# Cours : Introduction à Apache Airflow et Théorie des DAGs

Cours EFREI 2024-2025

**Yvann VINCENT**
*Machine Learning Engineer*

# Introduction

Ce cours est destiné à fournir une compréhension approfondie d'Apache Airflow en combinant théorie et pratique. Il explore en détail les concepts de DAG (Graphes Acycliques Dirigés) et leur importance dans l'orchestration de workflows. À la fin, les étudiants sauront concevoir, implémenter et gérer des pipelines de données avec Airflow.

—

# 1 Module 1 : Théorie des DAGs et Apache Airflow

## 1.1 La Théorie des Graphes et les DAGs

La **théorie des graphes** est une branche des mathématiques discrètes qui étudie les relations entre des objets au moyen de structures appelées graphes. Ces structures sont particulièrement pertinentes pour modéliser et analyser des processus ordonnés, comme ceux utilisés dans Apache Airflow.

### Définition Formelle d'un Graphe

Un graphe est défini comme un couple $G = (V, E)$, où :

- $V$ est un ensemble non vide de sommets (*vertices*), souvent noté $V = \{v_1, v_2, \ldots, v_n\}$.
- $E$ est un ensemble d'arêtes (*edges*), qui relient des sommets. Une arête est une paire $(u, v)$ où $u, v \in V$.

Un graphe est dit **orienté** si chaque arête a une direction. Dans ce cas, $E \subseteq V \times V$.

**Exemple** :

- $V = \{A, B, C, D\}$.
- $E = \{(A, B), (B, C), (C, D)\}$.
- Le graphe orienté résultant relie les sommets dans un ordre directionnel.

—

### Graphes Orientés Acycliques (DAGs)

Un **graphe orienté acyclique** (**Directed Acyclic Graph**, ou DAG) est un graphe orienté qui ne contient aucun cycle, c'est-à-dire aucun chemin fermé où un sommet pourrait être revisité.

### Propriétés des DAGs :

- **Ordre partiel** : Les sommets d'un DAG peuvent être ordonnés de manière à respecter les dépendances définies par les arêtes.
- **Pas de cycles** : Il n'existe pas de suite d'arêtes $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)$ où un sommet $v_1$ peut être revisité.
- **Représentation des dépendances** : Les DAGs sont idéaux pour modéliser des processus avec des dépendances strictes.

**Exemple Formel :** Si $G = (V, E)$ est un DAG avec $V = \{A, B, C, D\}$ et $E = \{(A, B), (A, C), (B, D), (C, D)\}$, alors :

- $A$ doit être exécuté avant $B$ et $C$.
- $B$ et $C$ doivent être exécutés avant $D$.
- Aucun cycle n'existe dans ce graphe.

—

### 1.2 Applications Pratiques des DAGs

Les DAGs apparaissent dans de nombreux contextes pratiques où des dépendances doivent être respectées :

- **Planification des tâches dans un système d'exploitation** : Les DAGs modélisent l'exécution des tâches avec des contraintes de dépendance, comme les processus qui doivent attendre des fichiers ou des ressources.
- **Analyse des dépendances dans un projet logiciel** : Les outils comme `make` ou `CMake` utilisent des DAGs pour organiser les étapes de compilation en fonction des dépendances entre fichiers source.
- **Orchestration de workflows complexes** : Dans les pipelines de données, les DAGs modélisent les transformations successives appliquées aux données, tout en garantissant que les étapes précédentes sont complétées avant de passer à la suivante.

—

#### Pourquoi les DAGs pour Apache Airflow ?

Airflow utilise les DAGs pour modéliser les workflows car :

- Ils assurent un ordonnancement clair des tâches, sans ambiguïté sur leur ordre d'exécution.
- Les dépendances entre tâches peuvent être représentées explicitement par les arêtes.
- L'absence de cycles garantit que les workflows ne s'exécutent pas indéfiniment.

—

#### Propriétés des DAGs dans les Algorithmes

Les DAGs sont utilisés dans plusieurs algorithmes pour leur capacité à modéliser des processus ordonnés :

- **Chemin critique** : Calculer la durée minimale pour compléter un projet, en identifiant les étapes critiques d'un DAG.
- **Flots dans les réseaux** : Les DAGs servent dans l'analyse des flots maximaux lorsque les graphes sont orientés et acycliques.
- **Optimisation des bases de données** : Les plans d'exécution de requêtes SQL complexes sont souvent représentés sous forme de DAGs.

### 1.3 Introduction à Apache Airflow

**Apache Airflow** est une plateforme open-source permettant de définir, planifier et surveiller des workflows. Les workflows sont modélisés sous forme de DAGs.

#### Caractéristiques principales

- Définition de workflows en Python.
- Gestion centralisée via une interface web.
- Surveillance des workflows avec visualisation des dépendances et des logs.
- Extensibilité avec des opérateurs personnalisés.

#### Composants d'Airflow

- **Webserver** : Interface utilisateur pour surveiller et gérer les workflows.
- **Scheduler** : Responsable de l'exécution des tâches selon le plan défini.
- **Worker** : Exécute les tâches individuelles d'un workflow.
- **Metadata Database** : Stocke les informations sur les DAGs et les tâches.

### 1.4 Concepts Appliqués : Comparaison entre DVC et Apache Airflow

DVC et Apache Airflow sont deux outils couramment utilisés pour orchestrer des workflows de données, mais ils ont des approches et des cas d'utilisation très différents. Tous deux utilisent des concepts de dépendances similaires à ceux des DAGs, mais leur implémentation et leur objectif diffèrent considérablement.

—

#### Utilisation des Graphes dans DVC et Airflow

**DVC :** DVC repose sur une approche orientée par les données et les fichiers. Chaque étape d'un pipeline DVC est définie dans un fichier de configuration (`dvc.yaml`) où les dépendances entre les fichiers sont explicites.

- Les étapes d'un pipeline sont organisées sous forme de graphes directs où chaque nœud représente une étape du pipeline et chaque arête représente une dépendance entre les fichiers produits ou consommés.
- L'objectif principal de DVC est de garantir la reproductibilité des pipelines en versionnant non seulement le code, mais aussi les données et les résultats intermédiaires.
- DVC utilise un ordonnancement déterminé par la résolution des dépendances entre fichiers pour exécuter les étapes de manière séquentielle ou parallèle.

**Apache Airflow :** Apache Airflow, en revanche, utilise des DAGs explicitement définis pour modéliser des workflows. Chaque nœud d'un DAG représente une tâche (souvent exécutée sur un cluster ou une machine distante) et chaque arête représente une dépendance logique entre ces tâches.

- Les dépendances dans un DAG Airflow ne concernent pas directement les fichiers, mais plutôt les étapes d'un processus métier.
- Airflow est conçu pour exécuter des tâches dans des environnements complexes et distribués, comme des clusters Kubernetes ou des fermes de serveurs.
- L'objectif principal d'Airflow est d'orchestrer et de monitorer des workflows complexes, souvent avec des tâches nécessitant des outils ou des environnements hétérogènes.

—

### Différences Clés : DVC vs Airflow

| Caractéristique | DVC | Airflow |
|---|---|---|
| **Dépendances** | Basées sur des fichiers et leur versionnement. | Basées sur des tâches et leurs relations logiques. |
| **Représentation des workflows** | Défini via un fichier YAML (`dvc.yaml`). | Défini dans du code Python en utilisant des objets DAGs. |
| **Exécution** | Locale, centrée sur les données. | Orientée cluster/distribuée, centrée sur les tâches. |
| **Cas d'usage** | Pipelines de données reproductibles pour le machine learning. | Orchestration de workflows complexes et intégrations avec plusieurs outils. |
| **Reproductibilité** | Forte, grâce au suivi des versions des fichiers et des données. | Faible à modérée, dépend de la gestion manuelle des états des tâches. |
| **Exécution parallèle** | Basée sur la disponibilité des dépendances. | Basée sur l'ordonnancement par le Scheduler d'Airflow. |

Table 1: Comparaison entre DVC et Airflow

—

### Choix de l'Outil

Le choix entre DVC et Airflow dépend principalement du cas d'usage :

- **DVC** est recommandé pour les projets où la reproductibilité des pipelines et des données est cruciale, comme dans les projets de machine learning.
- **Airflow** est mieux adapté pour des workflows complexes impliquant plusieurs outils, environnements ou équipes.

En comprenant les similitudes et les différences entre ces outils, il devient plus facile de choisir celui qui correspond le mieux à vos besoins tout en exploitant les principes des DAGs pour modéliser et exécuter vos workflows.

---

# 2 Module 2 : Pratique avec Airflow

## 2.1 Exemple 1 : Installation d'Apache Airflow

### Objectif

Installer Apache Airflow dans un environnement local.

### Étapes

1. Installez Airflow :

   ```
   pip install apache-airflow
   ```

2. Initialisez la base de données Airflow :

   ```
   airflow db init
   ```

3. Créez un utilisateur administrateur :

   ```
   airflow users create --role Admin --username admin --password admin --email admin@example.c
   ```

4. Démarrez le webserver et le scheduler :

   ```
   airflow webserver --port 8080 &
   airflow scheduler &
   ```

**Résultat attendu** : Accédez à l'interface Airflow via http://localhost:8080.

---

## 2.2 Exemple 2 : Création d'un DAG simple

### Objectif

Créer un DAG qui affiche "Hello, Airflow!" à l'aide d'un opérateur Python.

### Code Python

```
from datetime import datetime, timedelta
from airflow.decorators import dag, task

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'email_on_failure': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

@dag(
    default_args=default_args,
```

```
    description='Un DAG simple qui dit Bonjour!',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2023, 1, 1),
    catchup=False,
)
def hello_airflow():
    @task
    def say_hello():
        print("Hello, Airflow!")
    say_hello()

dag = hello_airflow()
```

**Instructions** :

- Placez ce fichier dans le répertoire `dags`.
- Rafraîchissez l'interface Airflow pour voir le DAG.
- Activez et exécutez le DAG.

---

# 3 Module 3 : Concepts Avancés et Cas d'Utilisation

## 3.1 DAGs Multi-Tâches et Dépendances

### Objectif

Créer un DAG avec plusieurs tâches dépendantes.

### Code Exemple

```
from airflow.decorators import dag, task
from datetime import datetime, timedelta

default_args = {
    'owner': 'airflow',
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

@dag(
    default_args=default_args,
    description='Un DAG avec plusieurs tâches',
    schedule_interval=timedelta(days=1),
    start_date=datetime(2023, 1, 1),
    catchup=False,
)
def multi_task_dag():
    @task
    def task_1():
        print("Tâche 1 terminée.")

    @task
    def task_2():
```

```
24          print("Tâche 2 terminée.")
25
26      @task
27      def task_3():
28          print("Tâche 3 terminée.")
29
30      task_1() >> [task_2(), task_3()]
31
32  dag = multi_task_dag()
```

---

## Conclusion

- Les DAGs sont la base de la gestion des workflows dans Airflow.
- Apache Airflow fournit un cadre puissant et flexible pour orchestrer des pipelines ETL.
- La pratique régulière avec des exercices est essentielle pour maîtriser cet outil.

# 4 Module 4 : Scheduling et Execution

## 4.1 Planification des Tâches dans Airflow

### Qu'est-ce que le Scheduling ?

Le **scheduling** consiste à déterminer quand et à quelle fréquence les tâches d'un workflow doivent être exécutées. Dans Airflow, cela est défini au niveau du DAG grâce à deux paramètres principaux :

- **start_date** : La date et l'heure à partir desquelles le DAG commence à être exécuté.
- **schedule_interval** : L'intervalle entre deux exécutions du DAG.

### Expression Cron et Intervalle

Airflow prend en charge plusieurs formats pour définir le *schedule_interval* :

- **timedelta** : Par exemple, `timedelta(days=1)` pour une exécution quotidienne.
- **Expression cron** : Par exemple, `'0 9 * * *'` pour une exécution tous les jours à 9h00.
- **None** : Aucune planification automatique. Le DAG doit être déclenché manuellement.

### Catchup et Backfilling

- **Catchup** : Lorsque le planning est activé et qu'une période de temps s'est écoulée depuis la dernière exécution, Airflow exécute toutes les exécutions manquantes.
- **Backfilling** : Exécution des tâches rétroactivement pour des dates passées, souvent utilisée lors de l'ajout d'un nouveau DAG.

**Exemple de configuration** :

```
@dag(
    default_args=default_args,
    description='Exemple de scheduling',
    schedule_interval='0 12 * * *',  # Tous les jours à 12h00
    start_date=datetime(2023, 1, 1),
    catchup=False,  # Désactivation du rattrapage
)
def scheduled_dag():
    pass
```

---

## 4.2 L'Exécution des Tâches dans Airflow

### Modes d'Exécution

Le mode d'exécution des tâches dans Airflow est déterminé par le paramètre `executor` dans la configuration. Les principaux modes sont :

- **SequentialExecutor** : Exécution séquentielle des tâches, adaptée aux environnements de test.
- **LocalExecutor** : Exécution en parallèle des tâches sur la machine locale.
- **CeleryExecutor** : Exécution distribuée des tâches à l'aide d'un cluster Celery.
- **KubernetesExecutor** : Exécution des tâches dans un cluster Kubernetes.

### Redémarrage et Retentative des Tâches

- **Retries** : Nombre de tentatives de réexécution en cas d'échec, défini dans les arguments par défaut (`default_args`).
- **retry_delay** : Temps d'attente entre deux tentatives.
- **on_failure_callback** : Fonction exécutée en cas d'échec d'une tâche.

**Exemple de configuration de retries** :

```
default_args = {
    'owner': 'airflow',
    'retries': 3,
    'retry_delay': timedelta(minutes=10),
}
```

### État des Tâches

Chaque tâche dans Airflow peut avoir différents états, notamment :

- **queued** : En attente d'exécution.
- **running** : En cours d'exécution.
- **success** : Exécutée avec succès.
- **failed** : Échec de l'exécution.
- **upstream_failed** : Non exécutée en raison de l'échec d'une tâche en amont.

---

## 4.3 Exemple Pratique : Pipeline avec Scheduling et Retry

### Code Python

```python
from datetime import datetime, timedelta
from airflow.decorators import dag, task

default_args = {
    'owner': 'airflow',
    'retries': 2,  # 2 tentatives de réexécution
    'retry_delay': timedelta(minutes=5),
}

@dag(
    default_args=default_args,
    description='Pipeline avec scheduling et retry',
    schedule_interval='@daily',  # Exécution quotidienne
    start_date=datetime(2023, 1, 1),
    catchup=True,  # Activer le rattrapage
)
def advanced_pipeline():
    @task
    def extract():
        print("Extraction des données.")
        return "Données extraites"

    @task
    def transform(data):
        print(f"Transformation des données : {data}")
        return data.upper()

    @task
    def load(transformed_data):
        print(f"Chargement des données transformées : {transformed_data}")

    data = extract()
    transformed_data = transform(data)
    load(transformed_data)

dag = advanced_pipeline()
```

### Instructions

- Ajoutez ce fichier dans le répertoire `dags`.
- Rafraîchissez l'interface Airflow pour visualiser le DAG.
- Exécutez manuellement une instance pour tester les retries.
- Simulez un échec en ajoutant `raise Exception("Test Failure")` dans l'une des tâches.

---

# 5 Module 5 : Concepts Complémentaires

## 5.1 XCom : Communication entre Tâches

**XCom** (Cross-Communication) permet aux tâches de partager des données entre elles. Ces données sont stockées dans la base de métadonnées d'Airflow.

### Exemple avec XCom

```python
from airflow.decorators import dag, task

@dag(
    schedule_interval=None,
    start_date=datetime(2023, 1, 1),
    catchup=False,
)
def xcom_example():
    @task
    def push_data():
        return {"key": "value"}

    @task
    def pull_data(data):
        print(f"Données reçues : {data}")

    data = push_data()
    pull_data(data)

dag = xcom_example()
```

---

# Conclusion

- Le scheduling et l'exécution sont des piliers fondamentaux pour orchestrer des pipelines avec Airflow.
- Les DAGs offrent une représentation puissante des workflows avec des mécanismes avancés comme les retries, XCom, et le catchup.
- Apache Airflow est extensible et s'adapte à divers cas d'utilisation, du local au distribué.