateurs avec mode "deferrable"
- Attente asynchrone d'événements externes
- Libération des slots de workers pendant les attentes
- Monitoring de conditions de déclenchement (triggers)
- Reprise automatique des tâches après événement
- Support de milliers d'attentes concurrentes

Cas d'Usage

- Attente de complétion de jobs externes (EMR, Databricks)
- Polling de fichiers ou APIs avec longs délais
- Sensors avec temps d'attente élevé
- Tâches déclenchées par événements externes

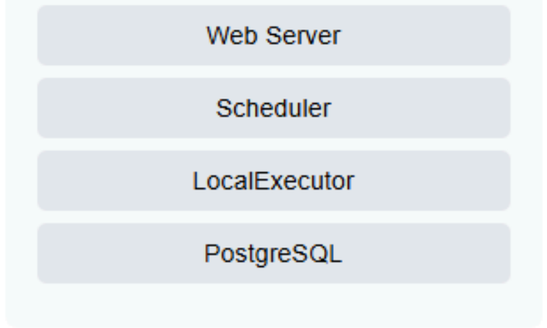
Avantage

Optimisation massive des ressources: un worker qui attendait peut maintenant exécuter d'autres tâches pendant que le Triggerer surveille l'événement.

Architecture d'Airflow

Modes de déploiement

Single Node

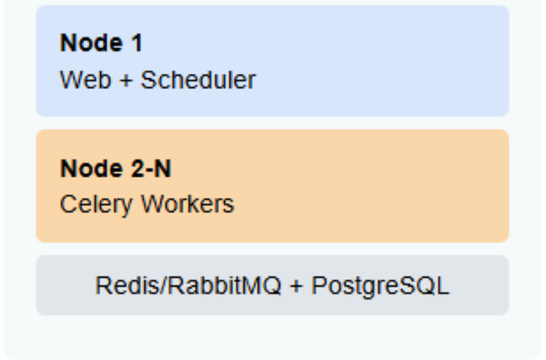


Usage : Dev, petits projets

Avantages : Simple à gérer

Limites : Pas de HA

Multi-Node Celery

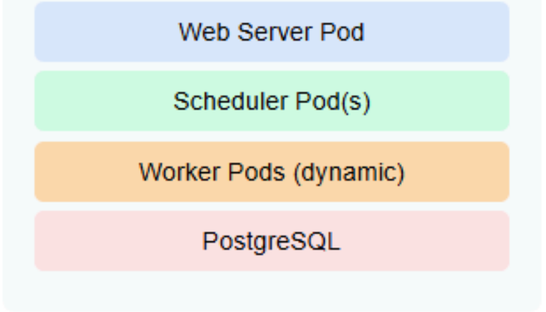


Usage : Production

Avantages : Scalable

Limites : Complexité

Kubernetes



Usage : Cloud-native

Avantages : Auto-scaling, HA

Limites : Très complexe

Choix selon le contexte

Petite équipe

Single Node + LocalExecutor

Production

Multi-Node + CeleryExecutor

Enterprise / Cloud

Kubernetes + KubernetesExecutor

Architecture d'Airflow

Recapitulatif

Flux d'Exécution

- **DAG Processor** analyse les fichiers Python et extrait les DAGs
- **Scheduler** lit les DAGs depuis la Database et planifie les tâches
- **Executor** reçoit les tâches et les distribue aux Workers
- **Workers** exécutent les tâches et renvoient les résultats
- **Triggerer** gère les tâches en attente d'événements externes
- **Database** centralise tous les états et métadonnées
- **Web Server** affiche l'état via l'interface utilisateur
- **API Server** permet l'accès programmatique à toutes les fonctionnalités

Communication

Tous les composants communiquent via la Database comme source unique de vérité. Le Web Server et l'API Server consultent la Database pour afficher les informations. Le Scheduler et les Executors mettent à jour la Database avec les changements d'état.

Scalabilité

Chaque composant peut être scalé indépendamment selon les besoins: plusieurs Schedulers, Workers distribués, Triggerers multiples, Web Servers derrière un load balancer.

Démonstration Pratique

Apache Airflow en action

LIVE DEMO

Création et exécution d'un DAG réel

Création de DAG

Écrire un workflow pas à pas

Interface Web UI

Explorer les vues principales

Exécution

Monitoring en temps réel

Gestion des erreurs

Retry et débogage

Conclusion