Airflow

Guide du SDK des tâches Apache Airflow 2.7+

Le **Task SDK d'Apache Airflow** (introduit en 2.7 et stabilisé en 3.0) fournit une interface Pythonique pour définir des workflows. Le SDK découple l'écriture des DAGs des composants internes d'Airflow, offrant ainsi une API stable et rétrocompatible[1].

En pratique, cela signifie que votre code DAG peut être plus clair et compatible avec les futures versions. Toutes les importations principales proviennent désormais de `airflow.sdk` (par ex. `from airflow.sdk import DAG, dag, task, asset`) plutôt que des modules internes[2].

L'utilisation des décorateurs du SDK (`@dag`, `@task`, `@asset`, etc.) transforme de simples fonctions Python en DAGs et tâches Airflow

---

Définir des DAGs et des tâches avec @dag, @task, @asset

- Utilisez `@dag` pour transformer une fonction Python en DAG.
- À l'intérieur de la fonction décorée, chaque appel de fonction imbriquée décorée avec `@task` devient une tâche du DAG

Exemple :

```
from airflow.sdk import dag, task
import pendulum

@dag(schedule="@hourly", start_date=pendulum.datetime(2025, 1, 1), catc
hup=False, tags=["example"])
def weather_pipeline(city: str = "London"):
    @task(multiple_outputs=True)
    def fetch_weather(city: str) → dict:
        """Récupère les données météo depuis une API publique."""
```

```
        import requests
        url = f"https://api.weather.com/v1/{city}/conditions"
        res = requests.get(url)
        data = res.json()
        # Extraire la température et l'humidité de la réponse de l'API
        return {"temp": data["temp"], "humidity": data["humidity"]}

    @task
    def format_report(temp: float, humidity: float) -> str:
        """Met en forme les données météo en une chaîne de rapport."""
        return f"Météo à {city} : Temp={temp}°C, Humidité={humidity}%"

    # Les dépendances entre tâches sont définies par le flux de données (vo
ir section suivante)
    weather_data = fetch_weather(city)              # XComArg de fetch_weath
er
    report = format_report(weather_data["temp"],
                           weather_data["humidity"])

# Instancier le DAG
weather_pipeline = weather_pipeline()
```

Ici, **fetch_weather** et **format_report** deviennent des tâches Airflow.

Le décorateur `@dag` enregistre **weather_pipeline** comme un DAG (avec son calendrier, sa date de début, etc.), et chaque fonction décorée avec `@task` est une tâche dans ce DAG

Dans du code de production, vous utiliseriez de la même manière `@dag` pour encapsuler les fonctions de tâches et les paramètres (planification, arguments par défaut, tags, etc.).

Vous pouvez également utiliser `with DAG(...) as dag:` et des opérateurs standards à l'intérieur, mais le style `@dag` / `@task` produit généralement un code plus concis et plus lisible.

Pour les actifs de données

Utilisez `@asset` (similaire à `@dag`) pour déclarer un DAG qui produit un jeu de données persistant unique.

Exemple :

```
from airflow.sdk import asset
import pandas as pd

@asset(uri="s3://my-bucket/raw-weather.csv", schedule="@daily")
def raw_weather_data() → None:
    """Télécharger et enregistrer les données météo brutes dans S3."""
    df = pd.read_csv("https://example.com/weather/today.csv")
    df.to_csv("/tmp/raw-weather.csv", index=False)
```

Cette simple définition crée automatiquement un DAG Airflow (ID `raw_weather_data`) avec une seule tâche qui produira (émettra) l'actif `s3://my-bucket/raw-weather.csv`.

En général, `@asset` est une simplification pour un DAG avec une seule tâche ayant la sortie spécifiée ; il crée automatiquement un objet **Asset**, un **DAG**, et une seule tâche avec cette sortie

Le paramètre **uri** identifie les données, et `schedule="@daily"` signifie que cet actif sera mis à jour quotidiennement.

Dépendances Pythoniques (sans >> / <<)

Dans le **Task SDK**, les dépendances entre tâches s'expriment par des appels de fonction et des valeurs de retour, et non par des opérateurs de décalage binaire (`>>`, `<<`).

Lorsque vous appelez une tâche depuis une autre (ou que vous passez la sortie d'une tâche en argument à une autre), Airflow crée automatiquement la dépendance

Exemple :

```
data1 = taskA()        # taskA est un @task ; cet appel renvoie un XComArg
data2 = taskB(data1)    # taskB dépend de taskA ; la sortie de taskA est pas
sée en entrée
taskC(data2)           # taskC dépend de taskB
```

En interne, chaque appel à une fonction décorée par `@task` renvoie un objet **XComArg**, et le fait de le passer en paramètre à une autre tâche la rend automatiquement dépendante (downstream).

La documentation du SDK précise : « *Les tâches communiquent via XComArg* » et il n'est plus nécessaire d'écrire `t1 >> t2` manuellement

La **task mapping dynamique** est tout aussi élégante :

```
results = multiply.expand(x=[1, 2, 3])
```

créera automatiquement trois tâches parallèles

Ce flux de données Pythonique élimine le code répétitif et rend les DAGs plus lisibles. Vous pouvez toujours utiliser des fonctions utilitaires comme `chain()` ou `chain_linear()` si nécessaire, mais dans la plupart des cas, il suffit de retourner des valeurs et d'appeler les tâches directement.

Sorties des tâches et XComs

Chaque fonction décorée avec `@task` peut retourner une valeur (ou un dictionnaire de valeurs), qui est ensuite envoyée dans le système **XCom** d'Airflow.

- Par défaut, une seule valeur de retour est stockée sous la clé `"return_value"`.
- Si vous définissez `@task(multiple_outputs=True)`, vous pouvez retourner un dictionnaire, et Airflow créera automatiquement une entrée XCom distincte pour chaque clé[6].

Exemple :

Dans l'exemple précédent, **fetch_weather** retourne un dictionnaire contenant `"temp"` et `"humidity"`, accessibles via `weather_data["temp"]` et `weather_data["humidity"]`.

Cela permet d'utiliser facilement les sorties des tâches :

```
@task(multiple_outputs=True)
def split_name(full: str) -> dict:
    parts = full.split()
    return {"first": parts[0], "last": parts[-1]}

name = split_name("Jane Doe")
print_name = print(name["last"])  # affiche "Doe"
```

Vous pouvez également extraire des **XComs entre différents DAGs** si nécessaire.

Par exemple, en utilisant le contexte hérité (`ti.xcom_pull`) tel que montré dans la documentation d'Astronomer, il est possible pour un DAG producteur de partager ses données avec un autre DAG consommateur en récupérant les XComs de l'exécution du DAG producteur

---

Intégration avec des systèmes externes (Spark, HDFS, Kafka, REST)

Airflow s'intègre avec la plupart des systèmes de données via des **packages providers**.

La bonne pratique consiste à utiliser ces **opérateurs/hooks** plutôt que de réinventer la roue[8].

Apache Spark

Utilisez le **SparkSubmitOperator** pour soumettre des jobs.

Exemple :

```
from airflow_sdk import dag
from airflow_sdk.operators.spark import SparkSubmitOperator
import pendulum

@dag(
    dag_id="spark_pipeline",
    schedule="@daily",
    start_date=pendulum.datetime(2025, 1, 1),
    catchup=False,
)
def spark_pipeline():
    run_spark = SparkSubmitOperator(
        task_id="run_spark_job",
        application="/opt/spark/jobs/aggregate_data.py",
        conf={"spark.master": "yarn", "spark.executor.memory": "4g"},
    )

    run_spark
```

```
spark_pipeline()
```

HDFS

Utilisez les opérateurs du provider HDFS (par ex. **HDFSPutFileOperator**, **HdfsSensor**) ou ses hooks.

Exemple :

```
from airflow_sdk import dag
from airflow_sdk.operators.hdfs import HDFSPutFileOperator
import pendulum

@dag(
    dag_id="hdfs_example",
    schedule="@daily",
    start_date=pendulum.datetime(2025, 1, 1),
    catchup=False,
)
def hdfs_example():
    put = HDFSPutFileOperator(
        task_id="store_csv",
        local_file="/tmp/data.csv",
        hdfs_path="/data/output/data.csv",
    )

    put


hdfs_example()
```

Kafka

Utilisez le provider Kafka ou les bibliothèques Python standards.

Par exemple, une tâche peut **produire** ou **consommer** des messages Kafka :

```
@task
def publish_to_kafka(topic: str, message: str):
    """Envoyer un message vers un topic Kafka."""
    from confluent_kafka import Producer
    p = Producer({'bootstrap.servers': 'kafka-broker:9092'})
    p.produce(topic, value=message)
    p.flush()

@task
def consume_from_kafka(topic: str) -> str:
    """Lire un message depuis un topic Kafka."""
    from confluent_kafka import Consumer
    c = Consumer({'bootstrap.servers': 'kafka-broker:9092', 'group.id': 'airfl
ow'})
    c.subscribe([topic])
    msg = c.poll(timeout=5.0)
    return msg.value().decode('utf-8') if msg else ""
```

Ces tâches peuvent être utilisées dans un DAG ou déclenchées via un **scheduling basé sur des événements**.

(Remarque : Airflow n'est pas destiné aux flux à latence ultra-faible, mais il peut sonder ou réagir aux nouveaux messages si nécessaire.)

REST / HTTP

Vous pouvez appeler des API REST directement dans des tâches Python, ou utiliser **SimpleHttpOperator** / **HttpSensor**.

Dans l'exemple météo ci-dessus, nous avons simplement utilisé la bibliothèque Python `requests` dans une fonction décorée avec `@task`.

Alternativement :

```
from airflow_sdk import dag
from airflow_sdk.operators.http import SimpleHttpOperator
import pendulum

@dag(
    dag_id="http_example",
```

```
    schedule="@daily",
    start_date=pendulum.datetime(2025, 1, 1),
    catchup=False,
)
def http_example():
    call_api = SimpleHttpOperator(
        task_id="get_weather",
        http_conn_id="weather_api",  # connexion définie dans Airflow
        endpoint="v1/London/conditions",
    )

    call_api


http_example()
```

Bonne pratique

Dans tous les cas, suivez le principe de l'orchestration via providers :

- Déléguez le calcul lourd à des outils spécialisés (par ex. **Spark** pour le Big Data).
- Utilisez **Airflow uniquement pour soumettre ou orchestrer** ces jobs.

Astronomer recommande de déporter le traitement intensif de données vers des frameworks comme **Spark**, et d'utiliser Airflow uniquement pour « *orchestrer ces jobs* »

Pipelines Lambda (Batch + Streaming)

Une **architecture Lambda** combine le traitement **batch** et **streaming** :

- une **couche batch** pour les données historiques à grande échelle,
- une **couche vitesse** pour les mises à jour en temps réel,
- et une **couche de service** pour fusionner les résultats

Dans Airflow, cela se met généralement en œuvre sous forme de **DAGs séparés** :

- **DAG couche batch** : un DAG planifié (par exemple quotidien) qui exécute des jobs ETL lourds ou des analyses. Par exemple, un DAG qui lance un job Spark pour traiter toutes les données collectées jusqu'à présent.
- **DAG couche vitesse** : un DAG déclenché par l'arrivée de nouvelles données (ou via des planifications très fréquentes) pour traiter des données incrémentales ou en streaming. Cela peut être déclenché par un événement (nouveau message Kafka ou fichier) ou exécuté chaque minute/heure via un capteur, ou déclenché depuis l'extérieur (voir la planification basée sur les assets/événements ci-dessous).
- **Couche de service** : peut être un système externe ou un autre DAG Airflow qui fusionne les sorties des couches batch et vitesse (par exemple pour mettre à jour une base de données de service).

Exemple :

```
@dag(schedule="@daily", start_date=pendulum.datetime(2025,1,1), catchu
p=False)
def daily_aggregation():
    @task
    def aggregate_all():
        # ex. lancer un job Spark sur les données historiques
        print("Exécution de l'agrégation batch quotidienne...")
    aggregate_all()

daily_aggregation = daily_aggregation()



@dag(schedule=None, start_date=pendulum.datetime(2025,1,1), catchup=F
alse)
def realtime_updates():
    @task
    def handle_event(event: str):
        print(f"Traitement d'un événement temps réel : {event}")
    # Ce DAG serait déclenché manuellement ou via un capteur d'événement
    handle_event("incoming-stream-data")

realtime_updates = realtime_updates()
```

En séparant **batch** et **stream** dans des DAGs différents, vous pouvez les **scaler et planifier indépendamment**.

Airflow 2.7+ permet également une **planification événementielle** via les **Assets** : par exemple, vous pourriez avoir un DAG couche vitesse utilisant `@asset` avec `uri="kafka://mytopic"` ou déclenché sur un événement dataset.

Astronomer note que vous pouvez planifier un DAG en fonction de messages dans une file d'attente (Kafka, Pub/Sub, etc.) comme planification événementielle

Par exemple, une tâche peut écouter de nouveaux messages Kafka et générer un événement Asset, qui déclenche ensuite les pipelines en aval.

Assets, URIs de données et Lineage

Les **Assets** dans Airflow sont des entités de données logiques (fichiers, tables, flux) identifiées par des **URIs**[12].

Exemples : `Asset("s3://bucket/data.csv")` ou `Asset("kafka://weather_topic")`.

- Les tâches peuvent produire un asset en le spécifiant dans `outlets=` ou en utilisant `@asset` comme montré précédemment.
- Lorsqu'une tâche se termine, Airflow émet un **événement asset** avec l'URI et le timestamp.
- Ces événements permettent la **planification basée sur les données** et le **suivi de lineage**.

> Comme indiqué dans la documentation : « Les DAGs qui accèdent aux mêmes données peuvent avoir des relations explicites et visibles, et les DAGs peuvent être planifiés en fonction des mises à jour de ces assets »

Exemple avec l'approche Astronomer :

```
@asset(schedule="@daily")
def raw_data():
    # produire raw_data.csv dans S3
    ...

@asset(schedule=raw_data)
def processed_data(context):
```

```
    # Ce DAG s'exécute après la mise à jour de raw_data. Il peut récupérer vi
a XCom la tâche raw_data.
    raw = context["ti"].xcom_pull(dag_id="raw_data", task_ids="raw_data")
    ...
```

- Le **planning du second DAG dépend du premier asset** (`schedule=raw_data`), il s'exécute donc dès que **raw_data** est produit
- Dans l'UI, cela apparaît comme un **graphique de dépendances d'assets**.

Lineage dans Airflow

L'outil de **lineage d'Airflow (AIP-60)** s'accroche à ces événements asset.

- En interne, les tâches/hooks peuvent appeler l'API **Lineage** pour déclarer les assets en entrée/sortie, et Airflow construit les **métadonnées de lineage AIP-60**
- En pratique, simplement utiliser `outlets=` ou `@asset` fait générer des **événements de lineage** compatibles avec **OpenLineage/DataHub**.
- Des métadonnées supplémentaires peuvent être attachées en renvoyant un objet **Metadata** depuis une tâche (par ex. nombre de lignes), enrichissant davantage le suivi de lineage.

Bonnes pratiques pour des DAGs prêts pour la production

Pour rendre vos DAGs **réutilisables, clairs et de qualité production**, considérez les pratiques suivantes :

- **Tâches petites et idempotentes** :

  Découpez le travail en étapes atomiques (ex. extraction, transformation, chargement) afin que chaque tâche puisse être **réessayée indépendamment**. L'idempotence garantit que réexécuter une tâche avec les mêmes entrées **ne duplique pas et ne corrompt pas les données**.

- **Utiliser les retries et timeout** :

  Définissez des **retries**, **retry_delay** et **timeouts** pertinents sur les tâches ou dans `default_args` pour gérer les échecs transitoires.

- **Exploiter les opérateurs des providers** :

Comme vu précédemment, utilisez les **opérateurs/hooks intégrés** pour Spark, HDFS, Kafka, REST, etc., plutôt que d'écrire toute la logique en Python brut.

- **Déléguer le traitement lourd** :

  Pour de grands ensembles de données, ne tentez pas de tout faire dans Airflow pur. Utilisez **Spark** ou des services cloud natifs pour les jobs lourds, et orchestrez-les depuis Airflow[9].

- **Paramétrage et configuration** :

  Utilisez des arguments de DAG/fonction et les **Params/Variables d'Airflow** pour les valeurs spécifiques à l'environnement. Évitez de coder en dur les URIs ou identifiants dans le code (préférez **Connections** et **Variables**).

- **Nommage clair et tags** :

  Donnez aux DAGs et aux tâches des `dag_id` / `task_id` descriptifs et ajoutez des tags pour faciliter la recherche et le filtrage dans l'UI.

- **TaskGroup pour organiser la structure** :

  Si vous avez de nombreuses tâches liées, regroupez-les avec `@task_group` ou `airflow.sdk.task_group()` pour garder le graphe organisé.

- **Éviter la logique au niveau global** :

  Ne mettez pas de calculs lourds ou d'I/O lors de l'import du module. Définissez uniquement les DAGs/tâches au niveau supérieur ; exécutez la logique **à l'intérieur des tâches**.

  (Voir les best practices Airflow pour plus de conseils.)

- **Structure de projet cohérente** :

  Suivez une structure logique du code (ex. dossiers séparés `dags/`, `plugins/`, etc.) Cela facilite la maintenance et la collaboration.

- **Surveillance et logging** :

  Incluez toujours du **logging** (`print()` ou `logging`) dans les tâches, et configurez des **SLAs/alertes** si nécessaire.

- **Vérifier le flux de dépendances** :

  Inspectez visuellement le graphe du DAG pour vous assurer qu'il correspond à votre architecture prévue (batch vs stream, dépendances d'assets, etc.).

- **Utiliser des tâches setup/teardown si nécessaire** :

  Pour la gestion de ressources (ex. créer un cluster puis le supprimer), utilisez les nouveaux décorateurs `@setup` / `@teardown` (fonctionnalité Airflow 2.7) pour gérer automatiquement ces dépendances[18].

- **Tester vos DAGs** :

  Écrivez des **tests unitaires** pour la logique des tâches et utilisez `airflow tasks test` ou des pipelines CI pour valider les définitions de DAG avant le déploiement.
