



Apache
Airflow



Programme

Module 01 INTRODUCTION

- Apache Airflow
- L'histoire d'Apache Airflow
- Les Fondateurs d'Apache Airflow
- Les raisons derrière la création d'Apache Airflow
- Chronologie de la création et de l'évolution d'apache airflow
- Quand utiliser airflow
- Outils similaires à apache airflow

Module 02 ARCHITECTURE ET COMPOSANTS

- Airflow et ses composants
- Architecture airflow
- L'interface utilisateur
- La ligne de commande

Module 03 ANATOMIE DE WORKFLOWS

- Fonctionnement et la planification des workflows
- Les best practices
- Limitations de airflow

Module 04 DIRECTED ACYCLIC GRAPHS (DAG)

- Déclaration d'un DAG
- Exécution des DAG
- Best practices





Programme

Module 05

TASKS & OPERATORS

- Instances de tâches
- Les dépendances entre tache / opérateurs
- Regrouper les tâches connexes en utilisant des groupes de tâches
- Best practices

Module 06

EXÉCUTEURS

- Differents types d'exécuteurs
- Exécuteurs séquentiels avec sqlite
- Exécuteurs locaux avec postgresql
- Configurer un dag avec un exécuteur local et postgresql
- Les exécuteurs avec postgresql et rabbitmq
- Les exécuteurs multi-noeuds
- Exécuteurs : Comparaison

Module 07

GESTION DES WORKFLOWS

- Les operateurs : sensoroperator, actionoperator et transferoperator
- Data-aware scheduling
- Task Factory
- Dynamic task mapping
- Defferable operators
- Best practices

Programme

- Module 08 AIRFLOW OPTIMISATION**
 - Bottleneck & deadlocks
 - Deadlock : causes
 - Prévention des deadlock
 - Bottleneck : causes et traitements
 - Jinja templating
- Module 09 DAG BRANCHING**
 - Branching dans le dag
 - Ajout de tâches supplémentaires de récupération/nettoyage
- Module 10 OPTIMISER LE TRAITEMENT DES DONNEES**
 - Polling custom conditions
 - Efficient data handling
- Module 11 GESTION DE CONCURRENCE AVEC LES POOLS DE RESSOURCES**
 - Pools de ressources
- Module 12 MONITORING W/NOTIFICATIONS**
 - Alert monitoring
- Module 13 MONITORING LONG RUNNING TASK W/ SLA**
 - Sla monitoring





Programme

Module 14

POLLING W/ SENSORS

- Polling custom conditions
- Un FileSensor attend qu'un chemin de fichier existe

Module 15

BACKFILLING W/ TRIGGER OPERATOR

Module 16

POLLING W/ EXTERNAL SENSORS

- POLLING THE STATE OF OTHER DAG

Module 17

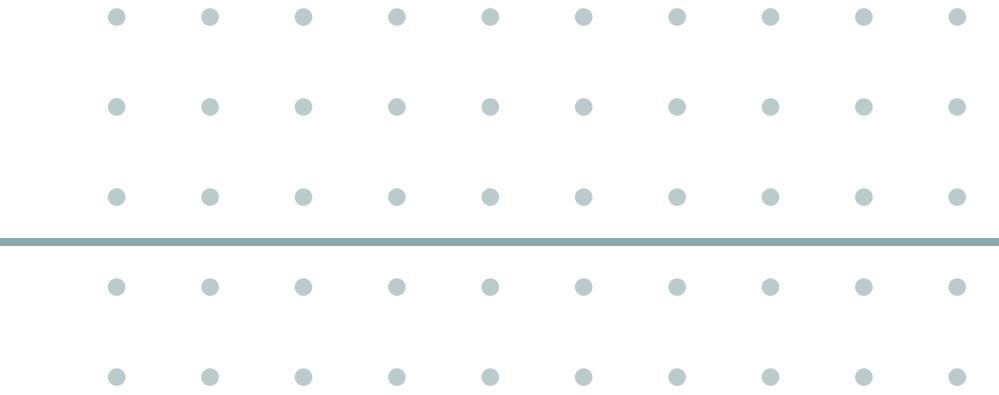
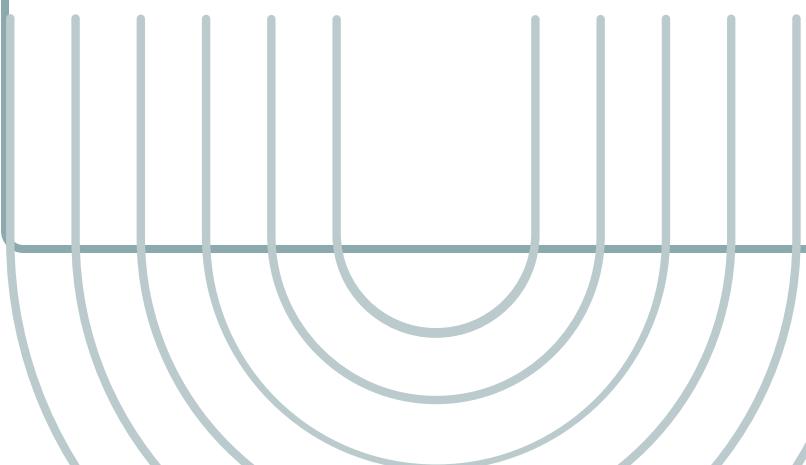
STARTING WORKFLOWS WITH REST/CLI

- POLLING THE STATE OF OTHER DAG



01.

INTRODUCTION





APACHE AIRFLOW

- Apache Airflow est une plateforme de planification de flux de travail open-source, très utilisée dans le domaine de l'ingénierie des données.
- La plateforme Apache Airflow permet de créer, de planifier et de surveiller des workflows (flux de travail) par le biais de la programmation informatique. Il s'agit d'une solution totalement open source, très utile pour l'architecture et l'orchestration de pipelines de données complexes et le lancer de tâches.
- Les workflows sont architecturés et exprimés sous forme de Directed Acyclic Graphs (DAGs) :Graphe orienté acyclique en français , dont chaque noeud représente une tâche spécifique.





L'HISTOIRE D' APACHE AIRFLOW

- L'histoire d' **Apache Airflow** commence en 2015, dans les bureaux de **AirBnB**. A cette époque, la plateforme de locations de vacances fondée en 2008 connaît une croissance fulgurante et croule sous un volume de données toujours plus massif.
- L'entreprise californienne recrute des **Data Scientists**, des **Data Analysts** et des **Data Engineers** à tour de bras, et ces derniers doivent automatiser de nombreux processus en écrivant des batch jobs planifiés.





LES FONDATEURS D'APACHE AIRFLOW

- Afin de les épauler, l'ingénieur des données **Maxime Beauchemin** crée un outil open-source avec son équipe «  » intitulé **Airflow**.
- Cet outil de planification vise à permettre aux équipes de créer, de surveiller les pipelines de données en lot et d'itérer.
- En quelques années, Airflow s'est imposé comme un standard dans le domaine du Data Engineering.



Maxime Beauchemin
**Fondateur et PDG de Preset,
créateur d'Apache Superset
et d'Apache Airflow**



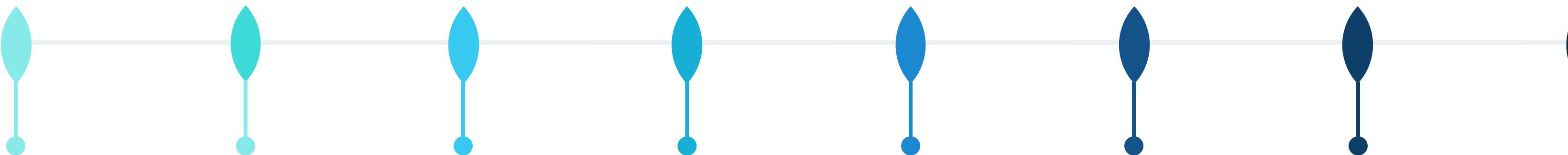
LES RAISONS DERRIÈRE LA CRÉATION D'APACHE AIRFLOW

- Airflow a été créé pour aider à automatiser et à orchestrer des workflows complexes de traitement de données dans des environnements de Big Data.
- En raison des limites rencontrées avec les outils existants pour gérer leurs flux de travail de traitement de données à grande échelle.
- Les fondateurs ont donc créé Airflow pour répondre à leur propre besoin, puis ont décidé de le rendre open-source pour aider d'autres organisations à résoudre des problèmes similaires.
- Aujourd'hui, Airflow est largement utilisé dans le monde entier par de nombreuses entreprises pour gérer et orchestrer des workflows de traitement de données.





CHRONOLOGIE DE LA CRÉATION ET DE L'ÉVOLUTION D'APACHE AIRFLOW



2014

L'équipe de **data engineering** chez **Airbnb** commence à travailler sur **Airflow** pour répondre à leurs **besoins internes** de gestion de flux de travail de traitement de données.

2015

Airflow est **open-source** et publié sous licence **Apache 2.0**.

2016

Airflow est présenté lors de la conférence **PyCon** et **gagne en popularité** dans la communauté Python.

2017

La fondation Apache Software Foundation accepte **Airflow en tant que **projet incubateur**.**

2018

La version 1.10.0 d'Airflow est publiée avec de nouvelles fonctionnalités et améliorations, notamment **un nouveau moteur d'exécution**.

2019

Airflow devient un projet **Apache** de premier niveau (top-level project) et publie **la version 1.10.9**.

2020

La version 2.0 d'Airflow est publiée avec une refonte majeure de **l'interface utilisateur** et plusieurs nouvelles fonctionnalités et améliorations.

2021

Airflow continue de se développer et d'être largement utilisé dans le monde entier, avec une communauté active de développeurs et d'utilisateurs qui contribuent régulièrement au projet.



QUAND UTILISER AIRFLOW

Raisons pour choisir Airflow

- La capacité de mettre en œuvre des pipelines à l'aide de code Python vous permet de créer des pipelines arbitrairement complexes en utilisant tout ce que vous pouvez imaginer en Python.
- La fondation Python d'Airflow facilite l'extension et l'ajout d'intégrations avec de nombreux systèmes différents.
- Riche collection d'extensions développées par la communauté qui permettent à Airflow de s'intégrer à de nombreux types de bases de données, de services cloud, et ainsi de suite.
- Les sémantiques de planification permettent d'exécuter les pipelines à intervalles réguliers et de construire des pipelines efficaces qui utilisent un traitement incrémental pour éviter le recalcul coûteux des résultats existants.



QUAND UTILISER AIRFLOW

Raisons pour choisir Airflow

- Des fonctionnalités telles que le backfilling permettent de (re)traiter facilement des données historiques,
- de recomputer tous les ensembles de données dérivés après avoir apporté des modifications à votre code.
- L'interface Web riche d'Airflow fournit une vue facile pour surveiller les résultats de vos exécutions de pipeline et déboguer les éventuelles défaillances qui ont pu se produire.



QUAND UTILISER AIRFLOW

Raisons pour ne pas choisir Airflow

- Gestion des pipelines de streaming, car Airflow est principalement conçu pour exécuter des tâches récurrentes ou orientées lot (batch), plutôt que des charges de travail en streaming.
- Mise en place de pipelines hautement dynamiques, dans lesquels les tâches sont ajoutées/supprimées entre chaque exécution du pipeline.
- Bien qu'Airflow puisse implémenter ce type de comportement dynamique, l'interface web ne montrera que les tâches qui sont toujours définies dans la version la plus récente du DAG.
- Ainsi, Airflow favorise les pipelines qui ne changent pas de structure à chaque exécution.



QUAND UTILISER AIRFLOW

Raisons pour ne pas choisir Airflow

- Équipes avec peu ou pas d'expérience en programmation (Python), car la mise en œuvre de DAG en Python peut être intimidante avec peu d'expérience en Python.
- Dans de telles équipes, l'utilisation d'un gestionnaire de workflow avec une interface graphique (comme Azure Data Factory) ou une définition de workflow statique peut être plus judicieuse.
- De même, le code Python dans les DAG peut rapidement devenir complexe pour des cas d'utilisation plus importants.
- La mise en place et la maintenance des DAG Airflow nécessitent une rigueur d'ingénierie appropriée pour maintenir les choses maintenables à long terme.



OUTILS SIMILAIRES À APACHE AIRFLOW

- **Luigi** est un package Python qui aide à construire des pipelines complexes de tâches batch. Il gère la résolution de dépendances, la gestion des workflows, la visualisation, la gestion des échecs, l'intégration en ligne de commande et bien plus encore.
- **Oozie** est un système open-source d'orchestration de flux de travail développé pour le framework Hadoop. Il permet de planifier, de coordonner et de gérer des tâches de traitement de données complexes dans un environnement distribué.
- **Azkaban** est un gestionnaire de flux de travail distribué, implémenté chez LinkedIn pour résoudre le problème des dépendances de tâches Hadoop. Il offre également une interface Web pour la définition et la visualisation de flux de travail.





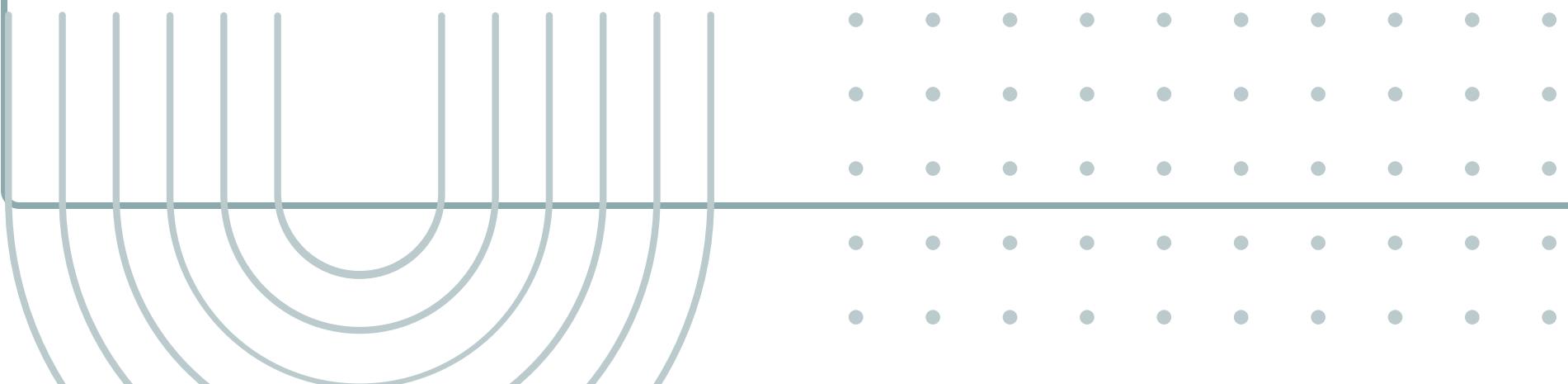
OUTILS SIMILAIRES À APACHE AIRFLOW

Voici quelques avantages et inconvénients distinctifs d'outils d'orchestration de flux de travail de traitement de données en comparaison avec Apache Airflow :

- **Luigi:**
 - Avantages: Facile à apprendre et à utiliser, idéal pour des tâches simples et linéaires, bon support de Python
 - Inconvénients: Limité en termes de fonctionnalités avancées, manque d'évolutivité
- **Oozie:**
 - Avantages: Conçu pour Hadoop et écosystème Hadoop, fonctionnalités avancées pour les tâches de données sur Hadoop
 - Inconvénients: Configuration complexe, pas très adapté aux tâches en dehors de Hadoop, manque de support de Python
- **Azkaban:**
 - Avantages: Interface utilisateur conviviale, bonne évolutivité, grande communauté
 - Inconvénients: Configuration complexe, manque de fonctionnalités en comparaison avec Airflow, faible support de Python

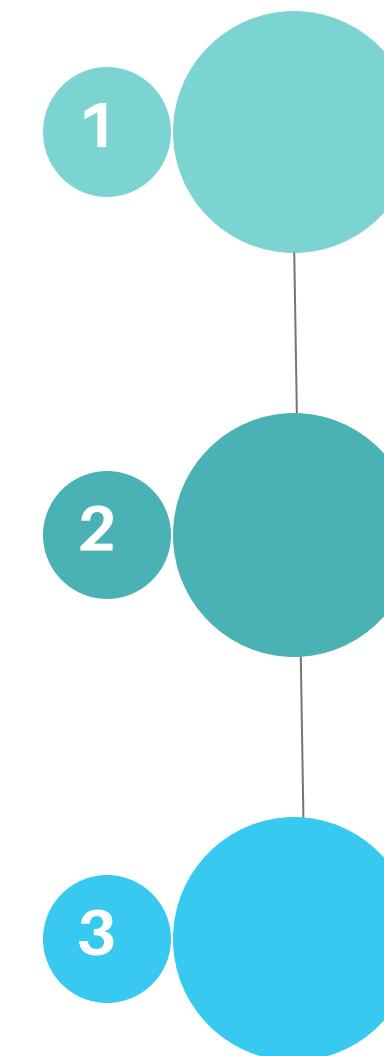
02.

ARCHITECTURE et COMPOSANTS





AIRFLOW ET SES COMPOSANTS



Serveur web

- Le serveur web est le composant qui est responsable de la gestion de toutes les API de l'interface utilisateur et de REST.

Planificateur (Scheduler)

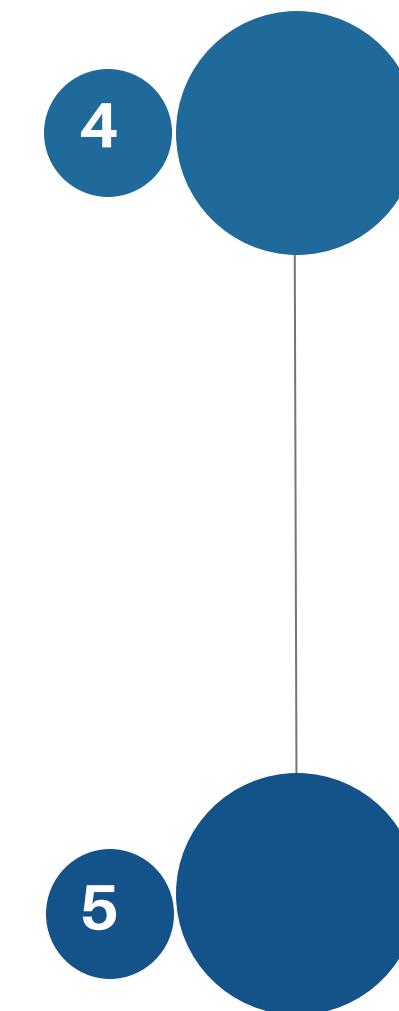
- Le planificateur passe en revue les DAG toutes les n secondes et programme la tâche à exécuter. Le planificateur possède également un composant interne appelé Executor.

Exéuteurs

- responsable de l'exécution des tâches définies dans un graphe acyclique orienté (DAG). Orchestre l'exécution des tâches en fonction des dépendances sur les ressources allouées aux workers.



AIRFLOW ET SES COMPOSANTS



Worker

- Les workers sont les process qui exécutent de façon effective les tâches qui leurs sont confiéee par l'exécuteur.

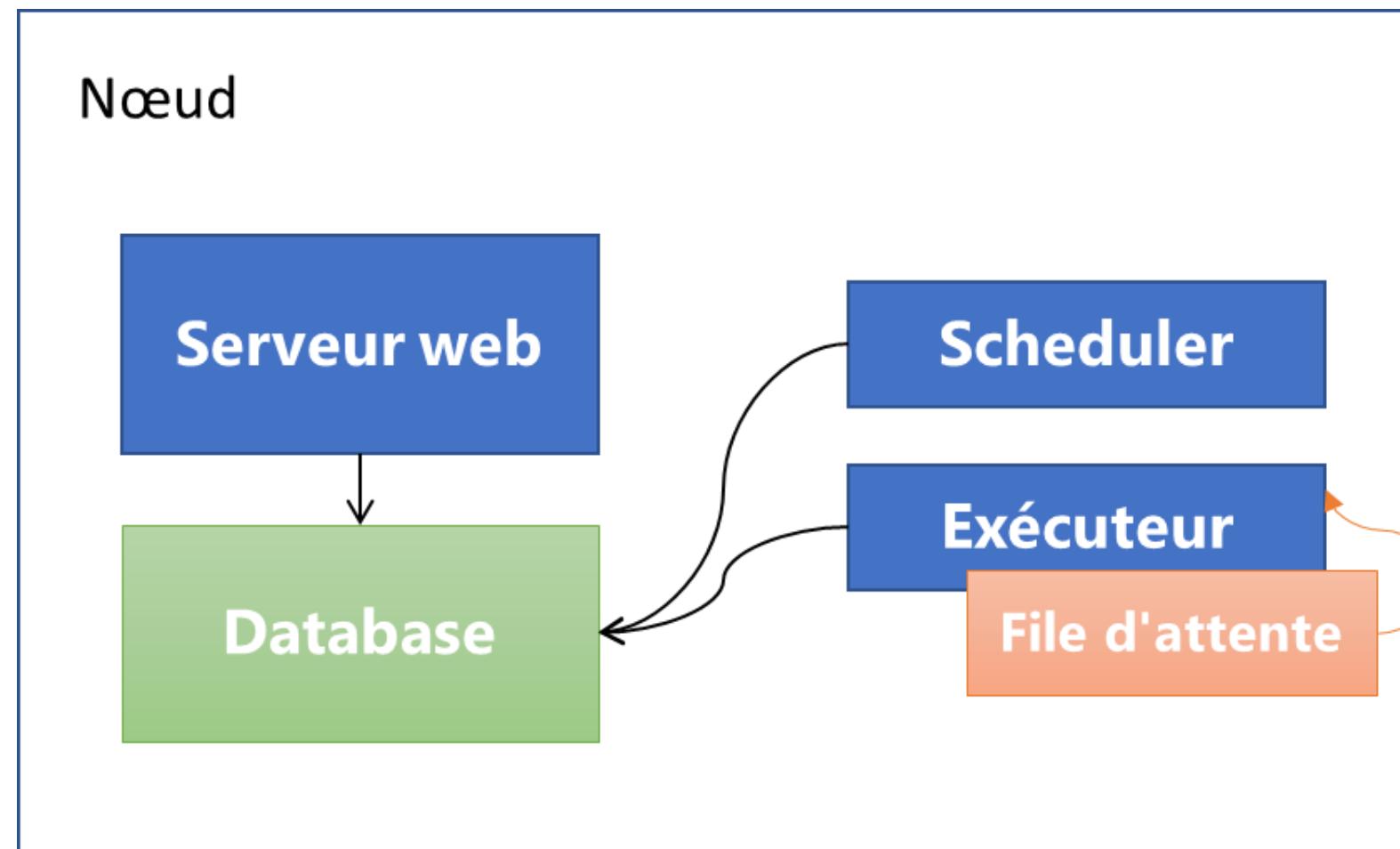
Database (Meta)

- Une base de données dans laquelle toutes les métadonnées du DAG et des tâches sont stockées. Il s'agit généralement d'une base de données Postgres, mais MySQL, MsSQL et SQLite sont également pris en charge.



ARCHITECTURE AIRFLOW

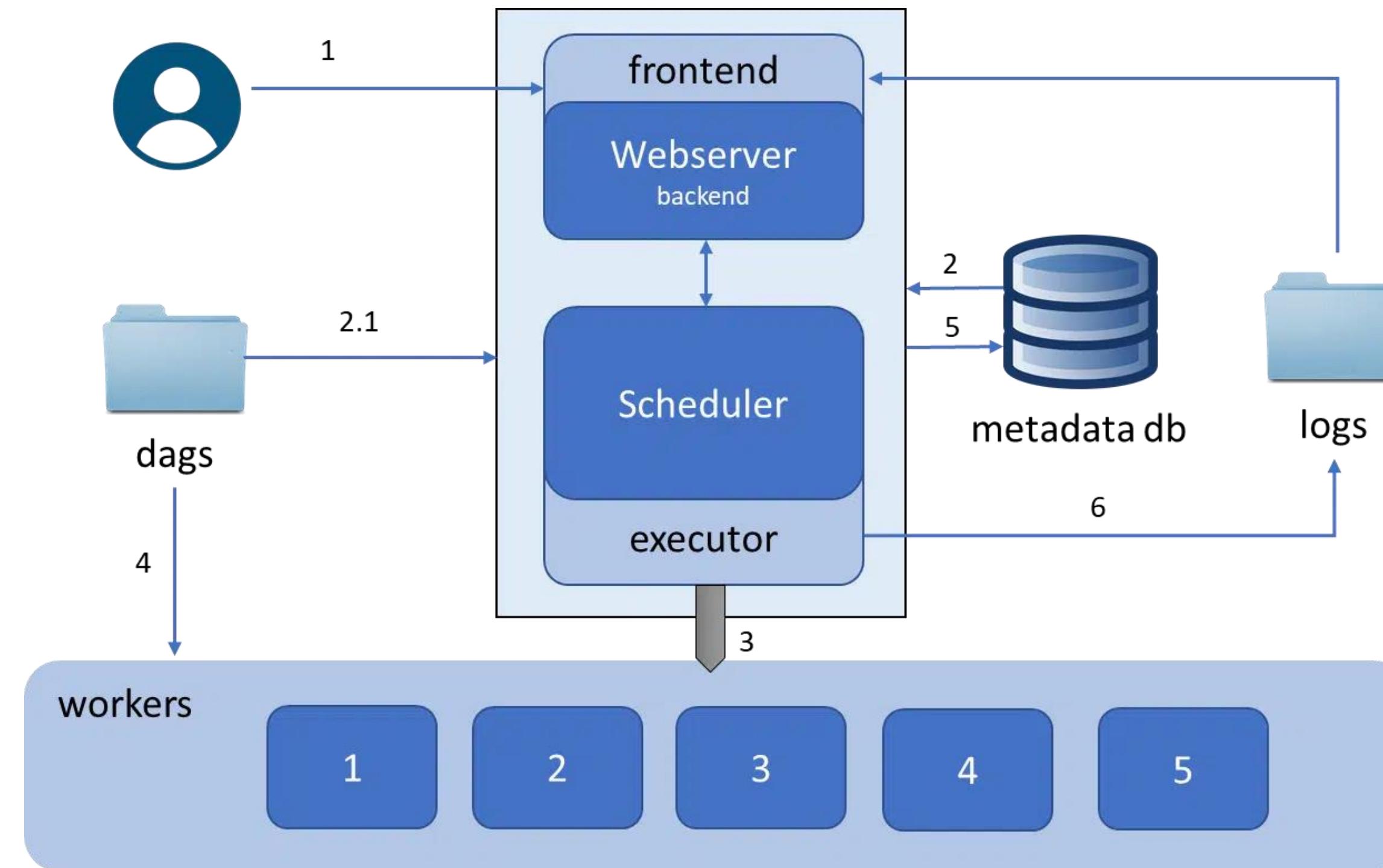
ARCHITECTURE À NŒUD UNIQUE



- L'architecture de nœud unique ("single node") d'Airflow est une configuration dans laquelle tous les composants d'Airflow, tels que le webserver, le scheduler, la base de données et l'executeur, sont tous exécutés sur une seule machine.
- Le serveur web communique avec la base de données et, bien évidemment, le planificateur ainsi que l'exécuteur communiquent également avec la métabase de données.
- La métabase de données permet d'échanger des données entre les différents composants d'Airflow, et qu'il existe également une file d'attente qui vous permet d'exécuter les tâches dans le bon ordre.
- Il y a toujours une file d'attente, quel que soit l'exécuteur utilisé.



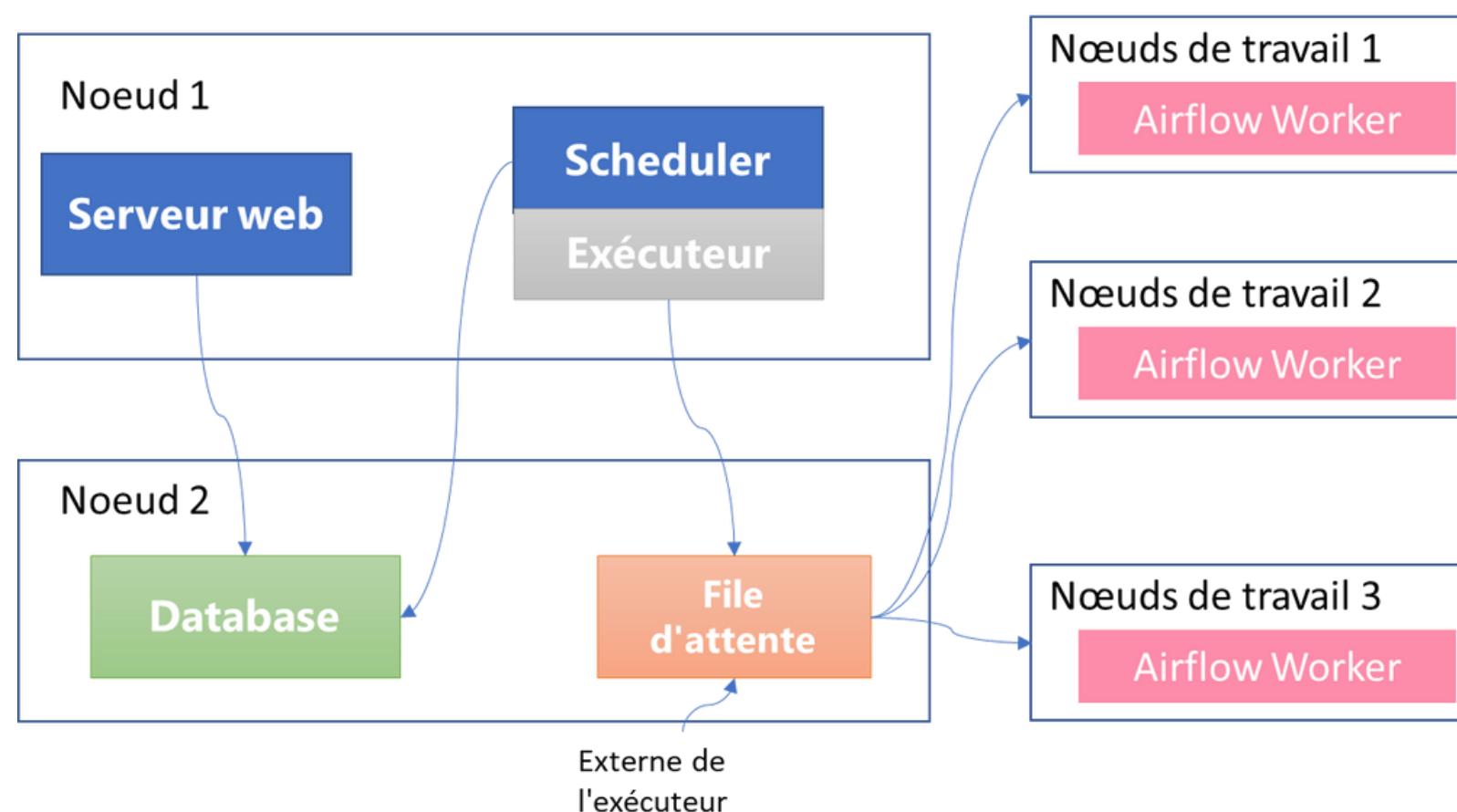
ARCHITECTURE AIRFLOW





ARCHITECTURE AIRFLOW

ARCHITECTURE MULTI-NŒUDS

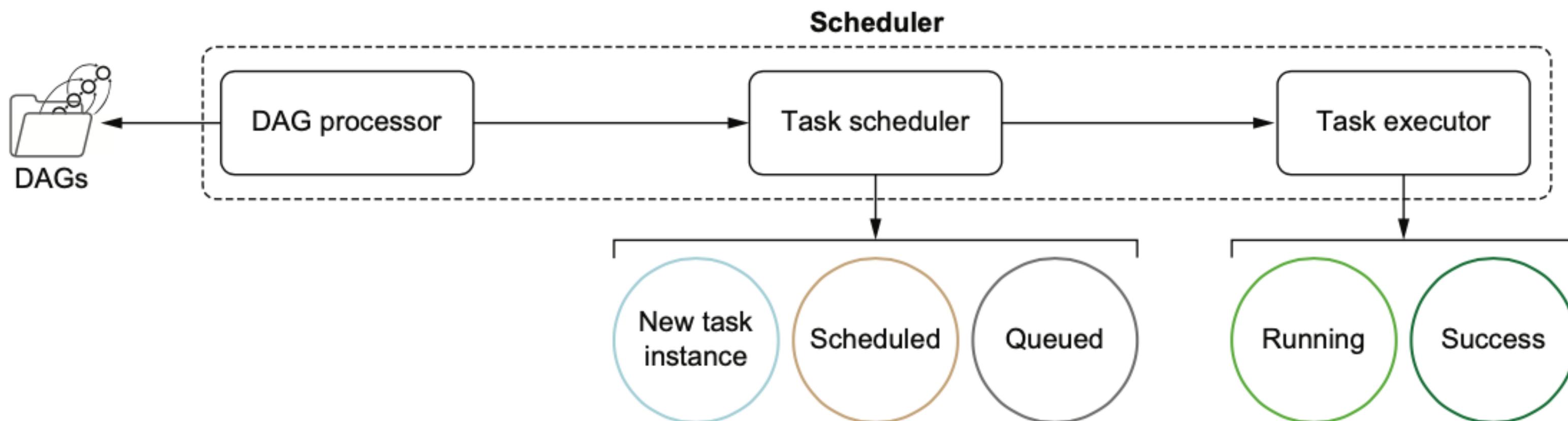


- Il est déconseillé d'avoir une architecture à un seul nœud pour garantir la fiabilité d'Airflow en production.
- Une architecture Multi-Noeud est préférable pour éviter les points de défaillance uniques.
- Cette architecture consiste en un premier nœud avec le serveur Web, le planificateur et l'exécuteur, et un deuxième nœud avec la métabase de données et la file d'attente.
- Il y a également trois nœuds de travail dédiés à l'exécution des tâches, chacun équipé d'un Airflow Worker.
- Les différents nœuds communiquent entre eux pour permettre la gestion et l'exécution des tâches dans l'ordre approprié.



ARCHITECTURE AIRFLOW

ZOOM SUR LE SCHEDULER





L'INTERFACE UTILISATEUR

- Apache Airflow comprend une interface utilisateur web qui permet de gérer les workflows (DAG), de gérer l'environnement Airflow et d'effectuer des actions d'administration.

Airflow DAGs Security Browse Admin Docs 21:11 UTC RH

DAGs

All 26	Active 10	Paused 16	Filter DAGs by tag	Search DAGs			
<i>DAG</i>	Owner	Runs	Schedule	Last Run	Recent Tasks	Actions	Links
example_bash_operator example example2	airflow	2	0 0 * * *	2020-10-26, 21:08:11	6	[green circle] [blue circle] [red circle]	[...]
example_branch_dop_operator_v3 example	airflow	0	* /1 * * *		0	[blue circle] [red circle]	[...]
example_branch_operator example example2	airflow	1	@daily	2020-10-23, 14:09:17	11	[green circle] [blue circle] [red circle]	[...]
example_complex example example2 example3	airflow	1	None	2020-10-26, 21:08:04	37	[green circle] [blue circle] [red circle]	[...]
example_external_task_marker_child	airflow	1	None	2020-10-26, 21:07:33	2	[green circle] [blue circle] [red circle]	[...]
example_external_task_marker_parent	airflow	1	None	2020-10-26, 21:08:34	1	[green circle] [blue circle] [red circle]	[...]
example_kubernetes_executor example example2	airflow	0	None		0	[blue circle] [red circle]	[...]
example_kubernetes_executor_config example3	airflow	1	None	2020-10-26, 21:07:40	5	[green circle] [blue circle] [red circle]	[...]
example_nested_branch_dag example	airflow	1	@daily	2020-10-26, 21:07:37	9	[green circle] [blue circle] [red circle]	[...]
example_passing_params_via_test_command example	airflow	0	* /1 * * *		0	[blue circle] [red circle]	[...]



LA LIGNE DE COMMANDE

- Il est possible d'exécuter des commandes CLI en utilisant les services airflow-* prédéfinis.

Par exemple, sous docker, pour exécuter la commande "airflow info", il suffit d'utiliser la commande la suivante :

```
docker compose run airflow-worker airflow info
```



LA LIGNE DE COMMANDE

- Il est possible de simplifier le travail en téléchargeant des scripts de wrapper optionnels qui permettent d'exécuter les commandes avec une commande plus simple.

```
curl -Lf0 'https://airflow.apache.org/docs/apache-airflow/2.5.1/airflow.sh'  
chmod +x airflow.sh
```

- Il est maintenant possible d'exécuter des commandes plus facilement.



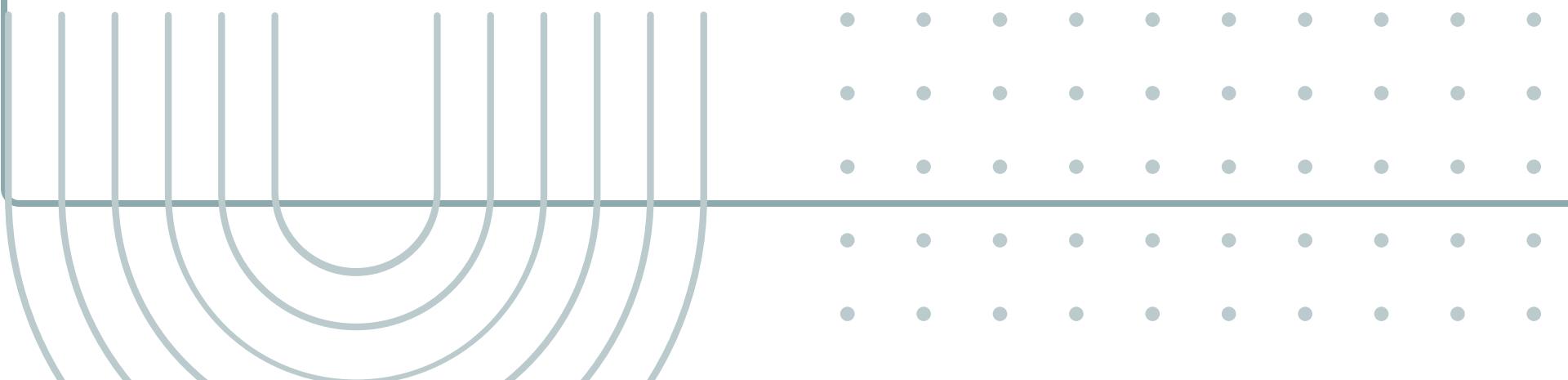
LA LIGNE DE COMMANDE

Vous pouvez également utiliser bash comme paramètre pour entrer un shell bash interactif dans le conteneur.

```
./airflow.sh bash
```

03.

ANATOMIE DE WORKFLOWS





FONCTIONNEMENT ET LA PLANIFICATION DES WORKFLOWS

FONCTIONNEMENT

Le fonctionnement d'Airflow peut être divisé en trois grandes étapes :

La définition des workflows

Airflow utilise un langage de programmation appelé Python pour définir les workflows sous forme de DAGs. Les développeurs peuvent écrire des fonctions Python qui représentent des tâches et des opérations à exécuter. Ces fonctions sont ensuite assemblées dans un DAG pour décrire la logique de flux de travail.

La planification des tâches

Airflow utilise un planificateur pour déterminer quand chaque tâche doit être exécutée en fonction des dépendances définies dans le DAG. Le planificateur détermine également quelles tâches peuvent s'exécuter en parallèle et quelles tâches doivent attendre l'achèvement d'autres tâches.

L'exécution et la surveillance des tâches

Airflow utilise des exéuteurs pour lancer les tâches en fonction de la planification et surveiller leur progression. Les résultats des tâches sont stockés dans une base de données, ce qui permet aux utilisateurs de surveiller l'état d'exécution.





FONCTIONNEMENT ET LA PLANIFICATION DES WORKFLOWS

PLANIFICATION DES WORKFLOWS

Un flux de travail (ou workflow) est :

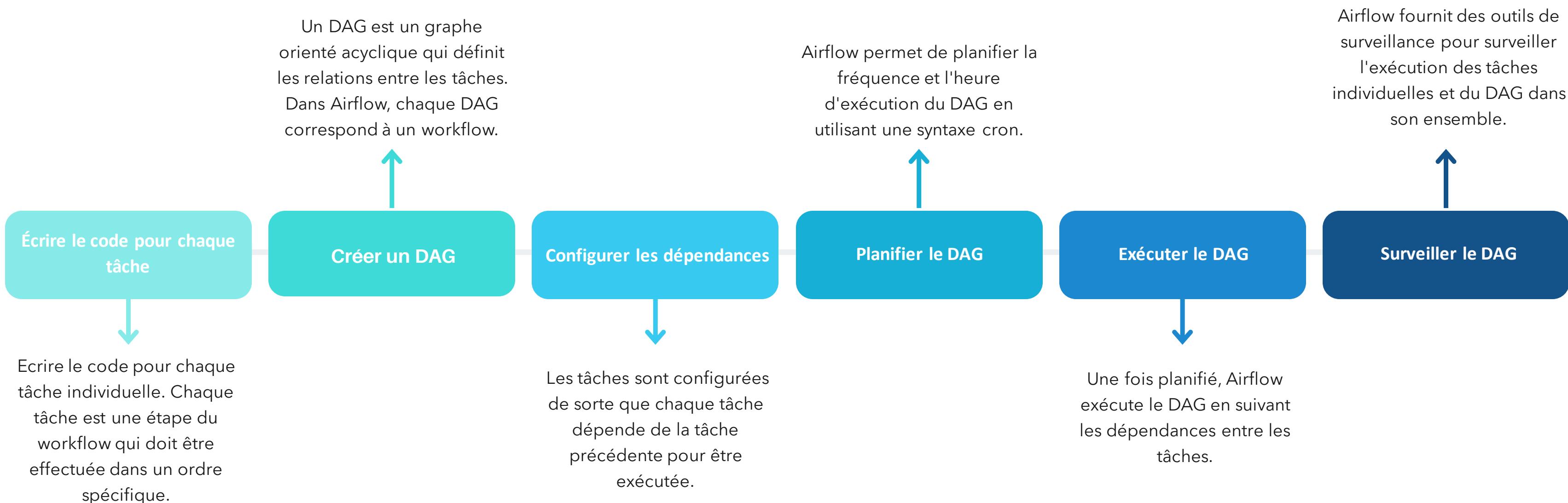
- Un ensemble d'étapes pour accomplir une tâche d'ingénierie de données donnée Tel que :
 - Téléchargement de fichiers
 - Copie de données
 - Filtrage d'informations
 - Ecriture dans une base de données, etc.
- De niveaux de complexité variables
- Un terme avec diverses significations selon le contexte.



FONCTIONNEMENT ET PLANIFICATION DE WORKFLOWS

PLANIFICATION DES WORKFLOWS

Voici les étapes pour créer un workflow dans Airflow :





LES BEST PRACTICES

LES BEST PRACTICES

- **Utiliser des variables et des connexions :**

Airflow permet de stocker des variables et des connexions dans la métabase de données. Il est recommandé d'utiliser ces fonctionnalités pour stocker les informations de connexion et les variables qui sont nécessaires pour exécuter les tâches du workflow.

- **Tester les workflows :**

Il est important de tester les workflows avant de les déployer en production. Airflow propose des outils de test intégrés pour valider les tâches et les DAGs.



LES BEST PRACTICES

LES BEST PRACTICES

- **Surveiller les workflows :**

Airflow propose une interface de monitoring pour surveiller les workflows en temps réel.

Il est important de surveiller les workflows pour détecter les échecs et les anomalies.

- **Utiliser les opérateurs appropriés :**

Airflow propose une grande variété d'opérateurs pour exécuter différentes tâches.

Il est important de choisir l'opérateur approprié pour chaque tâche.

- **Utiliser des conteneurs :**

Il est recommandé d'utiliser des conteneurs pour exécuter les tâches dans des environnements isolés. Cela permet d'assurer la cohérence de l'environnement d'exécution et de simplifier le déploiement.





LIMITATIONS DE AIRFLOW

LES LIMITATIONS

- **Scalabilité :**

Airflow peut gérer un grand nombre de tâches et de workflows, mais il est important de prendre en compte les limitations de la machine sur laquelle il s'exécute.

- **Complexité :**

Les workflows complexes peuvent être difficiles à gérer dans Airflow. Il est important de diviser les workflows en tâches plus petites et plus simples pour faciliter la maintenance et le débogage.



LIMITATIONS DE AIRFLOW

LES LIMITATIONS

- **Performance :**

Airflow est un framework flexible, mais cela peut avoir un impact sur les performances.

Il est important de surveiller les performances et d'optimiser les workflows si nécessaire.

- **Sécurité :**

Airflow ne propose pas de mécanismes de sécurité avancés.

Il est important de mettre en place des mesures de sécurité pour protéger les workflows et les données.



ATELIER

INSTALLATION
CONFIGURATION



CONFIGURER APACHE AIRFLOW AVEC DOCKER-COMPOSE EN 5 MINUTES

AVANT DE COMMENCER

Cette procédure suppose une familiarité avec Docker et Docker Compose.

- **Docker** : Une plateforme de virtualisation de conteneurs qui permet d'exécuter des applications dans des environnements isolés, appelés conteneurs, afin de faciliter le déploiement et la gestion d'applications sur différents systèmes d'exploitation et infrastructures.
- **Docker-compose** : Est un outil de gestion de conteneurs qui permet de définir et de gérer plusieurs conteneurs Docker en même temps

RÉCUPÉRATION DOCKER-COMPOSE.YAML

Pour le déploiement d'Airflow sur Docker Compose, il est nécessaire de récupérer le fichier docker-compose.yaml.





CONFIGURER APACHE AIRFLOW AVEC DOCKER-COMPOSE EN 5 MINUTES

RÉCUPÉRATION DOCKER-COMPOSE.YAML

```
curl -Lf0 'https://airflow.apache.org/docs/apache-airflow/2.5.1/docker-compose.yaml'
```

Ce fichier contient plusieurs définitions de service:

- **airflow-init** : Le service d'initialisation.
- **airflow-scheduler** : Le planificateur surveille toutes les tâches et les DAG, puis déclenche les instances de tâches une fois leurs dépendances terminées.

```
airflow-scheduler:  
  <<: *airflow-common  
  command: scheduler  
  healthcheck:  
    test: ["CMD-SHELL", 'airflow jobs check --job-type SchedulerJob --hostname "${HOSTNAME}"']  
    interval: 10s  
    timeout: 10s  
    retries: 5  
  restart: always  
  depends_on:  
    <<: *airflow-common-depends-on  
    airflow-init:  
      condition: service_completed_successfully
```

- **airflow-webserver** : Le serveur Web est disponible sur <http://localhost:8080>.

```
airflow-webserver:  
  <<: *airflow-common  
  command: webserver  
  ports:  
    - 8080:8080  
  healthcheck:  
    test: ["CMD", "curl", "--fail", "http://localhost:8080/health"]  
    interval: 10s  
    timeout: 10s  
    retries: 5  
  restart: always  
  depends_on:  
    <<: *airflow-common-depends-on  
  airflow-init:  
    | condition: service_completed_successfully
```

- **airflow-worker** : Le travailleur qui exécute les tâches données par le planificateur.

```
airflow-worker:  
  <<: *airflow-common  
  command: celery worker  
  healthcheck:  
    test:  
      - "CMD-SHELL"  
      - 'celery --app airflow.executors.celery_executor.app inspect ping -d "celery@$${HOSTNAME}"'  
    interval: 10s  
    timeout: 10s  
    retries: 5  
  environment:  
    <<: *airflow-common-env  
    # Required to handle warm shutdown of the celery workers properly  
    # See https://airflow.apache.org/docs/docker-stack/entrypoint.html#signal-propagation  
    DUMB_INIT_SETSID: "0"  
  restart: always  
  depends_on:  
    <<: *airflow-common-depends-on  
  airflow-init:  
    | condition: service_completed_successfully
```

- **postgres** : La base de données.

```
postgres:  
  image: postgres:13  
  environment:  
    POSTGRES_USER: airflow  
    POSTGRES_PASSWORD: airflow  
    POSTGRES_DB: airflow  
  volumes:  
    - postgres-db-volume:/var/lib/postgresql/data  
  healthcheck:  
    test: ["CMD", "pg_isready", "-U", "airflow"]  
    interval: 5s  
    retries: 5  
  restart: always
```



- **redis - Les redis** : courtier (broker) qui transmet les messages du planificateur au travailleur.

```
redis:  
  image: redis:latest  
  expose:  
    - 6379  
  healthcheck:  
    test: ["CMD", "redis-cli", "ping"]  
    interval: 5s  
    timeout: 30s  
    retries: 50  
  restart: always
```



CONFIGURER APACHE AIRFLOW AVEC DOCKER-COMPOSE EN 5 MINUTES

INITIALISATION DE L'ENVIRONNEMENT

Avant de démarrer Airflow pour la première fois, vous devez préparer votre environnement, c'est-à-dire créer les fichiers nécessaires, répertoires et initialiser la base de données.

```
mkdir -p ./dags ./logs ./plugins  
echo -e "AIRFLOW_UID=$(id -u)" > .env
```

- La première commande crée trois répertoires dans le répertoire actuel (./), nommément dags, logs et plugins.
L'option -p signifie que les répertoires parents seront également créés si nécessaire.
- La deuxième commande crée un fichier nommé .env dans le répertoire actuel et y écrit une variable d'environnement appelée AIRFLOW_UID contenant l'ID utilisateur de l'utilisateur courant



CONFIGURER APACHE AIRFLOW AVEC DOCKER-COMPOSE EN 5 MINUTES

INITIALISATION DE L'ENVIRONNEMENT

- "**dags**" : C'est un répertoire où vous pouvez stocker des définitions de tâches d'Airflow, des workflows écrits en Python. Dans ce répertoire, vous pouvez créer un fichier tâche ou un groupe de tâches liées.
- "**logs**" : Ce répertoire est utilisé par Airflow pour stocker les fichiers de journalisation (logs) générés lors de l'exécution des tâches.
- "**plugins**" : Ce répertoire est utilisé pour stocker des plugins personnalisés pour Airflow. Les plugins sont des modules Python qui étendent les fonctionnalités d'Airflow, tels que des connecteurs personnalisés ou des opérateurs personnalisés.

```
> dags  
> logs  
> plugins  
⚙ .env  
➡ docker-compose.yaml
```





CONFIGURER APACHE AIRFLOW AVEC DOCKER-COMPOSE EN 5 MINUTES

INITIALISER LA BASE DE DONNÉES

- Sur tous les systèmes d'exploitation, vous devez exécuter les migrations de base de données et créer le premier compte utilisateur.

```
docker compose up airflow-init
```

- Une fois l'initialisation terminée, vous devriez voir un message comme celui-ci:

```
airflow-init_1      | Upgrades done
airflow-init_1      | Admin user airflow created
airflow-init_1      | 2.5.1
start_airflow-init_1 exited with code 0
```





CONFIGURER APACHE AIRFLOW AVEC DOCKER-COMPOSE EN 5 MINUTES

EXÉCUTION D'UN AIRFLOW

- Vous pouvez maintenant démarrer tous les services avec :

```
docker compose up -d
```

The screenshot shows the Airflow web interface. At the top left is the Airflow logo. At the top right are the time "12:35 UTC" and a "Log In" button. Below the header is a "Sign In" dialog box. The dialog has a placeholder "Enter your login and password below:" and two input fields: "Username" (with a user icon) and "Password" (with a lock icon). At the bottom of the dialog is a blue "Sign In" button. To the left of the dialog, there is a note in green text: "Il faut attendre quelques minutes à chaque fois le temps que le port 8080 charge !".



L'INTERFACE UTILISATEUR

- **DAGs View**

- Liste des DAG de votre environnement et ensemble de raccourcis vers des pages utiles.
- Il est possible de visualiser rapidement le nombre précis de tâches réussies, échouées ou en cours d'exécution.

Airflow DAGs Security Browse Admin Docs 21:11 UTC RH

DAGs

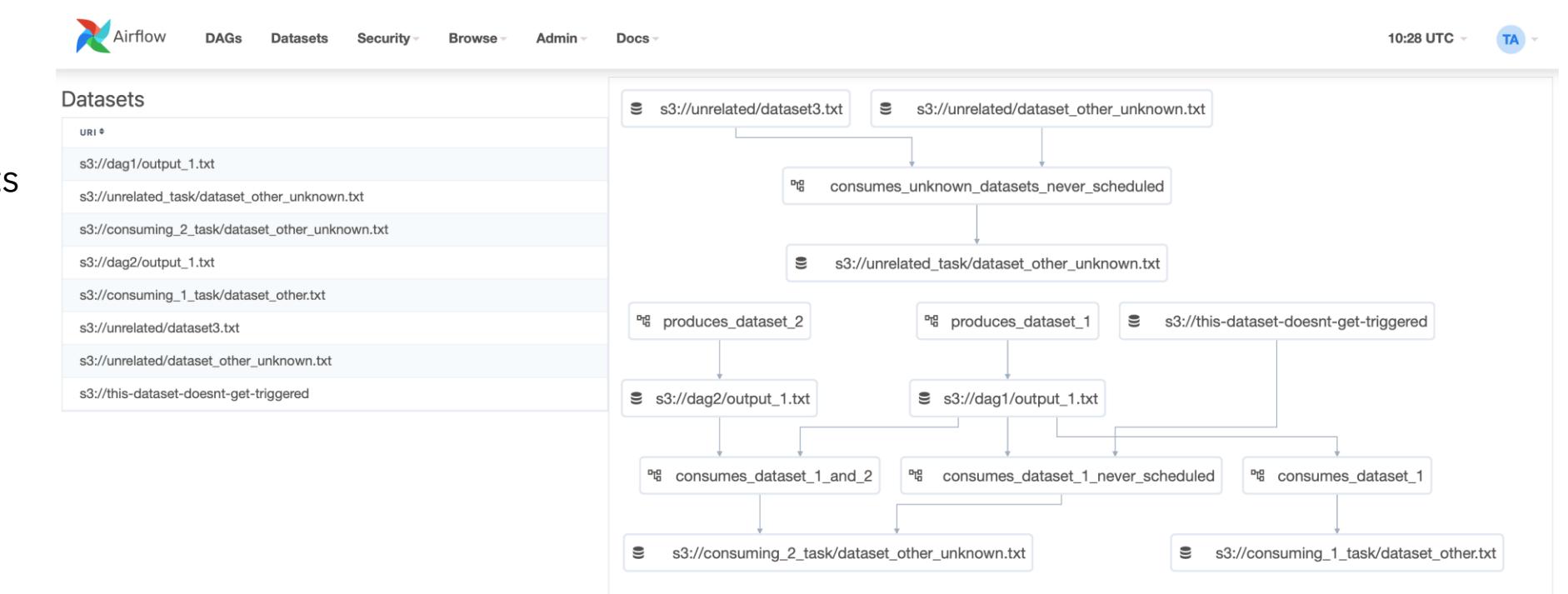
All 26	Active 10	Paused 16	Filter DAGs by tag	Search DAGs			
<i>DAG</i>	Owner	Runs	Schedule	Last Run	Recent Tasks	Actions	Links
example_bash_operator example example2	airflow	2	0 0 * * *	2020-10-26, 21:08:11	6	[green circle] [blue circle] [red circle]	[green arrow] [blue square] [red square] ...
example_branch_dop_operator_v3 example	airflow	0	* /1 * * *		0	[blue circle] [red circle]	[blue arrow] [blue square] [red square] ...
example_branch_operator example example2	airflow	1	@daily	2020-10-23, 14:09:17	11	[blue circle] [red circle]	[blue arrow] [blue square] [red square] ...
example_complex example example2 example3	airflow	1	None	2020-10-26, 21:08:04	37	[green circle] [red circle]	[green arrow] [blue square] [red square] ...
example_external_task_marker_child example	airflow	1	None	2020-10-26, 21:07:33	2	[blue circle] [red circle]	[blue arrow] [blue square] [red square] ...
example_external_task_marker_parent example	airflow	1	None	2020-10-26, 21:08:34	1	[green circle] [red circle]	[green arrow] [blue square] [red square] ...
example_kubernetes_executor example example2	airflow	0	None		0	[blue circle] [red circle]	[blue arrow] [blue square] [red square] ...
example_kubernetes_executor_config example3	airflow	1	None	2020-10-26, 21:07:40	5	[blue circle] [red circle]	[blue arrow] [blue square] [red square] ...
example_nested_branch_dag example	airflow	1	@daily	2020-10-26, 21:07:37	9	[blue circle] [red circle]	[blue arrow] [blue square] [red square] ...
example_passing_params_via_test_command example	airflow	0	* /1 * * *		0	[blue circle] [red circle]	[blue arrow] [blue square] [red square] ...



L'INTERFACE UTILISATEUR

- **Datasets View**

- Une liste combinée des ensembles de données actuels ainsi qu'un graphique illustrant comment ils sont produits et consommés par les DAG.
- En cliquant sur un ensemble de données dans la liste ou le graphique, celui-ci sera mis en évidence ainsi que ses relations, et la liste sera filtrée pour afficher l'historique récent des instances de tâches qui ont mis à jour cet ensemble de données et si cela a déclenché d'autres exécutions de DAG.

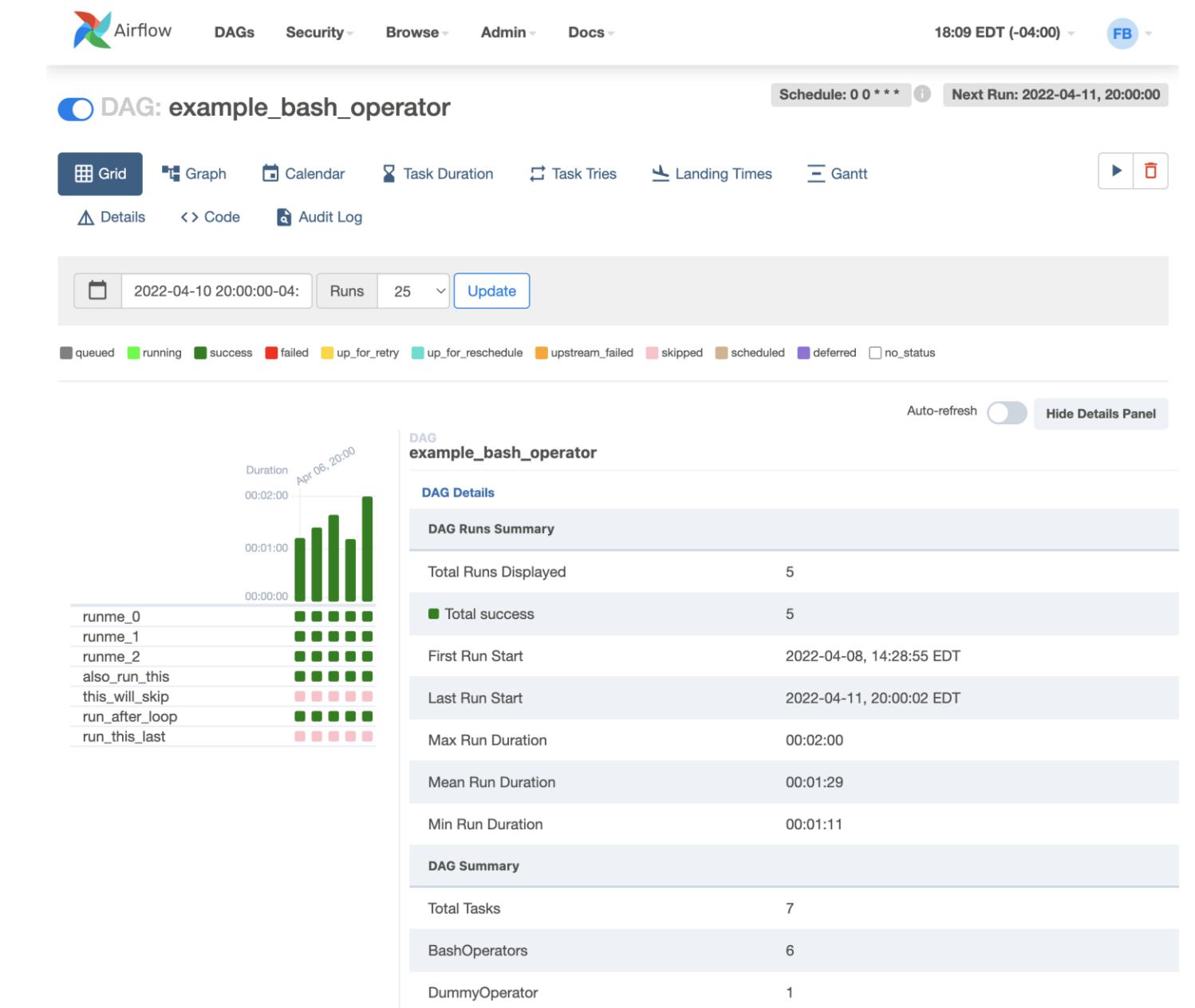




L'INTERFACE UTILISATEUR

- **Grid View**

- Un graphique en barres et une représentation en grille du DAG qui s'étend dans le temps.
- La rangée supérieure est un graphique des DAG Runs par durée, et en dessous, les instances de tâches.
- Si un pipeline est en retard, vous pouvez rapidement voir où se situent les différentes étapes et identifier celles qui bloquent.

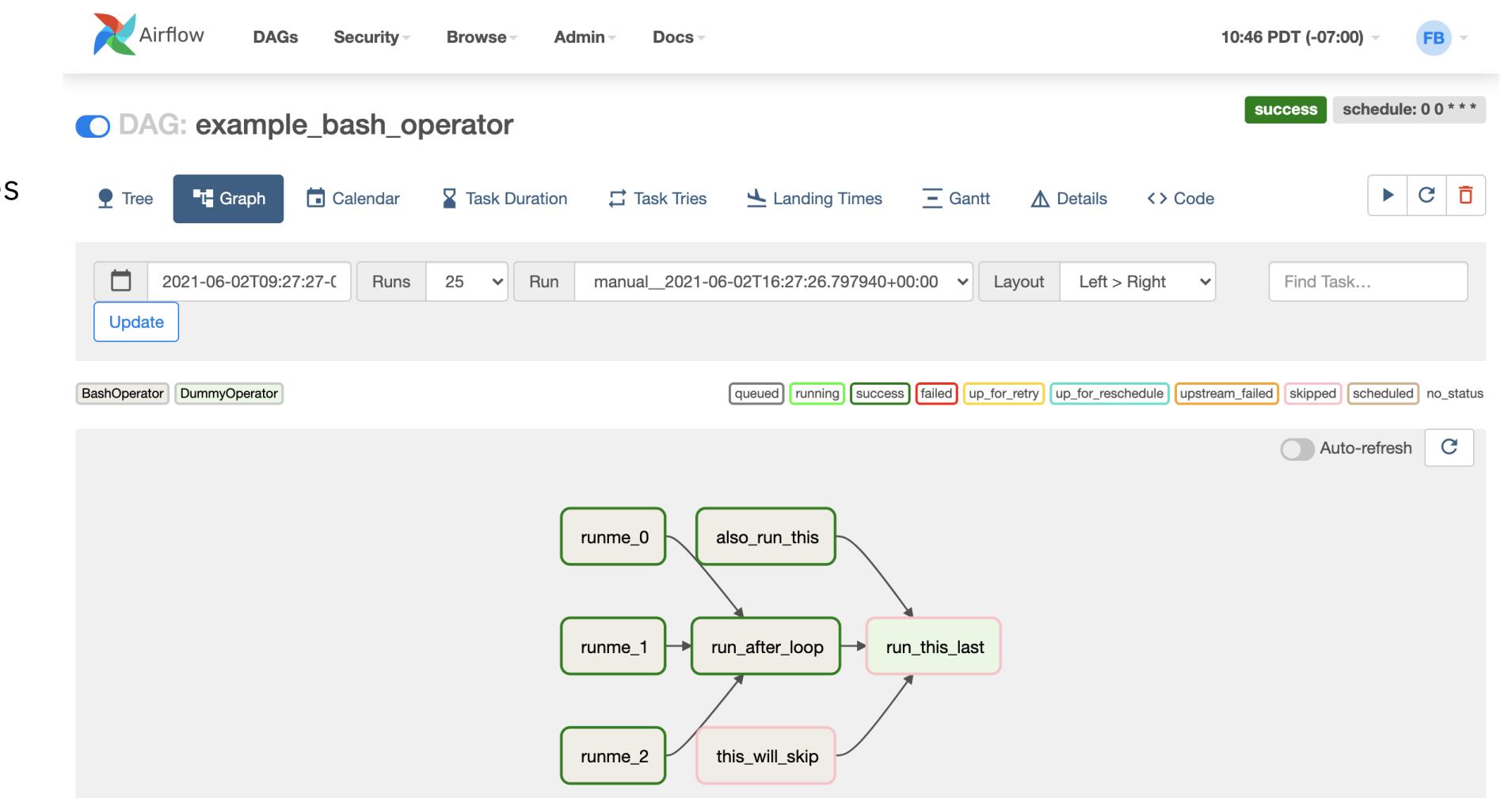




L'INTERFACE UTILISATEUR

- **Graph View**

- La vue graphique permet de visualiser les dépendances de votre DAG et leur état actuel pour une exécution spécifique.





L'INTERFACE UTILISATEUR

- **Variable View**

- La vue des variables vous permet de lister, créer, éditer ou supprimer les paires clé-valeur d'une variable utilisée pendant les jobs.
- La valeur d'une variable sera masquée si la clé contient des mots tels que ('password', 'secret', 'passwd', 'authorization', 'api_key', 'apikey', 'access_token') par défaut, mais peut être configurée pour être affichée en clair.

List Variable

Search▼

+ Actions◀ Record Count: 6

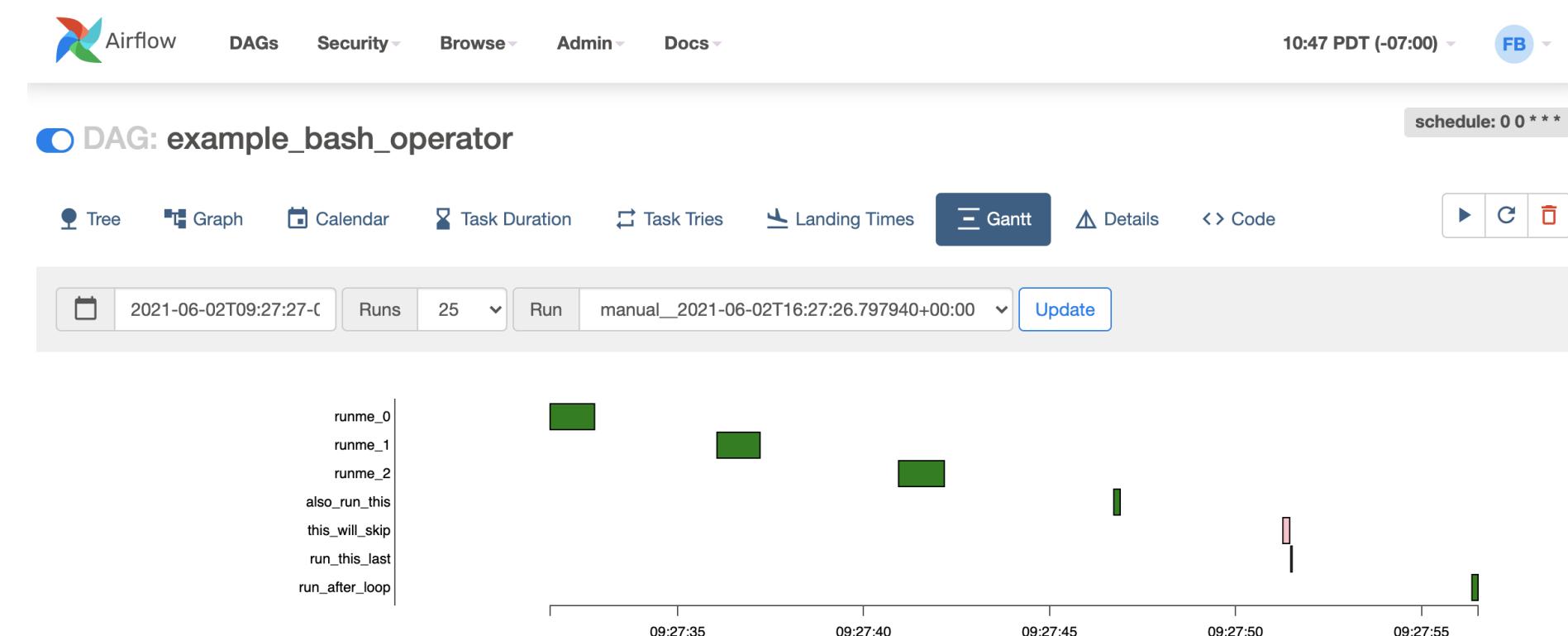
Key ↑	Val ↑	Is Encrypted ↑
airtable_api_key	*****	True
airtable_base_key	appzasdasdasdas	True
environment	prod	True
pipedrive_env	pipedrive	True
postgres_env	prod	True
snowflake_password	*****	True



L'INTERFACE UTILISATEUR

- **Gantt Chart**

- Le diagramme de Gantt permet d'analyser la durée et le chevauchement des tâches, et d'identifier rapidement les goulets d'étranglement ainsi que les zones où la majeure partie du temps est dépensée pour des exécutions de DAG spécifiques.

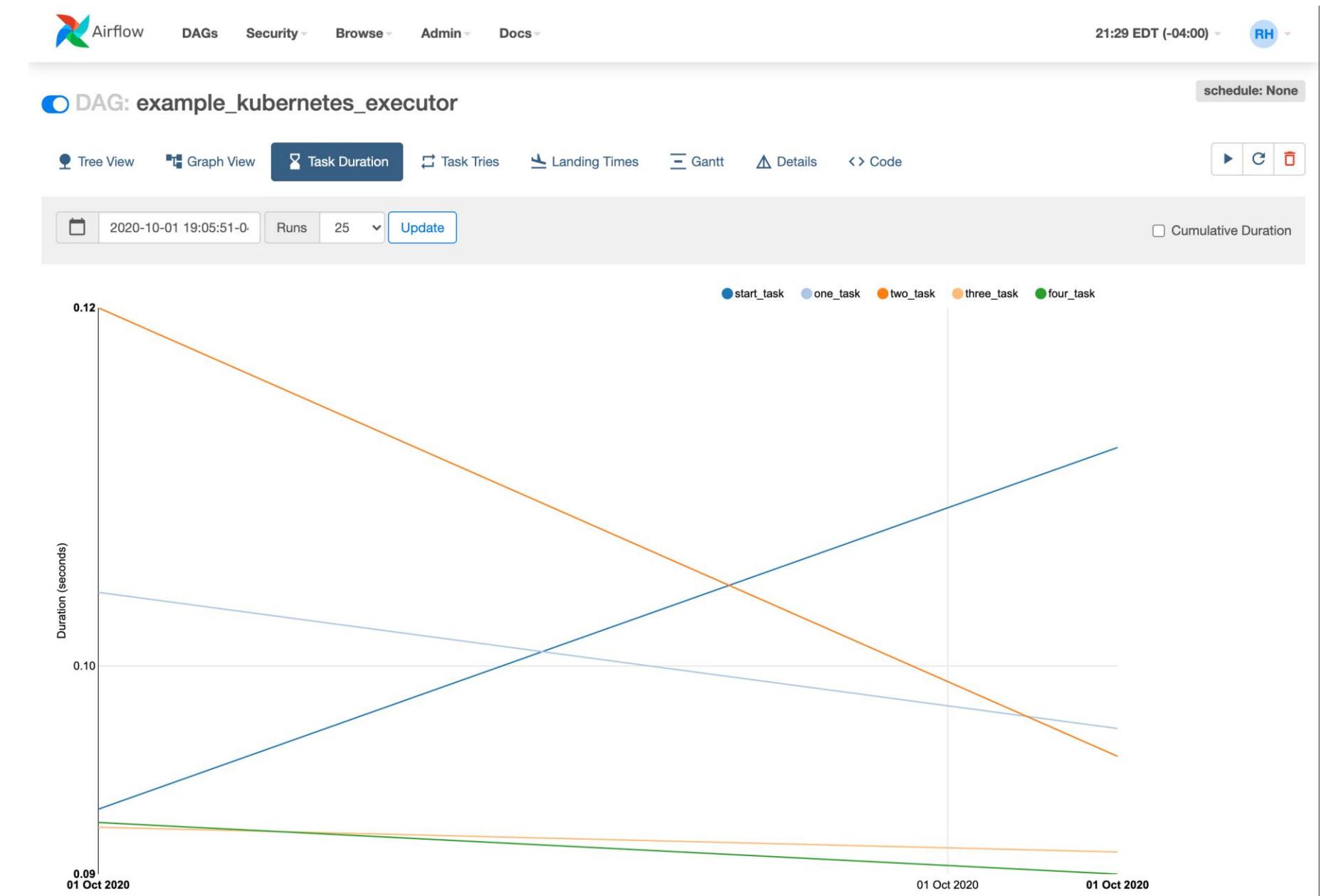




L'INTERFACE UTILISATEUR

- **Task Duration**

- La durée de chaque tâche pour les N dernières exécutions est affichée dans cette vue.
- Cela permet de repérer les valeurs aberrantes et de comprendre rapidement où le temps est principalement dépensé dans votre DAG sur plusieurs exécutions.





L'INTERFACE UTILISATEUR

- **Code View**

- La transparence est essentielle.
- Bien que le code de la pipeline soit stocké dans un contrôle de source, cette fonctionnalité permet d'accéder rapidement au code générant le DAG et fournit des informations supplémentaires.

The screenshot shows the Airflow web interface with the following details:

- Header:** Airflow, DAGs, Security, Browse, Admin, Docs, 10:47 PDT (-07:00), FB.
- Title:** DAG: example_bash_operator
- Toolbar:** Tree, Graph, Calendar, Task Duration, Task Tries, Landing Times, Gantt, Details, Code (selected).
- Code Editor:** Displays the Python code for the DAG:

```
1 #  
2 # Licensed to the Apache Software Foundation (ASF) under one  
3 # or more contributor license agreements. See the NOTICE file  
4 # distributed with this work for additional information  
5 # regarding copyright ownership. The ASF licenses this file  
6 # to you under the Apache License, Version 2.0 (the  
7 # "License"); you may not use this file except in compliance  
8 # with the License. You may obtain a copy of the License at  
9 #  
10 # http://www.apache.org/licenses/LICENSE-2.0  
11 #  
12 # Unless required by applicable law or agreed to in writing,  
13 # software distributed under the License is distributed on an  
14 # "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY  
15 # KIND, either express or implied. See the License for the  
16 # specific language governing permissions and limitations  
17 # under the License.  
18  
19 """Example DAG demonstrating the usage of the BashOperator."""  
20
```
- Buttons:** Toggle Wrap.



CONFIGURER APACHE AIRFLOW AVEC DOCKER-COMPOSE EN 5 MINUTES

EXÉCUTION D'UN AIRFLOW

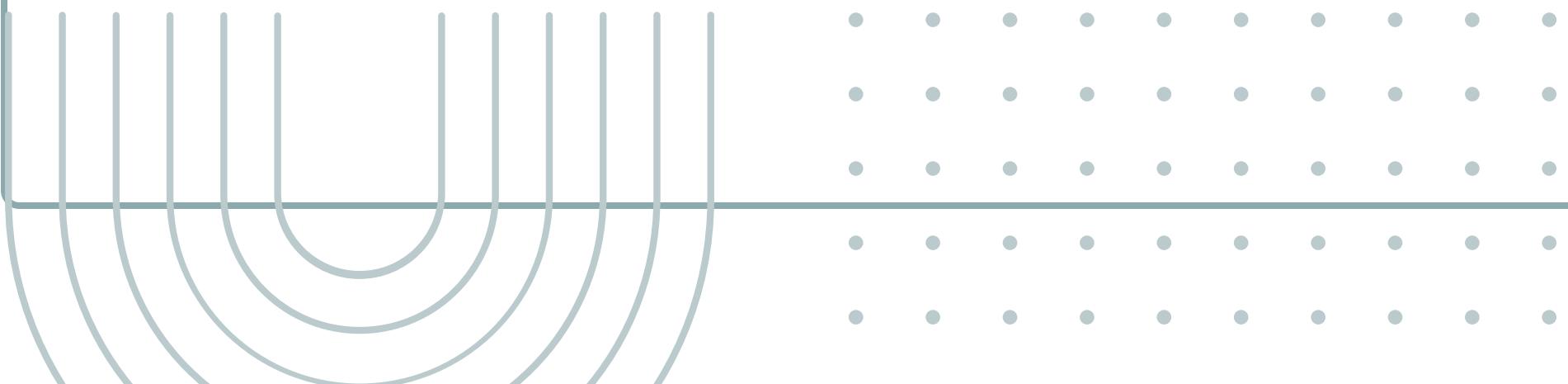
- Vous pouvez vérifier l'état des conteneurs et vous assurer qu'aucun conteneur n'est dans un état malsain:

```
docker container ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
eb668c646c87	apache/airflow:2.5.1	"/usr/bin/dumb-init ..."	6 minutes ago	Up 5 minutes (healthy)	8080/tcp	codespaces-blank-airflow-scheduler-1
9dbe9cea0e04	apache/airflow:2.5.1	"/usr/bin/dumb-init ..."	6 minutes ago	Up 5 minutes (healthy)	8080/tcp	codespaces-blank-airflow-worker-1
3f00148bfdc7	apache/airflow:2.5.1	"/usr/bin/dumb-init ..."	6 minutes ago	Up 5 minutes (healthy)	8080/tcp	codespaces-blank-airflow-triggerer-1
dad738406b62	apache/airflow:2.5.1	"/usr/bin/dumb-init ..."	6 minutes ago	Up 5 minutes (healthy)	0.0.0.0:8080->8080/tcp, :::8080->8080/tcp	codespaces-blank-airflow-webserver-1
f44065c0d1f1	postgres:13	"docker-entrypoint.s..."	2 hours ago	Up 9 minutes (healthy)	5432/tcp	codespaces-blank-postgres-1
1dd9cce842dc	redis:latest	"docker-entrypoint.s..."	2 hours ago	Up 9 minutes (healthy)	6379/tcp	codespaces-blank-redis-1

04.

Directed Acyclic Graphs (DAG)

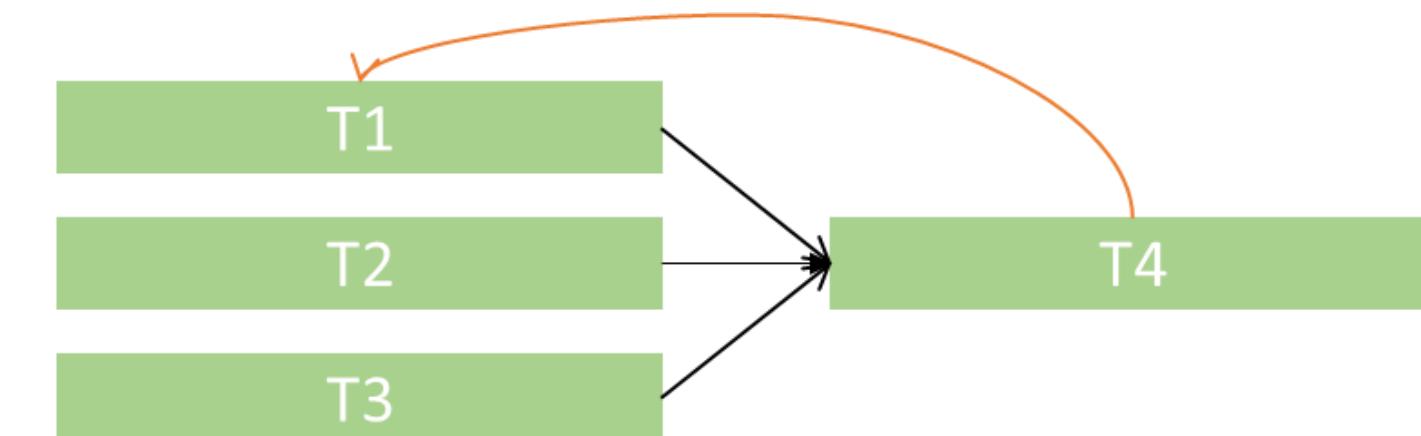
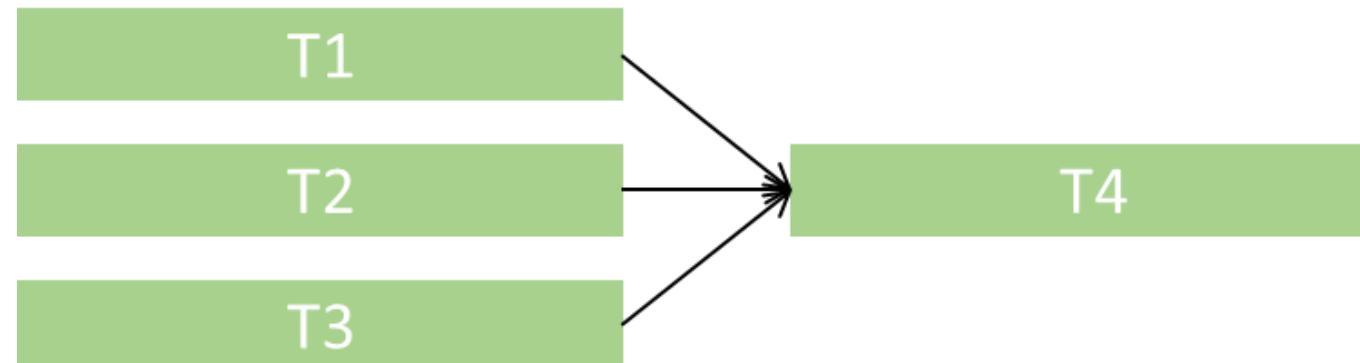




DAG (DIRECTED ACYCLIC GRAPH)

Un **DAG (Directed Acyclic Graph)** est le concept de base d'Airflow, rassemblant des tâches ensemble, organisées avec des dépendances et des relations pour dire comment elles doivent fonctionner.

Voici un exemple de base de DAG :



- Le DAG présent comprend quatre nœuds, à savoir T4, T1, T2 et T3, qui représentent des tâches. Des dépendances dirigées ou des bords existent entre ces tâches, où la T4 dépend de la T1, de la T2 et de la T3. Il est également important de noter que ce DAG ne présente pas de cycle.
- Ce qui est présenté ici n'est pas considéré comme un DAG, car il contient un cycle. La T4 dépend de la T1, tandis que la T1 dépend elle-même de la T4, créant ainsi un cycle. Ce type de configuration n'est pas pris en charge dans Airflow et entraînera une erreur si vous essayez de l'utiliser.



DAG (DIRECTED ACYCLIC GRAPH)

Déclaration d'un DAG

Il y a trois méthodes pour déclarer un DAG (Directed Acyclic Graph) : l'utilisation d'un gestionnaire de contexte qui ajoute implicitement le DAG à tout ce qui se trouve à l'intérieur :

```
with DAG(  
    "my_dag_name", start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),  
    schedule_interval="@daily", catchup=False  
) as dag:  
  
    op = DummyOperator(task_id="task")
```

Une autre méthode consiste à utiliser un constructeur standard en passant le DAG aux opérateurs que l'on utilise :

```
my_dag = DAG("my_dag_name", start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),  
             schedule_interval="@daily", catchup=False)  
  
op = DummyOperator(task_id="task", dag=my_dag)
```



DAG (DIRECTED ACYCLIC GRAPH)

Déclaration d'un DAG

Une autre option est d'utiliser le décorateur @dag pour transformer une fonction en un générateur de DAG :

```
@dag(start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),  
      schedule_interval="@daily", catchup=False)  
  
def generate_dag():  
  
    op = DummyOperator(task_id="task")  
  
dag = generate_dag()
```



DAG (DIRECTED ACYCLIC GRAPH)

Exécution des DAG

Les DAG s'exécutent de deux façons :

- Lorsqu'ils sont déclenchés manuellement ou via l'API.
- Selon un calendrier défini, qui est défini comme faisant partie du DAG.

Les DAG ne nécessitent pas de calendrier, mais il est très courant d'en définir un. L'argument **schedule_interval** permet de définir un calendrier pour le DAG, comme suit :

```
with DAG("my_daily_dag", schedule_interval="@daily"):
```

```
    ...
```

```
with DAG("my_daily_dag", schedule_interval="0 * * * *"):
```

```
    ...
```



DAG (DIRECTED ACYCLIC GRAPH)

Exécution des DAG

Lorsqu'un DAG est exécuté, une nouvelle instance appelée DAG Run est créée. Les **DAG Runs** pour un même DAG peuvent s'exécuter en parallèle, et chacun possède un intervalle de données défini qui spécifie la période de données sur laquelle les tâches doivent opérer.

➤ DAG RUN

- Une DAG Run est un objet représentant une instantiation du DAG dans le temps. Chaque fois que le DAG est exécuté, un DAG Run est créée et toutes les tâches qui y sont contenues sont exécutées.
- L'état de la DAG Run dépend des états des tâches. Chaque DAG Run est exécutée indépendamment des autres, ce qui signifie que vous pouvez exécuter plusieurs instances du DAG en même temps.



DAG (DIRECTED ACYCLIC GRAPH)

Exécution des DAG

- DAG RUN
- DAG RUN STATUS

Le statut d'une DAG Run est déterminé lorsque l'exécution du DAG est terminée. Il existe deux états finaux possibles pour une DAG Run :

- réussite** si l'état de toutes les feuilles est réussite ou ignoré,
- échec** si l'état de l'une des feuilles est échec ou upstream_failed.



BEST PRACTICES

- **Évitez de faire des calculs dans la définition de votre DAG**
 - Les DAG d'Airflow sont écrits en Python, offrant ainsi une grande flexibilité lors de la création. Cependant, cette approche présente un inconvénient:
 - Airflow doit exécuter le fichier DAG Python pour obtenir le DAG correspondant. En outre, Airflow doit relire le fichier régulièrement pour prendre en compte les modifications apportées au DAG et synchroniser les changements avec son état interne.
 - Cette analyse répétée des fichiers DAG peut entraîner des problèmes si l'un d'entre eux prend beaucoup de temps à charger, par exemple, si des calculs longs ou lourds sont effectués lors de la définition du DAG.



BEST PRACTICES

- Évitez de faire des calculs dans la définition de votre DAG

➤ Effectuer des calculs dans la définition du DAG (inefficace)

```
...  
task1 = PythonOperator(...)  
  
# Ce long calcul sera effectué à chaque fois que le DAG est analysé  
  
my_value = do_some_long_computation()  
  
task2 = PythonOperator(op_kwargs={"my_value": my_value})  
  
...
```

Ce type d'implémentation fera exécuter à Airflow do_some_long_computation à chaque fois que le fichier DAG est chargé, bloquant tout le processus d'analyse du DAG jusqu'à ce que le calcul soit terminé. Une façon d'éviter ce problème est de reporter le calcul à l'exécution de la tâche qui nécessite la valeur calculée.



BEST PRACTICES

- Évitez de faire des calculs dans la définition de votre DAG

➤ Effectuer des calculs au sein des tâches (plus efficace)

```
def _my_not_so_efficient_task(value, ...):  
    ...  
    PythonOperator(  
        task_id="my_not_so_efficient_task",  
        ...  
        op_kwargs={  
            # Ici, la valeur sera calculée à chaque fois que le DAG est analysé  
            "value": calc_expensive_value()  
        }  
    )
```

```
def _my_more_efficient_task(...):  
    value = calc_expensive_value() ←  
    ...  
    PythonOperator(  
        task_id="my_more_efficient_task",  
        python_callable=_my_more_efficient_task, ←  
        ...  
    )
```

En déplaçant le calcul dans la tâche, la valeur ne sera calculée que lorsque la tâche sera exécutée



BEST PRACTICES

- Évitez de faire des calculs dans la définition de votre DAG
 - Effectuer des calculs au sein des tâches (plus efficace)
- Une autre approche consisterait à écrire un hook/opérateur personnalisé, qui ne récupère les informations d'identification que lorsqu'elles sont nécessaires pour l'exécution, mais cela peut nécessiter un peu plus de travail.
- Dans des cas plus subtils, une configuration peut être chargée à partir d'une source de données externe ou d'un système de fichiers dans le fichier DAG principal.
- Par exemple, les informations d'identification peuvent être chargées à partir du métastore d'Airflow et partagées entre quelques tâches en utilisant une méthode similaire à celle-ci.



BEST PRACTICES

- Évitez de faire des calculs dans la définition de votre DAG

➤ Récupérer les informations d'identification depuis le métastore dans la définition du DAG (inefficace)

```
from airflow.hooks.base_hook import BaseHook

# Cet appel interrogera la base de données chaque fois que le DAG est analysé.

api_config = BaseHook.get_connection("my_api_conn")

api_key = api_config.login

api_secret = api_config.password

task1 = PythonOperator(
    op_kwargs={"api_key": api_key, "api_secret": api_secret},
    ...
)
...
```



BEST PRACTICES

- Évitez de faire des calculs dans la définition de votre DAG
 - Récupérer les informations d'identification depuis le métastore dans la définition du DAG (inefficace)
- Cependant, cette approche présente un inconvénient : elle récupère les informations d'identification de la base de données à chaque fois que le DAG est analysé, au lieu de ne le faire qu'au moment de son exécution.
- Ainsi, des requêtes répétées seront effectuées toutes les 30 secondes environ (en fonction de la configuration d'Airflow) pour récupérer ces informations, ce qui peut entraîner des problèmes de performance.
- Pour éviter ces problèmes, il est généralement recommandé de reporter la récupération des informations d'identification à l'exécution de la fonction de la tâche.



BEST PRACTICES

- Évitez de faire des calculs dans la définition de votre DAG

➤ **Récupération des informations d'identification à l'intérieur d'une tâche (plus efficace)**

```
from airflow.hooks.base_hook import BaseHook

def _task1(conn_id, **context):
    # Cette requête n'interrogera la base de données que lorsque la tâche sera exécutée.

    api_config = BaseHook.get_connection(conn_id)

    api_key = api_config.login

    api_secret = api_config.password

    ...

task1 = PythonOperator(op_kwargs={"conn_id": "my_api_conn"})
```



BEST PRACTICES

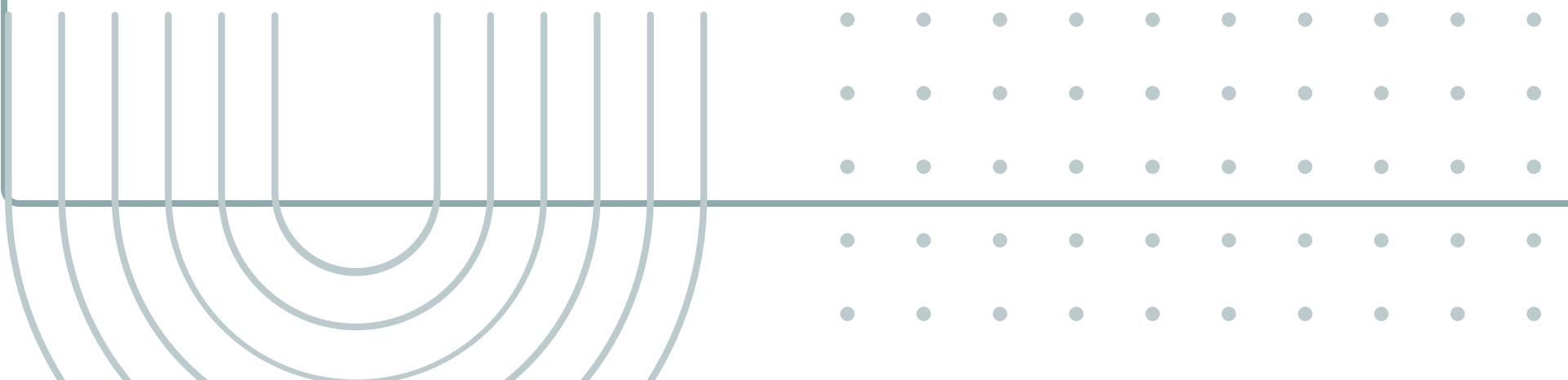
- **Évitez de faire des calculs dans la définition de votre DAG**

- **Récupération des informations d'identification à l'intérieur d'une tâche (plus efficace)**

- De cette façon, les informations d'identification ne sont récupérées que lorsque la tâche est effectivement exécutée, ce qui rend le DAG beaucoup plus efficace.
 - Ce type de propagation de calculs, dans lequel des calculs sont accidentellement inclus dans les définitions de DAG, peut être subtil et nécessite de la vigilance pour l'éviter.
 - De plus, certains cas peuvent être plus problématiques que d'autres : il est possible que le chargement répété d'un fichier de configuration à partir d'un système de fichiers local ne pose pas de problème, mais le chargement répété à partir d'un stockage cloud ou d'une base de données peut être moins souhaitable.

05.

TASKS & OPERATORS



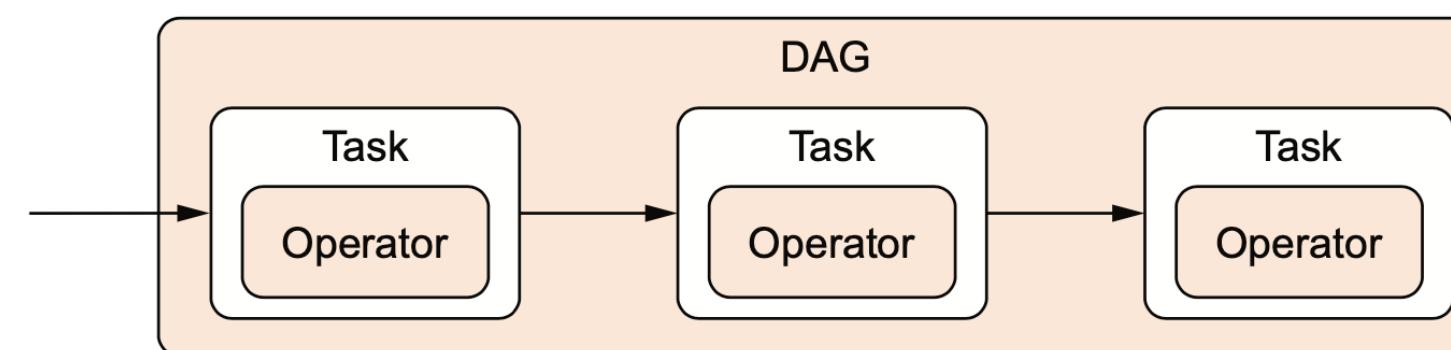


Tasks (Tâches)

Une tâche est l'unité de base d'exécution dans Airflow. Les tâches sont organisées en DAG, puis des dépendances amont et aval sont définies entre elles afin d'exprimer l'ordre dans lequel elles doivent être exécutées.

Il existe trois types de tâches de base :

- **Les opérateurs**, des modèles de tâches prédéfinis que vous pouvez rapidement combiner pour construire la plupart des parties de vos DAG.
- **Les capteurs (Sensors)**, une sous-classe spéciale d'opérateurs qui attendent uniquement un événement externe.
- **Les tâches TaskFlow-decorated** avec @task, qui sont des fonctions Python personnalisées encapsulées en tant que tâches.





Tasks (Tâches)

Instances de tâches

De la même manière qu'un DAG est instancié en un **DAG Run** à chaque fois qu'il s'exécute, les tâches sous un DAG sont instanciées en Task Instances.

Une instance de tâche est une exécution spécifique de cette tâche pour un DAG donné (et donc pour un intervalle de données donné). Elles représentent également l'état d'une tâche, indiquant à quel stade de son cycle de vie elle se trouve.

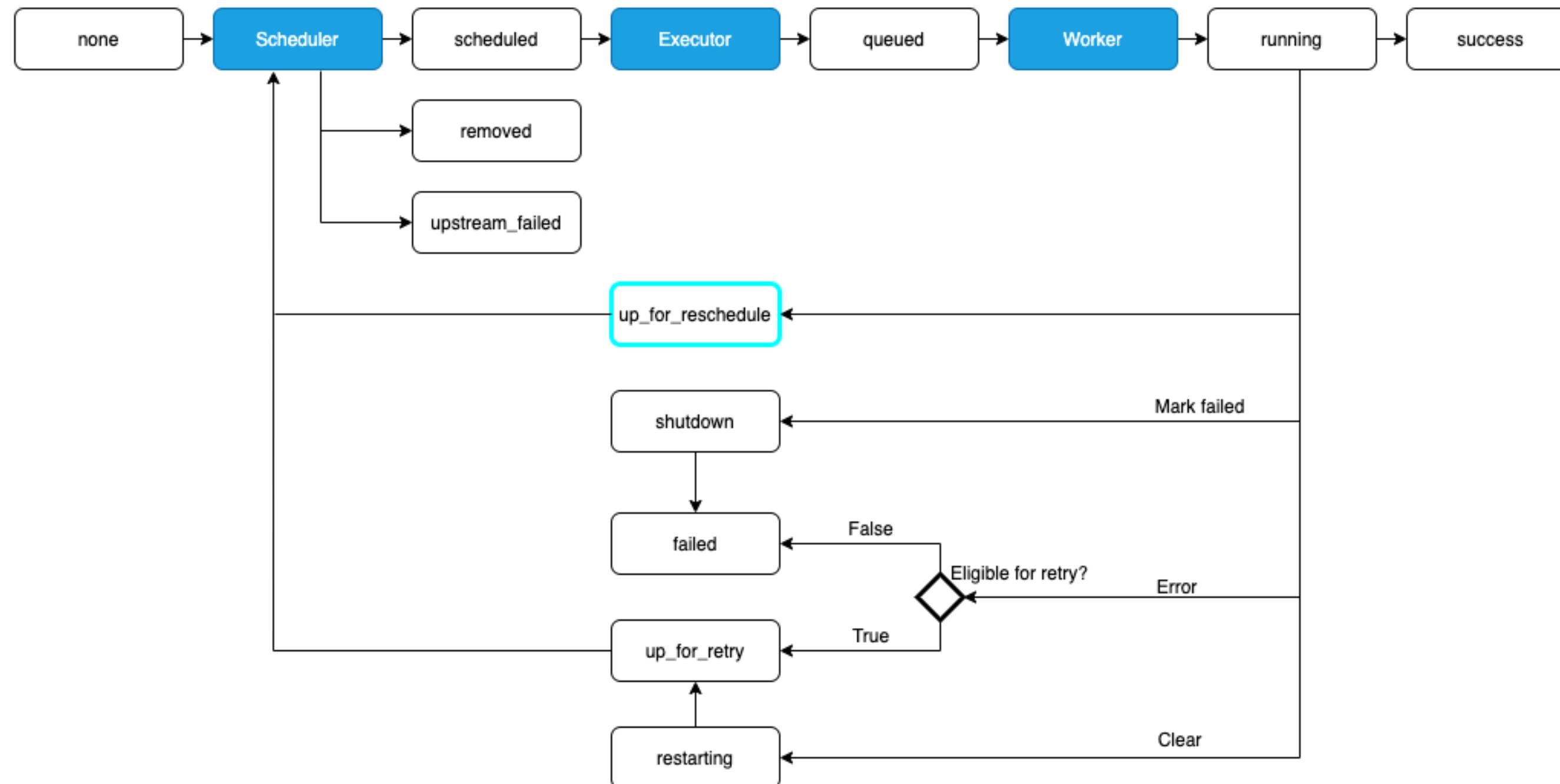
Les états possibles pour une instance de tâche sont :

deferred failed queued removed restarting running scheduled shutdown skipped success up_for_reschedule up_for_retry upstream_failed no_status





Tasks (Tâches)





LES DÉPENDANCES ENTRE TACHE / OPÉRATEURS

Dans Airflow, les dépendances entre opérateurs sont définies à l'aide d'objets appelés "objets de dépendance" ou "objets de liaison". Ces objets sont utilisés pour définir la relation entre les opérateurs dans un DAG (Directed Acyclic Graph).

Il existe plusieurs types d'objets de dépendance dans Airflow :

01

>> ou `set_downstream()`

Cette dépendance est utilisée pour définir un opérateur en aval d'un autre opérateur. Cela signifie que l'opérateur en amont doit être exécuté avant que l'opérateur en aval ne soit exécuté

03

<< ou `set_upstream()`

Cette dépendance est utilisée pour définir un opérateur en amont d'un autre opérateur. Cela signifie que l'opérateur en amont doit être exécuté après que l'opérateur en aval ait été exécuté.

02

>> et << ou `set_downstream()`

Cette dépendance permet de définir une relation bidirectionnelle entre deux opérateurs. Cela signifie que les opérateurs doivent être exécutés dans un ordre spécifique l'un par rapport à l'autre.

04

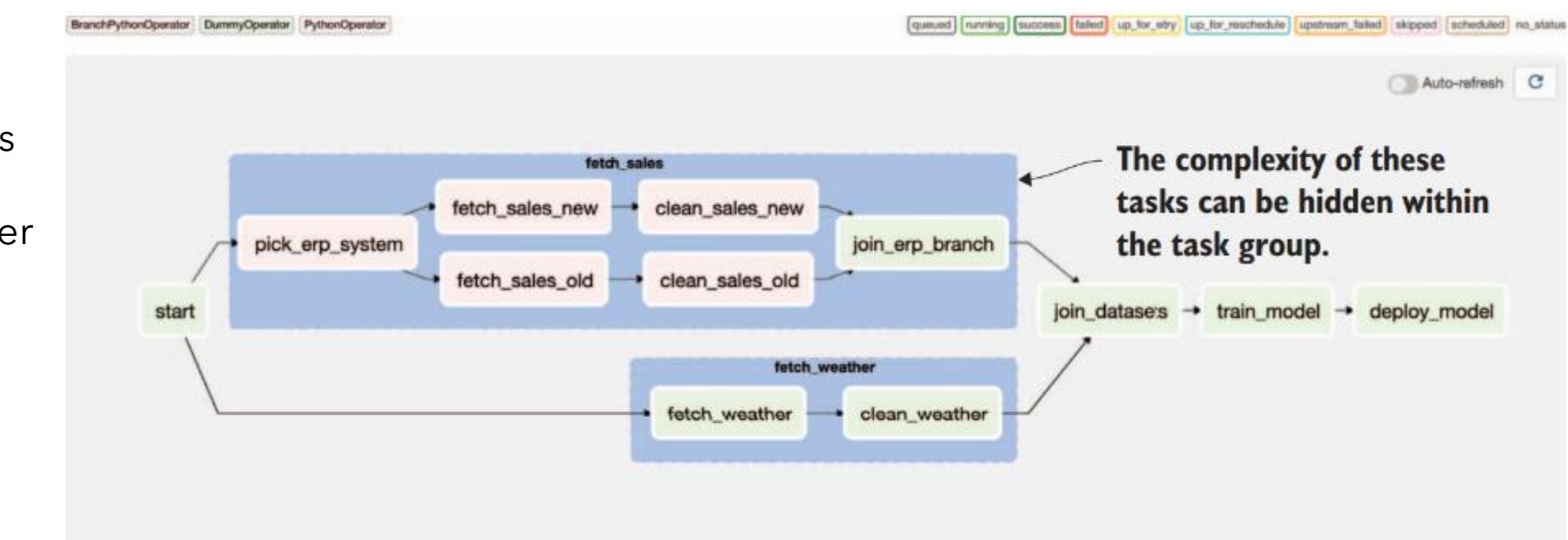
`set_downstream()` et `set_upstream()`

Cette dépendance permet de définir une relation de dépendance entre plusieurs opérateurs. Cela signifie que l'opérateur en amont doit être exécuté avant que tous les opérateurs en aval ne soient exécutés.

REGROUER LES TÂCHES CONNEXES EN UTILISANT DES GROUPES DE TÂCHES



- **Utiliser des groupes de tâches pour regrouper visuellement des tâches**
- Bien que cet exemple soit relativement simple, la fonctionnalité de groupes de tâches peut être très efficace pour réduire le bruit visuel dans des cas plus complexes.
- Par exemple, dans notre DAG pour l'entraînement de modèles d'apprentissage automatique, nous avons créé un nombre considérable de tâches pour récupérer et nettoyer les données météorologiques et de vente provenant de différents systèmes. Les groupes de tâches nous permettent de réduire la complexité apparente de ce DAG en regroupant les tâches liées aux ventes et aux données météorologiques dans leurs groupes de tâches respectifs.
- Cela nous permet de masquer la complexité des tâches de récupération de données par défaut, mais de zoomer sur les tâches individuelles lorsque cela est nécessaire.



REGROUER LES TÂCHES CONNEXES EN UTILISANT DES GROUPES DE TÂCHES



- Les DAG Airflow complexes, en particulier ceux générés à l'aide de méthodes de fabrication, peuvent souvent devenir difficiles à comprendre en raison de structures de DAG complexes ou du grand nombre de tâches impliquées.
- Pour aider à organiser ces structures complexes, Airflow 2 dispose d'une nouvelle fonctionnalité appelée groupes de tâches. Les groupes de tâches permettent efficacement de regrouper (visuellement) des ensembles de tâches en de plus petits groupes, rendant ainsi la structure de votre DAG plus facile à superviser et à comprendre.
- Vous pouvez créer des groupes de tâches en utilisant le gestionnaire de contexte TaskGroup. Par exemple, en reprenant notre exemple précédent de fabrication de tâches, nous pouvons regrouper les tâches générées pour chaque ensemble de données comme suit

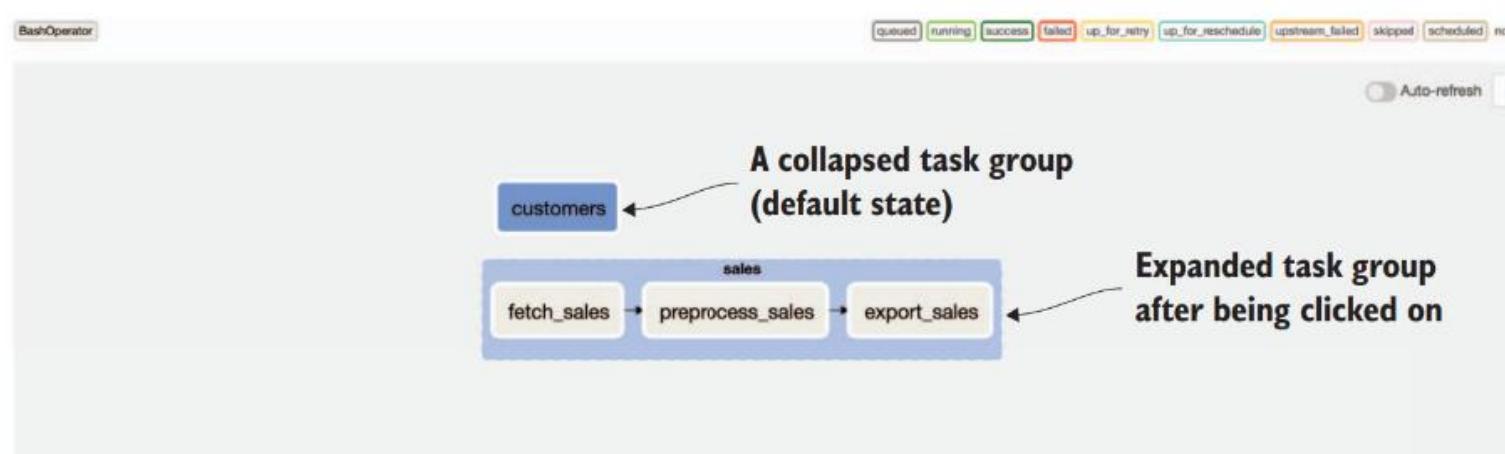


REGROUER LES TÂCHES CONNEXES EN UTILISANT DES GROUPES DE TÂCHES

- Utiliser des groupes de tâches pour regrouper visuellement des tâches

Cela regroupe efficacement l'ensemble des tâches générées pour les ensembles de données "ventes" et "clients" en deux groupes de tâches, un pour chaque ensemble de données.

En conséquence, les tâches regroupées sont affichées comme un seul groupe de tâches condensé dans l'interface Web, qui peut être développé en cliquant sur le groupe correspondant.



```
...
for dataset in ["sales", "customers"]:
    with TaskGroup(dataset, tooltip=f"Tasks for processing {dataset}"):
        generate_tasks(
            dataset_name=dataset,
            raw_dir="/data/raw",
            processed_dir="/data/processed",
            output_dir="/data/output",
            preprocess_script=f"preprocess_{dataset}.py",
            dag=dag,
```



Task : Best practices

Bien que les tâches d'Airflow puissent effectuer une grande partie du travail de backfilling et de relance, il est important de s'assurer que ces tâches remplissent certaines propriétés clés pour garantir des résultats appropriés.

deux des propriétés sont notamment importantes pour les tâches Airflow :

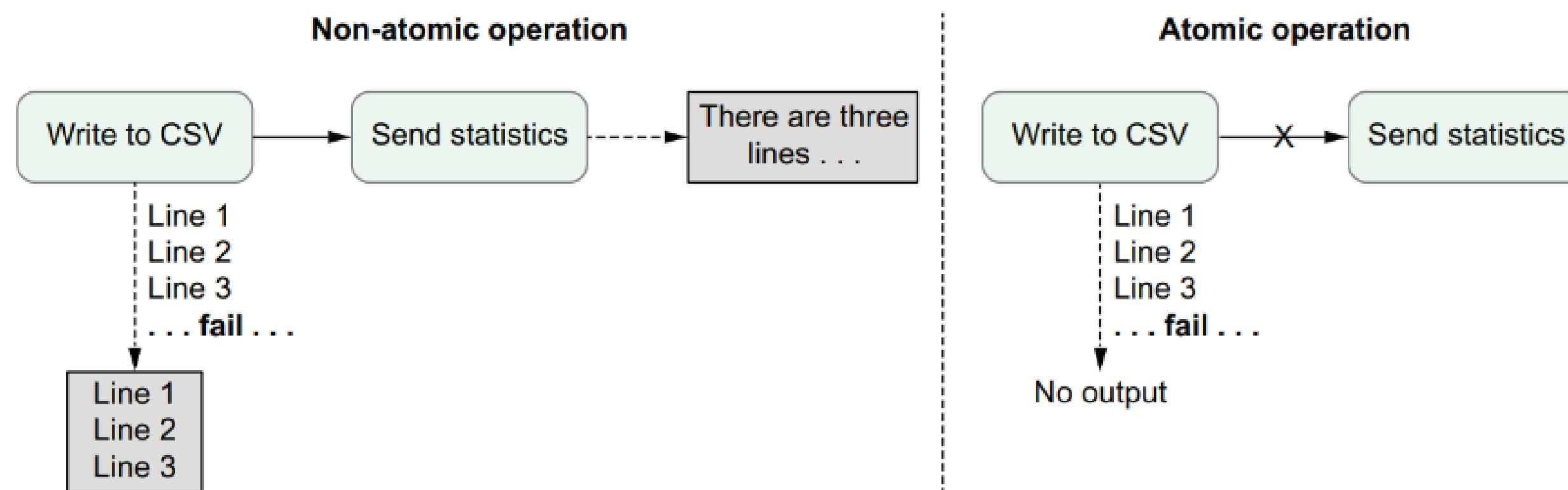
l'*atomicité* et **l'*idempotence***.



Task : Best practices

Atomicité

Le terme d'atomicité est fréquemment utilisé dans les systèmes de bases de données, où une transaction atomique est considérée comme une série indivisible et irréductible d'opérations de base de données telle que soit toutes les opérations sont effectuées, soit rien ne se produit. De même, dans Airflow, les tâches doivent être définies de sorte qu'elles réussissent et produisent un résultat approprié ou échouent de manière à ne pas affecter l'état du système.





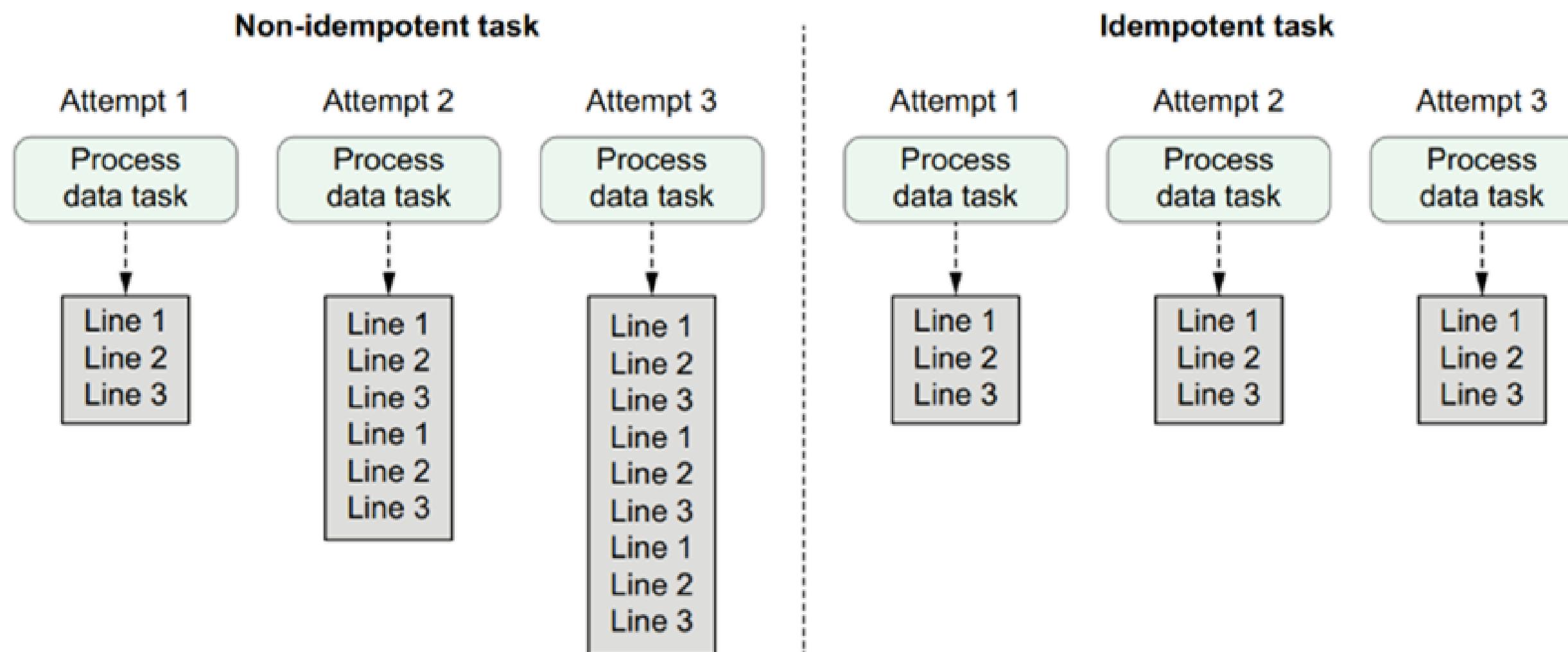
Task : Best practices

Idempotence

Les tâches sont dites idempotentes si l'appel de la même tâche plusieurs fois avec les mêmes entrées n'a aucun effet supplémentaire.

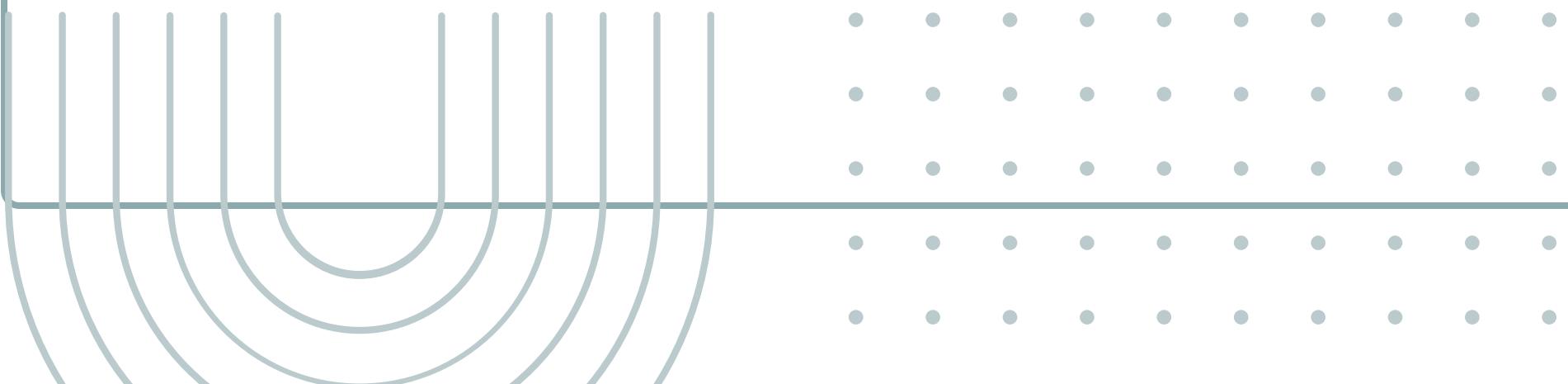
Cela signifie que la relance d'une tâche sans modifier les entrées ne devrait pas modifier la sortie globale.

Par exemple, considérons notre dernière implémentation de la tâche `fetch_events`, qui récupère les résultats pour une seule journée et les écrit dans notre ensemble de données partitionné.



06.

EXECUTORS





DIFFERENTS TYPES D'EXÉCUTEURS

Une fois qu'un DAG est défini, les étapes suivantes doivent être effectuées afin que les tâches à l'intérieur de ce DAG s'exécutent :

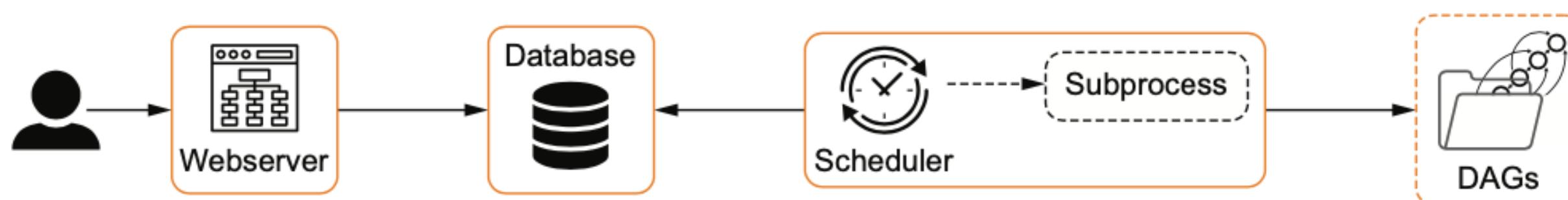
- La **base de métadonnées** enregistre en arrière-plan toutes les tâches d'un DAG ainsi que leur statut correspondant (en attente, planifiées, en cours d'exécution, réussies, échouées, etc.).
- Le **planificateur** (scheduler) lit la base de données de métadonnées pour vérifier l'état de chaque tâche et décider de ce qui doit être fait et quand.
- **L'exécuteur** (executor) travaille en étroite collaboration avec le planificateur pour déterminer les ressources qui permettront réellement de terminer ces tâches (en utilisant un processus **worker**) lorsqu'elles sont en attente.
- un exécuteur est responsable de l'exécution effective des tâches définies dans un DAG.
- Airflow prend en charge plusieurs exécuteurs qui peuvent être configurés en fonction des besoins du flux de travail.



EXÉCUTEURS SÉQUENTIELS AVEC SQLITE

SequentialExecutor : L'exécuteur séquentiel est l'exécuteur par défaut lorsque vous installez Airflow manuellement .

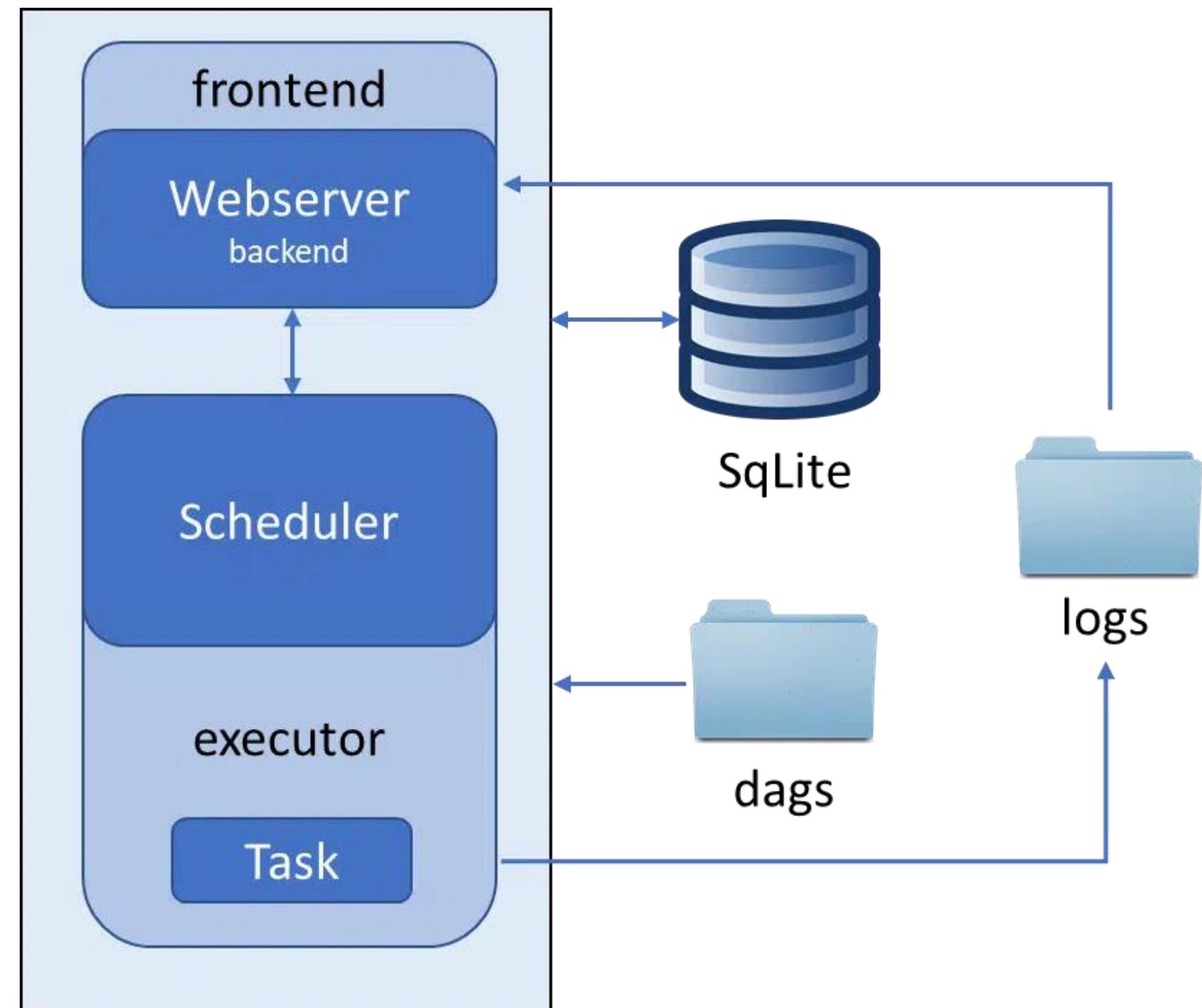
Si l'on dispose d'un serveur web, d'un planificateur (scheduler) et **d'une base de données SQLite**, il est impossible d'exécuter plusieurs tâches en même temps avec le DAG. Le planificateur exécutera seulement [une tâche à la fois](#).



With the SequentialExecutor, all components must run on the same machine.



EXÉCUTEURS SÉQUENTIELS AVEC SQLITE

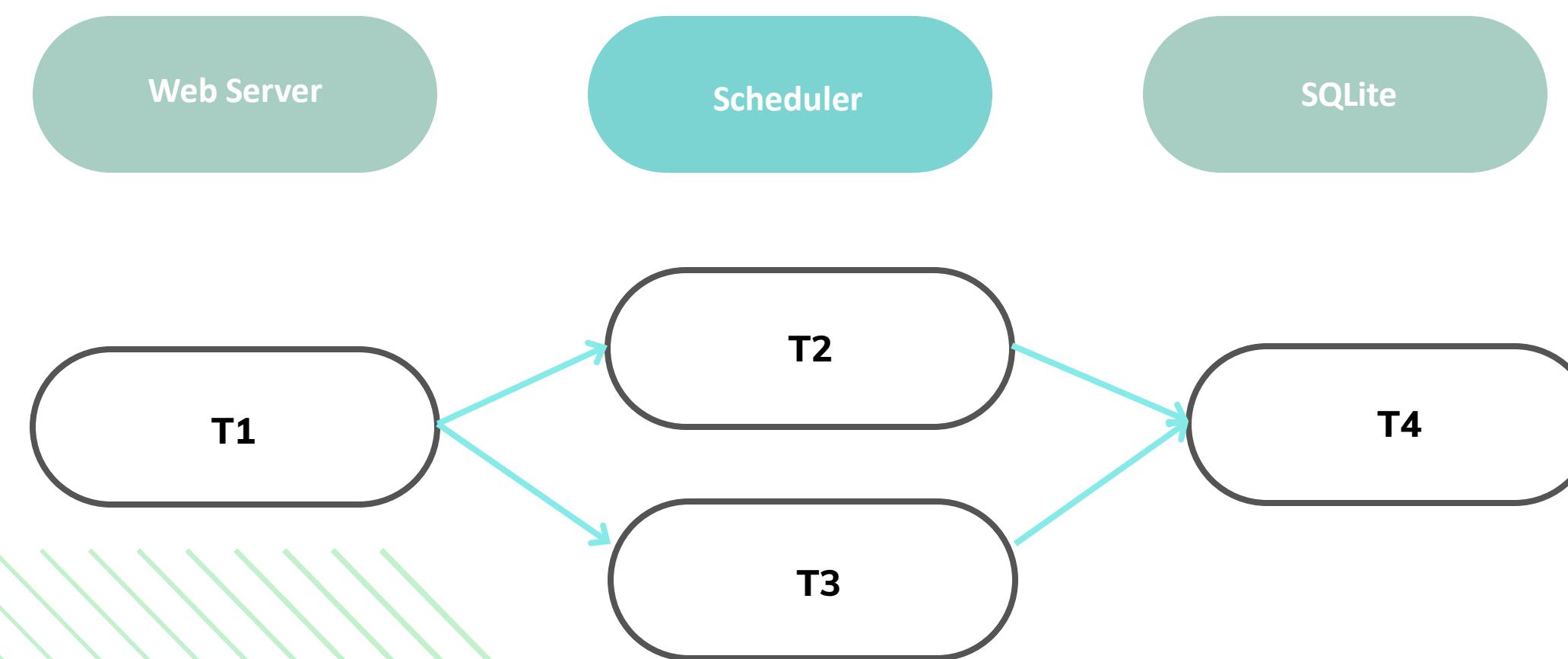




EXÉCUTEURS SÉQUENTIELS AVEC SQLITE

SequentialExecutor : L'exécuteur séquentiel est l'exécuteur par défaut lorsque vous installez Airflow manuellement .

Si l'on dispose d'un serveur web, d'un planificateur (scheduler) et **d'une base de données SQLite**, il est impossible d'exécuter plusieurs tâches en même temps avec le DAG. Le planificateur exécutera seulement une tâche à la fois.





EXÉCUTEURS SÉQUENTIELS AVEC SQLITE

SequentialExecutor : L'exécuteur séquentiel est l'exécuteur par défaut lorsque vous installez Airflow manuellement .

Si l'on dispose d'un serveur web, d'un planificateur (scheduler) et **d'une base de données SQLite**, il est impossible d'exécuter plusieurs tâches en même temps avec le DAG. Le planificateur exécutera seulement une tâche à la fois.



EXÉCUTEURS SÉQUENTIELS AVEC SQLITE

- Pour configurer l'exécuteur séquentiel avec SQLite, vous devez ajouter la configuration suivante dans votre fichier de configuration Airflow (généralement appelé airflow.cfg) :

```
executor = SequentialExecutor
sql_alchemy_conn =sqlite:///chemin/vers/votre/base/de/données/airflow.db
```

- La première ligne définit l'exécuteur à utiliser comme étant l'exécuteur séquentiel. La deuxième ligne définit la connexion à la base de données SQLite utilisée par Airflow. Vous devez remplacer le chemin avec le chemin absolu de votre base de données SQLite.



EXÉCUTEURS SÉQUENTIELS AVEC SQLITE

CONFIGURATION EXÉCUTEURS SÉQUENTIELS SUR DOCKER-COMPOSE.YAML

Pour configurer l'exécuteur séquentiels dans un fichier docker-compose.yaml, il suffit de :

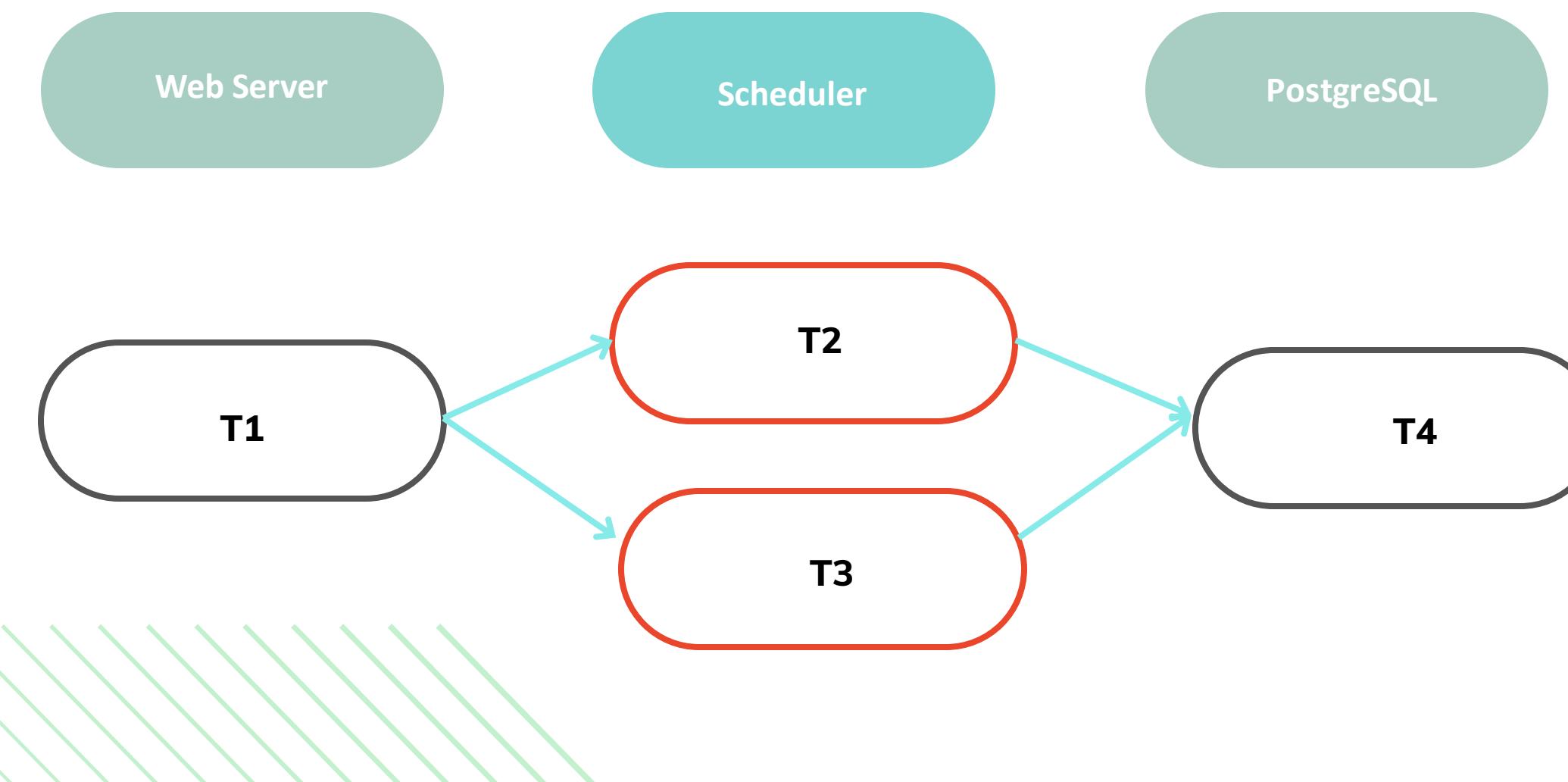
- Remplacer : '**AIRFLOW__CORE__EXECUTOR: CeleryExecutor**' avec '**AIRFLOW__CORE__EXECUTOR: SequentialExecutor**'
- Supprimer : '**AIRFLOW__CELERY__RESULT_BACKEND**' et '**AIRFLOW__CELERY__BROKER_URL**'

```
environment:  
  &airflow-common-env  
    AIRFLOW__CORE__EXECUTOR: SequentialExecutor  
    AIRFLOW__DATABASE__SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow  
      # For backward compatibility, with Airflow <2.3  
      AIRFLOW__CORE__SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow  
      #AIRFLOW__CELERY__RESULT_BACKEND: db+postgresql://airflow:airflow@postgres/airflow  
      #AIRFLOW__CELERY__BROKER_URL: redis://:@redis:6379/0
```



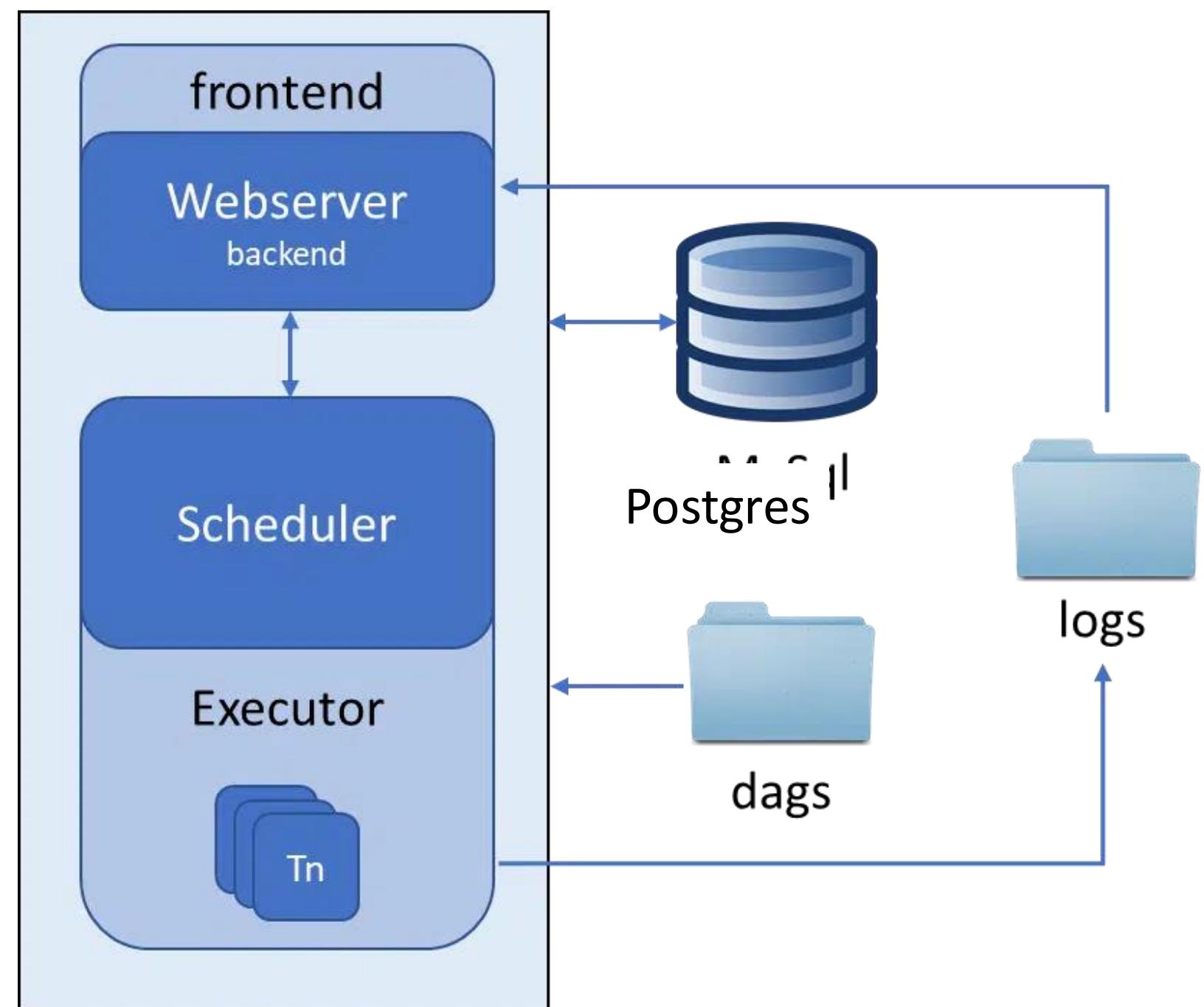
EXÉCUTEURS LOCAUX AVEC POSTGRESQL

LocalExecutor : L'exécuteur local va un peu plus loin que l'exécuteur séquentiel, car il vous permet d'exécuter plusieurs tâches en même temps . Mais cela reste toutefois limité à une seule machine.





EXÉCUTEURS LOCAUX AVEC POSTGRESQL





EXÉCUTEURS LOCAUX AVEC POSTGRESQL

Airflow prend en charge l'utilisation d'un **exécuteur local** avec **PostgreSQL** comme base de données. Voici comment configurer l'exécuteur local avec PostgreSQL :

- Installez PostgreSQL sur votre système d'exploitation.
- Créez une base de données PostgreSQL pour Airflow.
- Modifiez votre fichier de configuration Airflow (généralement appelé airflow.cfg) et ajoutez la configuration suivante :

```
executor = LocalExecutor  
sql_alchemy_conn =postgresql://username:password@hostname/database_name
```

Initialisez la base de données Airflow en exécutant la commande suivante

```
airflow initdb
```



EXÉCUTEURS LOCAUX AVEC POSTGRESQL

Démarrez le serveur Airflow en exécutant la commande suivante :

```
airflow webserver -p  
    <num_port>
```



EXÉCUTEURS LOCAUX AVEC POSTGRESQL

CONFIGURATION EXÉCUTEURS LOCAUX SUR DOCKER-COMPOSE.YAML

Pour configurer l'exécuteur local dans un fichier docker-compose.yaml, il suffit de :

- Remplacer : '**AIRFLOW_CORE_EXECUTOR: CeleryExecutor**' avec '**AIRFLOW_CORE_EXECUTOR: LocalExecutor**'

```
environment:  
  &airflow-common-env  
    AIRFLOW_CORE_EXECUTOR: LocalExecutor  
    AIRFLOW_DATABASE_SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow  
    # For backward compatibility, with Airflow <2.3  
    AIRFLOW_CORE_SQL_ALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow  
    AIRFLOW_CELERY_RESULT_BACKEND: db+postgresql://airflow:airflow@postgres/airflow  
    AIRFLOW_CELERY_BROKER_URL: redis://:@redis:6379/0
```



CONFIGURER UN DAG AVEC UN EXÉCUTEUR LOCAL ET POSTGRESQL

Pour configurer un DAG avec les exécuteurs locaux et PostgreSQL en utilisant Docker Compose, vous pouvez suivre les étapes suivantes :

- Remplacer : '**AIRFLOW__CORE__EXECUTOR: CeleryExecutor**' avec '**AIRFLOW__CORE__EXECUTOR: LocalExecutor**'

```
environment:  
    &airflow-common-env  
    AIRFLOW__CORE__EXECUTOR: LocalExecutor  
    AIRFLOW__DATABASE__SQLALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow  
    # For backward compatibility, with Airflow <2.3  
    AIRFLOW__CORE__SQLALCHEMY_CONN: postgresql+psycopg2://airflow:airflow@postgres/airflow  
    AIRFLOW__CELERY__RESULT_BACKEND: db+postgresql://airflow:airflow@postgres/airflow  
    AIRFLOW__CELERY__BROKER_URL: redis://:@redis:6379/0
```

- Créer les dossiers nécessaires : Vous devez créer les dossiers pour stocker les fichiers DAG et les fichiers de logs. Dans l'exemple ci-dessus, nous avons utilisé les dossiers ./dags et ./logs.

```
mkdir -p ./dags ./logs ./plugins
```



CONFIGURER UN DAG AVEC UN EXÉCUTEUR LOCAL ET POSTGRESQL

- Lancer les services : Pour lancer les services, exécutez la commande suivante :

```
docker-compose up -d
```

Cela lancera les services PostgreSQL et Airflow en arrière-plan. L'option -d signifie que les services seront exécutés en mode détaché.



CONFIGURER UN DAG AVEC UN EXÉCUTEUR LOCAL ET POSTGRESQL

Mise en place de la connexion PostgreSQL Airflow

Pour établir une connexion PostgreSQL Airflow, il faut tout d'abord créer la connexion Airflow pour se connecter à la base de données Postgres. Ensuite, dans l'onglet admin, choisir les connexions :

The screenshot shows the Airflow web interface with the 'Admin' menu open. The 'Connections' option is highlighted with a red box. The interface displays a list of DAGs (dataset_consumes_1 and dataset_consumes_1_and_2) and various configuration settings like owner, last run, and recent tasks.



CONFIGURER UN DAG AVEC UN EXÉCUTEUR LOCAL ET POSTGRESQL

Mise en place de la connexion PostgreSQL Airflow

Générer une nouvelle fenêtre pour passer les détails de la connexion Postgres. Ensuite, cliquer sur le bouton plus à côté de l'onglet action pour générer une connexion Airflow à Postgres

The screenshot shows the Airflow web interface with the 'DAGs' tab selected. The main content area is titled 'List Connection'. It features a search bar labeled 'Search' and a button with a '+' icon labeled 'Actions'. Below these, the text 'Record Count: 0' is displayed. At the bottom, the message 'No records found' is shown. The 'Actions' button is highlighted with a red box.



CONFIGURER UN DAG AVEC UN EXÉCUTEUR LOCAL ET POSTGRESQL

Mise en place de la connexion PostgreSQL Airflow

Fournir l'ID de connexion, le type de connexion (Postgres), l'hôte (localhost), le nom de schéma et les identifiants de connexion de Postgres. Le port par défaut à utiliser est 5432, ainsi que le mot de passe reçu précédemment.

The screenshot shows the 'Add Connection' dialog box in the Airflow web interface. The 'Connection Id' field is set to 'postgres'. The 'Connection Type' dropdown is set to 'Postgres', with a note below stating: 'Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package.' The 'Description' field is empty. The 'Host' field is set to 'postgres'. The 'Schema' field is empty. The 'Login' field is set to 'airflow'. The 'Password' field contains several dots ('.....'). The 'Port' field is set to '5432'. The 'Extra' field is empty. At the bottom, there are 'Save' and 'Test' buttons, along with a back arrow.



CONFIGURER UN DAG AVEC UN EXÉCUTEUR LOCAL ET POSTGRESQL

Mise en place de la connexion PostgreSQL Airflow

Fournir l'ID de connexion, le type de connexion (Postgres), l'hôte (localhost), le nom de schéma et les identifiants de connexion de Postgres. Le port par défaut à utiliser est 5432, ainsi que le mot de passe reçu précédemment.

Conn Id	Conn Type	Description	Host	Port	Is Encrypted	Is Extra Encrypted
postgres	postgres		postgres	5432	False	False



LES EXÉCUTEURS AVEC POSTGRESQL ET RABBITMQ

- L'utilisation de **PostgreSQL** comme base de données pour Airflow permet de stocker les **métadonnées** de tous les workflows, des tâches et des relations entre elles.
- Cela **facilite la planification, la gestion et la visualisation** des workflows dans Airflow.

PostgreSQL est également réputé pour sa fiabilité et sa robustesse, ce qui en fait un choix populaire pour les applications critiques.



100



LES EXÉCUTEURS AVEC POSTGRESQL ET RABBITMQ

- **RabbitMQ**, quant à lui, est un système de messagerie open-source qui permet de distribuer des messages entre des applications.
- Dans Airflow, **RabbitMQ** est utilisé comme système de messagerie pour la communication entre les différents processus d'exécution des tâches, notamment les workers et le scheduler.
- Cela permet **une gestion efficace des ressources, une meilleure répartition de charge et une évolutivité accrue.**





LES EXÉCUTEURS AVEC POSTGRESQL ET RABBITMQ

Le choix de l'exécuteur Airflow dépend **des besoins spécifiques** de votre environnement et de vos workflows.

Les exécuteurs les plus couramment utilisés avec PostgreSQL et RabbitMQ sont les suivants :

- **LocalExecutor** : cet exécuteur est utilisé pour exécuter les tâches en local sur le même nœud que le scheduler. Il est recommandé pour les environnements de développement et de test.

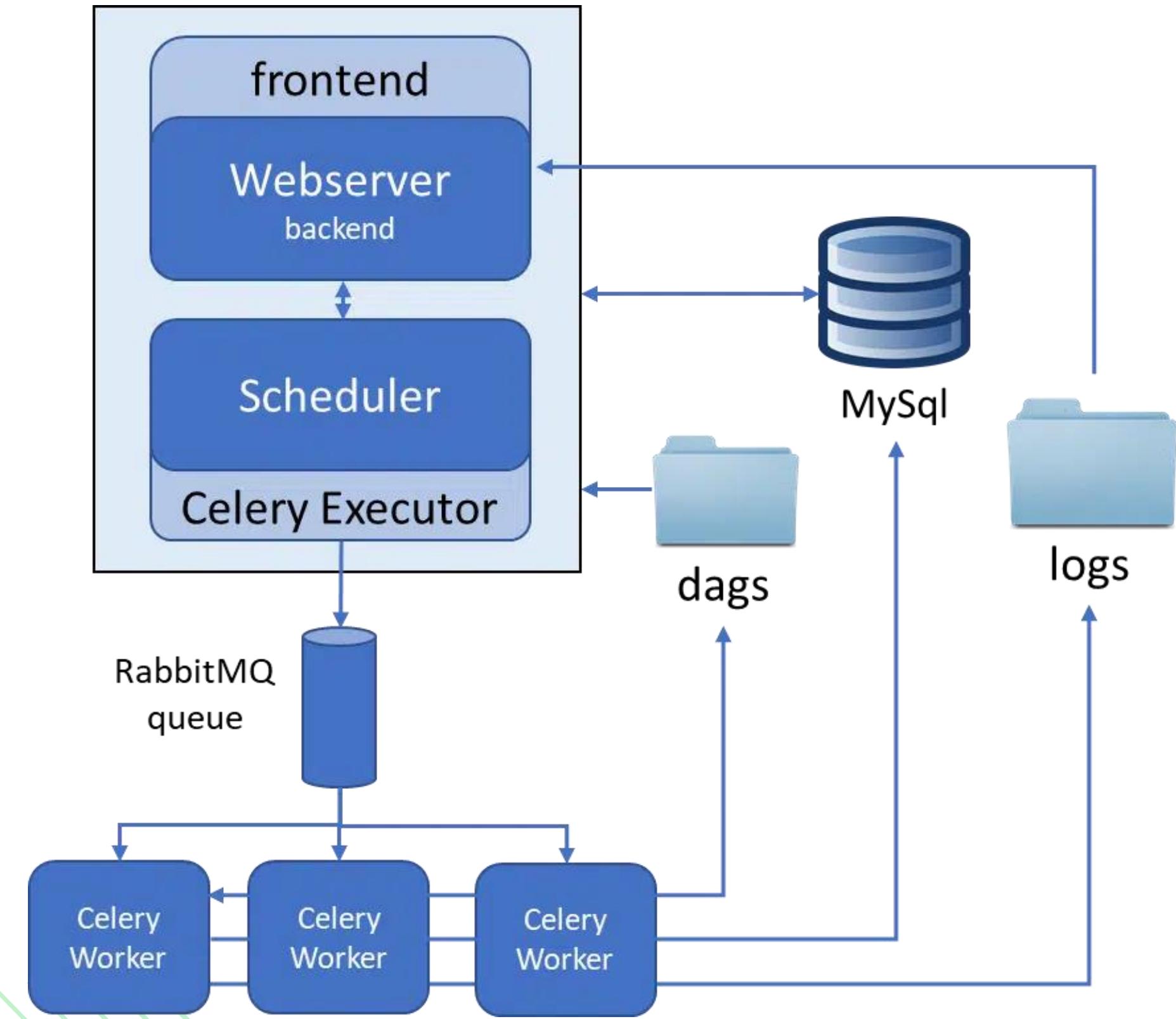


LES EXÉCUTEURS MULTI-NOEUDS

- **CeleryExecutor** : cet exécuteur utilise Celery comme moteur d'exécution pour distribuer les tâches sur plusieurs workers. Il prend en charge l'utilisation de RabbitMQ comme système de messagerie pour la communication entre le scheduler et les workers.



Exécuteur Celery



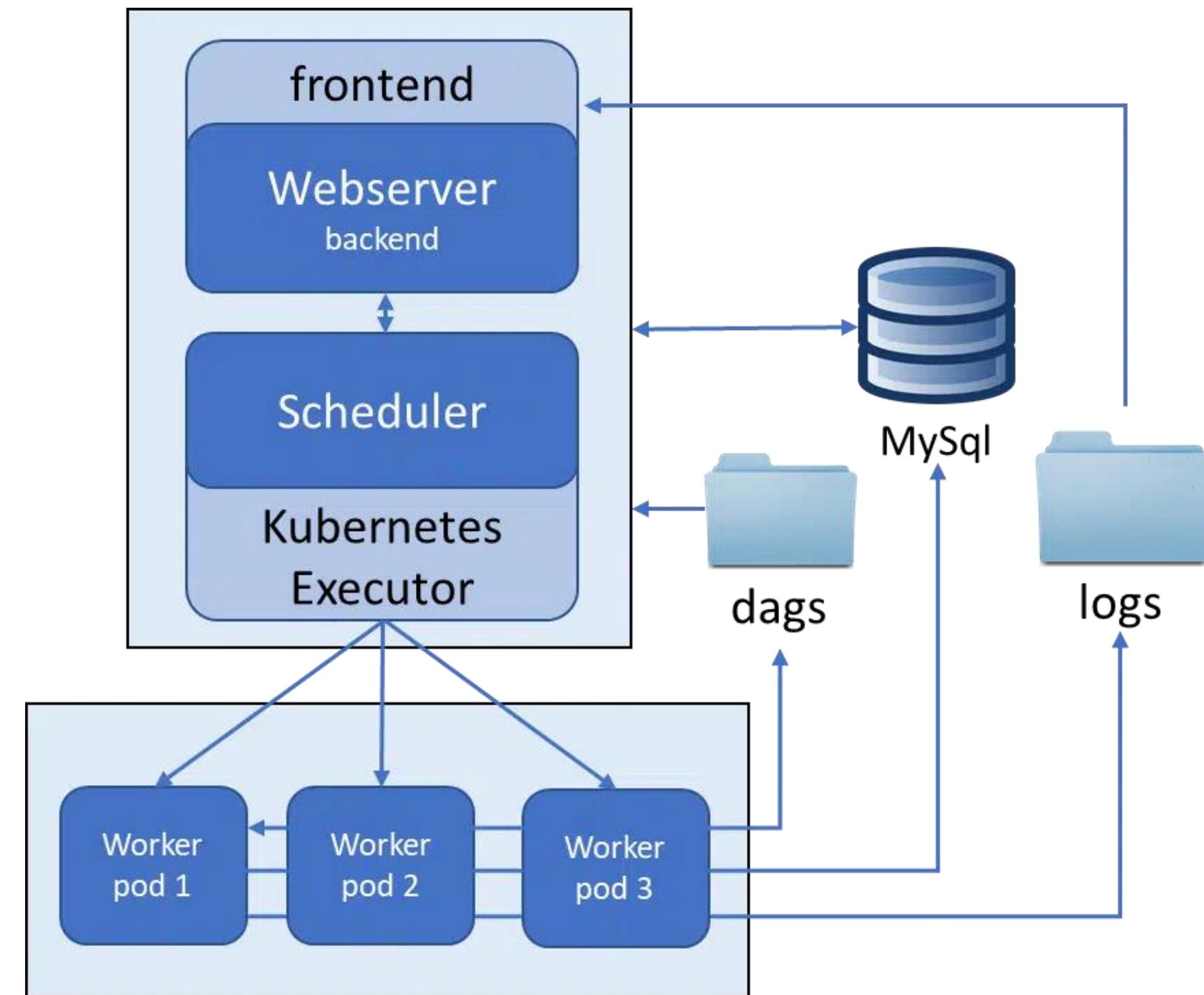


LES EXÉCUTEURS MULTI-Noeuds

- **KubernetesExecutor** : cet exécuteur utilise Kubernetes pour gérer les conteneurs des tâches d'Airflow. Il prend en charge l'utilisation de PostgreSQL et RabbitMQ pour la gestion des métadonnées et la communication entre les différents composants d'Airflow.



Exécuteur Kubernetes





Exéuteurs : Comparaison

Executor	Distributed	Ease of installation	Good fit
SequentialExecutor	No	Very easy	Demoing/testing
LocalExecutor	No	Easy	When running on a single machine is good enough
CeleryExecutor	Yes	Moderate	If you need to scale out over multiple machines
KubernetesExecutor	Yes	Complex	When you're familiar with Kubernetes and prefer a containerized setup

ATELIER



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

- Les files d'attente utilisées par Airflow sont similaires aux autres files d'attente et dépendent d'un système de messagerie, tels que **RabbitMQ** ou **ActiveMQ**.
- Le scheduler Airflow envoie des messages sous forme de tâches aux files d'attente, jouant ainsi le rôle d'éditeur.
- Les agents Airflow sont configurés pour surveiller les événements, c'est-à-dire les tâches, envoyés à des files d'attente spécifiques et les exécutent en conséquence.
- Cette approche permet une exécution distribuée des tâches et améliore l'efficacité du workflow.



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

- Voici les étapes pour configurer un DAG avec les exécuteurs Celery, PostgreSQL et RabbitMQ dans Airflow :
- Configurer Airflow pour utiliser **l'exécuteur Celery** dans le fichier **airflow.cfg** en modifiant la ligne suivante :

```
'executor' = 'CeleryExecutor'
```



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

- Configurer les connexions de base de données et de messagerie dans Airflow.
- Pour cela, accédez à **l'interface web Airflow** et cliquez sur "**Admin**" dans le menu de navigation, puis sélectionnez "**Connections**".
- Créez deux nouvelles connexions pour **RabbitMQ** et **PostgreSQL** en entrant les informations nécessaires.

Configurer un DAG avec Celery, PostgreSQL et RabbitMQ



The screenshot shows the Airflow web interface with the following details:

- Header:** Airflow, DAGs, Datasets, Security, Browse, Admin (with a red box around 'Connections'), Docs, 15:01 UTC, AA.
- DAGs Section:** All (47), Active (1), Paused (46).
 - dataset_consumes_1:** Owner: airflow, Last Run: On s3://dag1/output_1.txt, Recent Tasks: 0 of 2 datasets updated.
 - dataset_consumes_1_and_2:** Owner: airflow, Last Run: 0 of 2 datasets updated.
- Search Bar:** Search DAGs.
- Action Buttons:** Auto-refresh (on), Refresh.



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

Airflow DAGs Datasets Security ▾ Browse ▾ Admin ▾ Docs ▾ 12:52 UTC ▾ AA ▾

List Connection

Search ▾

Actions ▾ + Record Count: 1

Conn Id	Conn Type	Description	Host	Port	Is Encrypted	Is Extra Encrypted
postgres	postgres		postgres	5432	False	False



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

Configurez les champs suivants :

Conn Id : Identifiant pour référencer cette connexion. La valeur par défaut est rabbitmq_default.

login : Identifiant de connexion pour le serveur RabbitMQ.

password : Mot de passe pour le serveur RabbitMQ.

port : Port pour le serveur RabbitMQ, généralement 5672.

host : Hôte du serveur RabbitMQ.

vhost : L'hôte virtuel auquel vous souhaitez vous connecter.



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Modifier la ligne 58 du fichier docker-compose.yaml par :

```
AIRFLOW__CELERY__BROKER_URL: "amqp://defaultuser:defaultpassword@rabbitmq/"
```

l'URL du courtier de messages que Airflow utilise pour communiquer avec Celery, qui est un système de file d'attente de tâches distribué.

L'URL est au format "amqp://nom_utilisateur:mot_de_passe@nom_hôte/"



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Définir la condition de santé pour le service RabbitMQ

```
rabbitmq:  
    condition: service_healthy
```

La condition `service_healthy` est une fonctionnalité de Docker qui vérifie si le service RabbitMQ est prêt à recevoir des connexions avant de le considérer comme "healthy" (en bonne santé).



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Ajouter les paramètres du service rabbitmq

- **image: rabbitmq:3-management** : Cela définit l'image Docker utilisée pour le service RabbitMQ.
- **hostname: my-rabbit** : Cela définit le nom d'hôte du conteneur RabbitMQ.
- **ports: [...]** : Cela définit les ports à exposer pour RabbitMQ, qui sont 8672:15672 et 5672:5672 dans ce cas.
- **volumes: [...]** : Cela définit les volumes à monter pour RabbitMQ, qui sont rabbitmq_data:/var/lib/rabbitmq
- **environment: [...]** : Cela définit les variables d'environnement à passer au conteneur RabbitMQ
- **healthcheck: [...]** : Cela définit le test de santé à effectuer pour RabbitMQ
- **test** : Consiste à envoyer une requête de ping à RabbitMQ

```
rabbitmq:
  image: rabbitmq:3-management
  # container_name: some-rabbit
  hostname: my-rabbit
  ports:
    - '8672:15672'
    - '5672:5672'
  volumes:
    - 'rabbitmq_data:/var/lib/rabbitmq'
  environment:
    # ALLOW_EMPTY_PASSWORD is recommended only for development.
    # - ALLOW_EMPTY_PASSWORD=yes
    - RABBITMQ_DEFAULT_USER=defaultuser
    - RABBITMQ_DEFAULT_PASS=defaultpassword
  healthcheck:
    # test: ["CMD", "rabbitmq-diagnostics -q ping"]
    test:
      rabbitmq-diagnostics -q ping
  interval: 15s
  timeout: 5s
  retries: 6
```



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Ajouter la déclaration de volume dans le fichier docker-compose.yml pour le service RabbitMQ. (ligne 292):

```
rabbitmq_data:
```

Cela permet de stocker les données de RabbitMQ en dehors du conteneur, dans un volume qui peut être partagé entre plusieurs conteneurs ou entre le conteneur RabbitMQ et l'hôte Docker.



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Démarrer le conteneur Docker correspondant au service "airflow-init"

```
docker compose up airflow-init
```



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Interface Rabbitmq (port 8672) :

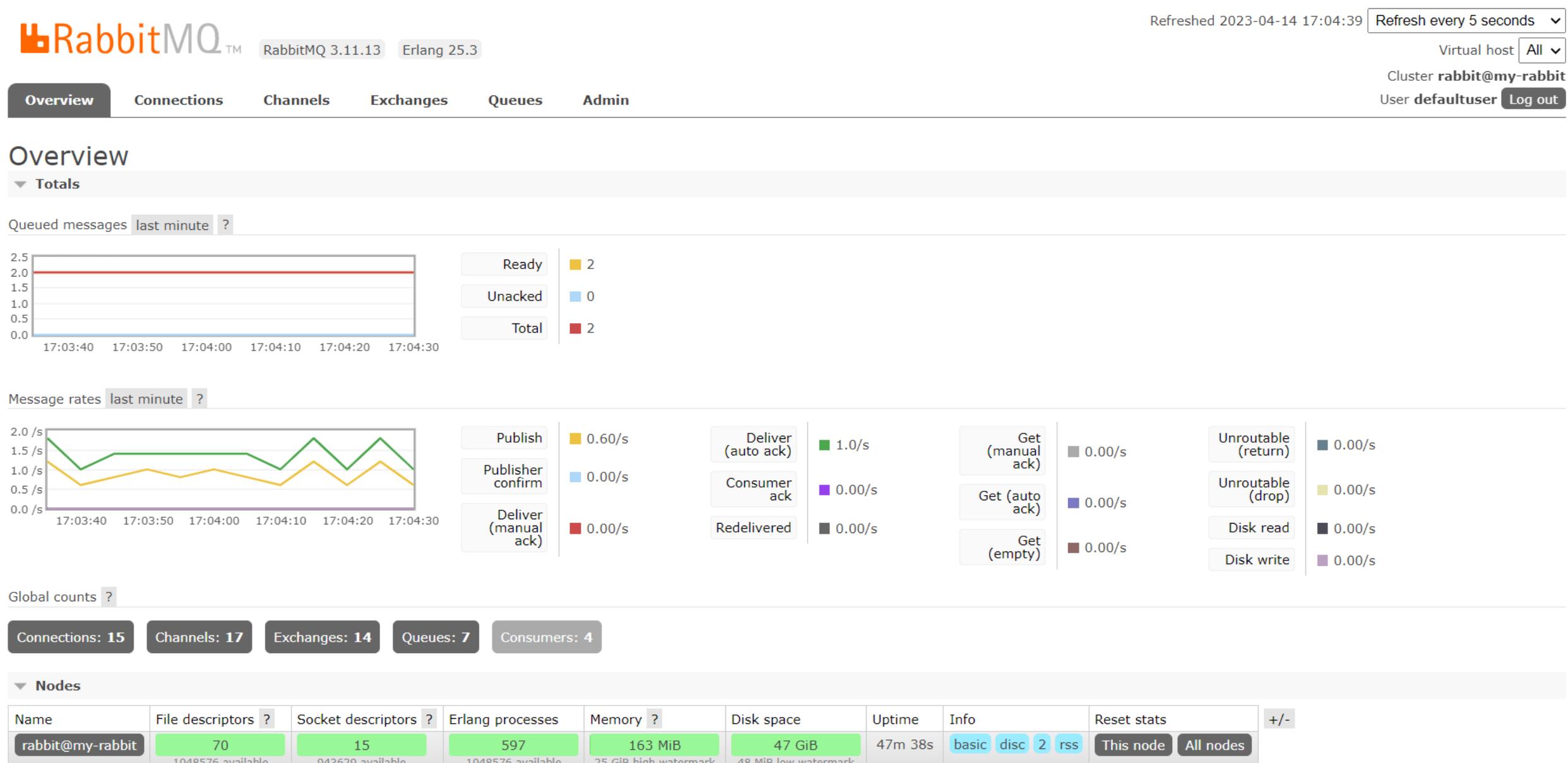




Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Interface Rabbitmq (port 8672) :



Configurer un DAG avec Celery, PostgreSQL et RabbitMQ

CONFIGURATION RABBITMQ SUR DOCKER-COMPOSE.YAML

Interface Flower

The screenshot shows the Flower web interface for monitoring Celery workers. At the top, there's a navigation bar with tabs: Flower (selected), Dashboard, Tasks, Broker, Docs, and Code. Below the navigation is a row of summary metrics: Active: 0, Processed: 0, Failed: 0, Succeeded: 0, and Retried: 0. The main content area contains a table of worker statistics. The table has columns for Worker Name, Status, Active tasks, Processed tasks, Failed tasks, Succeeded tasks, Retried tasks, and Load Average. One worker entry is visible: celery@e1798507d8f5, Status: Online, Active: 0, Processed: 0, Failed: 0, Succeeded: 0, Retried: 0, Load Average: 14.82, 17.27, 15.24. A search bar is located at the top right of the table area. At the bottom left of the table area, it says "Showing 1 to 1 of 1 entries".

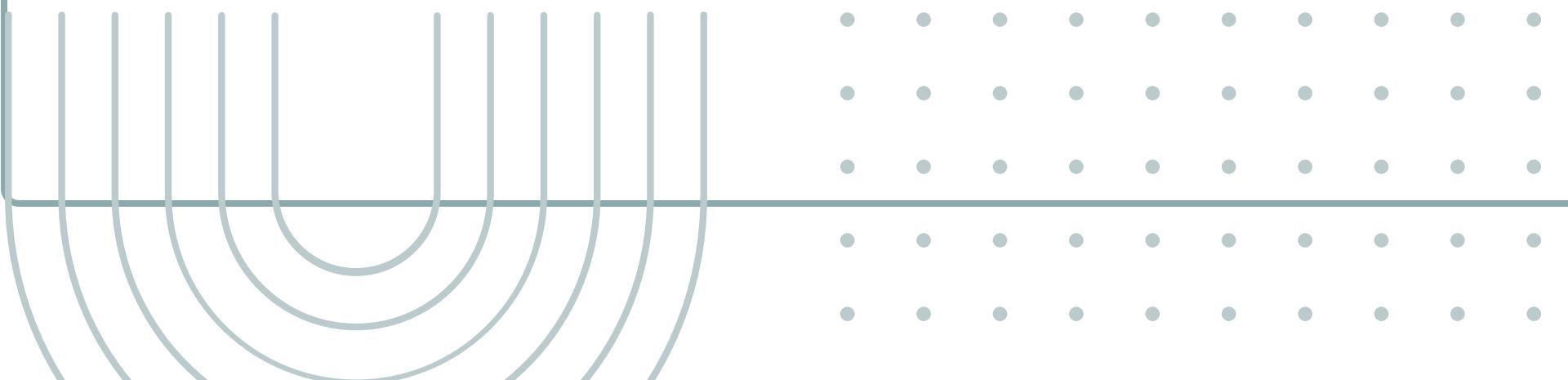
Worker Name	Status	Active	Processed	Failed	Succeeded	Retried	Load Average
celery@e1798507d8f5	Online	0	0	0	0	0	14.82, 17.27, 15.24



ATELIER

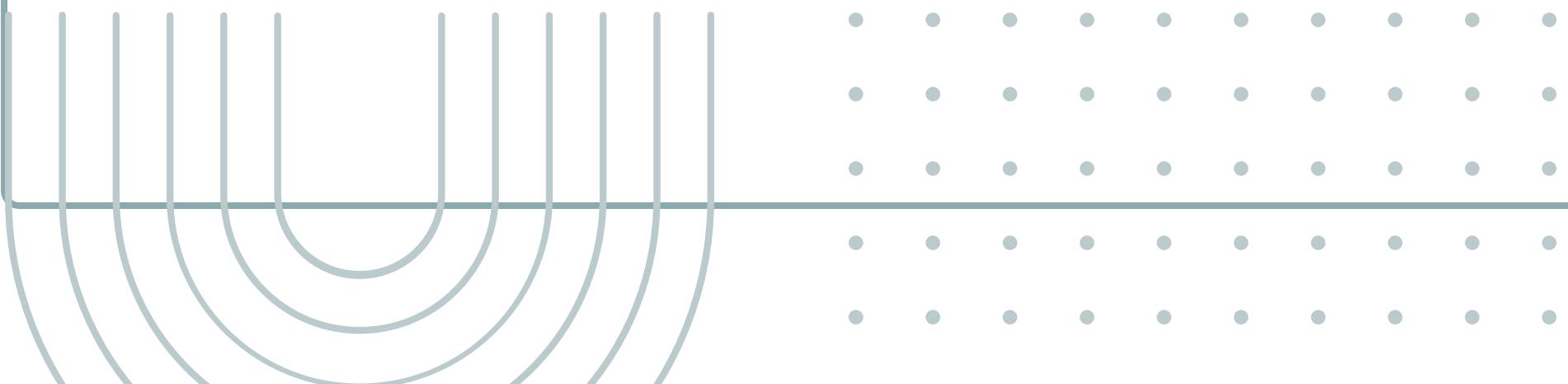
07.

GESTION DES WORKFLOWS



7.1

OPERATORS





LES OPERATEURS : SENSOROPERATOR, ACTIONOPERATOR ET TRANSFEROPERATOR

- Représente une tâche unique dans un flux de travail.
- Fonctionne de manière indépendante (généralement).
- Ne partage généralement pas d'informations.
- Différents opérateurs pour effectuer différentes tâches.

```
DummyOperator(task_id= 'example' , dag=dag)
```



LES OPERATEURS : SENSOROPERATOR, ACTIONOPERATOR ET TRANSFEROOPERATOR

On distingue trois catégories principales d'opérateurs :

- **SensorOperator**

Les capteurs sont très utiles car ils vous permettent d'attendre que quelque chose se passe avant de passer à la tâche suivante. Par exemple, si vous attendez des fichiers, vous utiliserez le FileSensor.

- **ActionOperator**

Un opérateur d'action exécute quelque chose Par exemple, le PythonOperator exécute une fonction Python le BashOperator exécute une commande bash.

- **TransferOperator**

Permettent essentiellement de transférer des données d'un point A à un point B . Par exemple, vous voulez transférer des données de MySQL à Redshift .



LES OPERATEURS : SENSOROPERATOR, ACTIONOPERATOR ET TRANSFEROOPERATOR

Le suivant sont quelques-uns des opérateurs Airflow les plus fréquemment utilisés :

- **PythonOperator**: Exécute une fonction Python.
- **BashOperator**: Exécute un script bash.
- **KubernetesPodOperator**: Exécute une tâche définie comme une image Docker dans un Pod Kubernetes.
- **SnowflakeOperator**: Exécute une requête sur une base de données Snowflake.
- **DeferrableOperators**: Libère leur emplacement de travailleur tout en attendant que leur travail soit terminé.



LES OPERATEURS : SENSOROPERATOR, ACTIONOPERATOR ET TRANSFEROOPERATOR

• PythonOperator

- Exécute une fonction / callable Python
- Fonctionne de manière similaire à l'opérateur Bash, avec plus d'options
- Peut transmettre des arguments au code Python.

```
from airflow.operators.python_operator import PythonOperator
def printme():
    print("This goes in the logs!")
python_task = PythonOperator( task_id= 'simple_print' ,
                             python_callable=printme,
                             dag=example_dag )
```



LES OPERATEURS : SENSOROPERATOR, ACTIONOPERATOR ET TRANSFEROOPERATOR

• BashOperator

- Exécute une commande ou un script Bash donné.
- Exécute la commande dans un répertoire temporaire.
- Peut spécifier des variables d'environnement pour la commande.

```
from airflow.operators.bash_operator import BashOperator

example_task = BashOperator(task_id= 'bash_ex' ,
                            bash_command= 'echo 1',
                            dag=dag)
```



LES OPERATEURS : SENSOROPERATOR, ACTIONOPERATOR ET TRANSFEROOPERATOR

• KubernetesPodOperator

- Exécuter une tâche définie en tant qu'image Docker dans un Pod Kubernetes
- Peut spécifier des conteneurs d'initiation, des conteneurs de côtés et des volumes
- Permet de personnaliser les politiques de redémarrage et de backoff
- Il est utile pour les tâches nécessitant une grande isolation ou pour celles qui nécessitent des ressources spécifiques à Kubernetes.

```
k8s_task = KubernetesPodOperator(  
    task_id="k8s_task",  
    name="k8s_task",  
    namespace="default",  
    image="docker/airflow:latest",  
    cmd=["bash", "-c"],  
    arguments=["echo 'Hello, World!'"],  
    labels={"app": "airflow", "framework": "kubernetes"},  
    image_pull_policy="Always",  
    is_delete_operator_pod=True,  
    hostnetwork=False,)
```



LES OPERATEURS : SENSOROPERATOR, ACTIONOPERATOR ET TRANSFEROOPERATOR

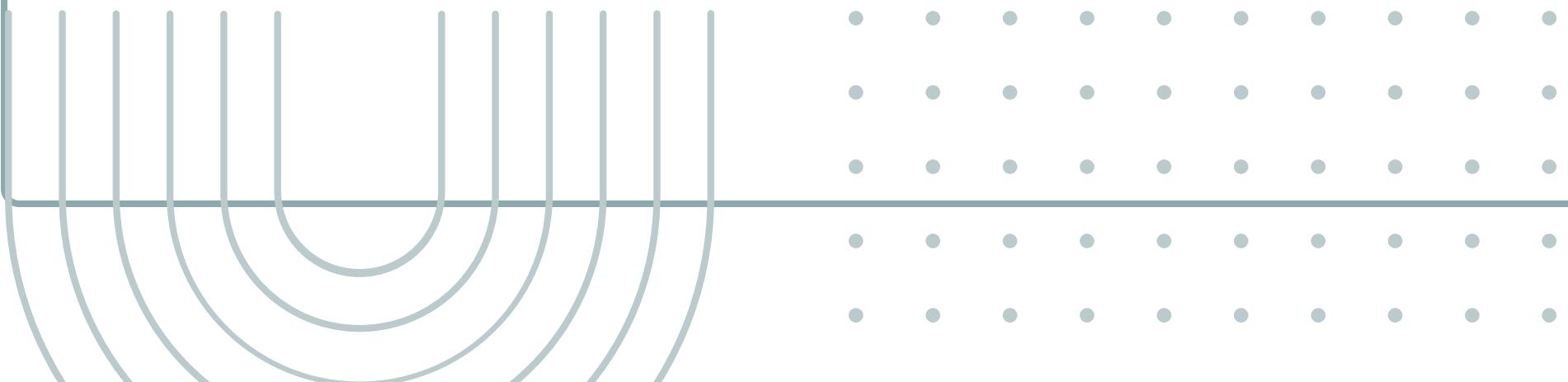
• SnowflakeOperator

- Exécuter des requêtes SQL sur une base de données Snowflake
- Les identifiants de connexion, la requête SQL à exécuter et le chemin de destination pour les résultats de la requête peuvent être spécifiés par les utilisateurs.
- Les résultats de la requête peuvent être stockés dans un fichier ou transmis à un autre opérateur Airflow pour une utilisation ultérieure.
- Il est utile pour récupérer des données stockées dans une base de données Snowflake et les utiliser dans le cadre d'un flux de travail Airflow.

```
task1 = SnowflakeOperator(  
    task_id='snowflake_query',  
    snowflake_conn_id='snowflake_connection',  
    sql='SELECT * FROM my_table WHERE my_column > 10',  
    warehouse='my_warehouse',  
    database='my_database',  
    schema='my_schema',  
    destination_table='my_results_table',  
    autocommit=True )
```

7.2

DATA AWARE SCHEDULING





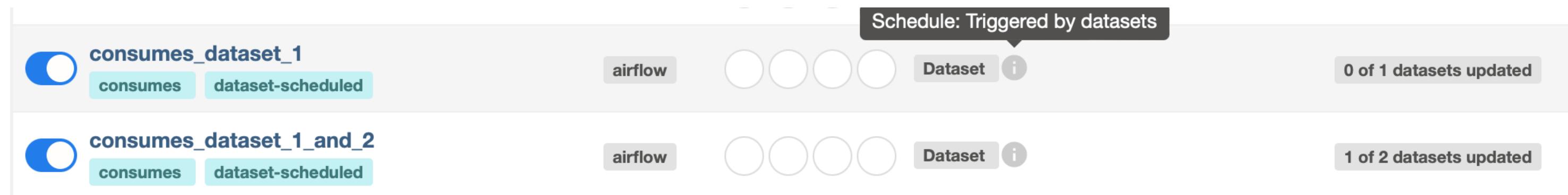
DATA-AWARE SCHEDULING

En plus de la planification des DAG en fonction du temps, ils peuvent également être planifiés en fonction de la mise à jour d'un ensemble de données par une tâche.

```
from airflow import Dataset

with DAG(...):
    MyOperator(
        # cette tâche met à jour example.csv
        outlets=[Dataset("s3://dataset-bucket/example.csv")], ...
    )
    with DAG(
        # ce DAG doit être exécuté lorsque example.csv est mis à jour (par dag1)
        schedule=[Dataset("s3://dataset-bucket/example.csv")],
        ...
    ):
        ...

    ...
```





DATA-AWARE SCHEDULING

Qu'est-ce qu'un "ensemble de données - dataset" ?

- Un ensemble de données Airflow est un substitut pour un regroupement logique de données.
- Les ensembles de données peuvent être mis à jour par des tâches amont "productrices", et les mises à jour d'ensembles de données contribuent à la planification des DAGs "consommateurs" en aval.

Un ensemble de données est défini par un identificateur de ressource uniforme (URI) :

```
from airflow import Dataset  
  
example_dataset = Dataset("s3://dataset-bucket/example.csv")
```



DATA-AWARE SCHEDULING

Qu'est-ce qu'un "ensemble de données - dataset" ?

Il y a deux restrictions sur l'URI de l'ensemble de données :

- Elle doit être un URI valide, ce qui signifie qu'elle doit être composée uniquement de caractères ASCII.
- Le schéma URI ne peut pas être "airflow" (cela est réservé pour une utilisation future).

Si l'un des exemples ci-dessous est utilisé, une erreur de ValueError sera levée par votre code, et Airflow ne l'importera pas.

```
# datasets non valide:  
reserved = Dataset("airflow://example_dataset")  
not_ascii = Dataset("èxample_datašet")
```

L'identificateur n'a pas besoin d'être un URI absolue, il peut s'agir d'un URI relative sans schéma, voire simplement d'un chemin ou d'une chaîne de caractères :

```
# datasets valide:  
schemeless = Dataset("//example/dataset")  
csv_file = Dataset("example_dataset")
```



DATA-AWARE SCHEDULING

Comment utiliser les datasets dans les DAGs ?

Les ensembles de données peuvent être utilisés pour spécifier les dépendances de données dans les DAG. Par exemple :

```
example_dataset = Dataset("s3://dataset/example.csv")

with DAG(dag_id="producer", ...):
    BashOperator(task_id="producer", outlets=[example_dataset], ...)

with DAG(dag_id="consumer", schedule=[example_dataset], ...):
    ...
```

Une fois que la tâche producteur dans le DAG producteur a été exécutée avec succès, Airflow planifie l'exécution du DAG consommateur.

Un dataset ne sera marqué comme mis à jour que si la tâche se termine avec succès - si la tâche échoue ou si elle est ignorée, aucune mise à jour n'aura lieu et le DAG consommateur ne sera pas planifié.



DATA-AWARE SCHEDULING

Multiples datasets ?

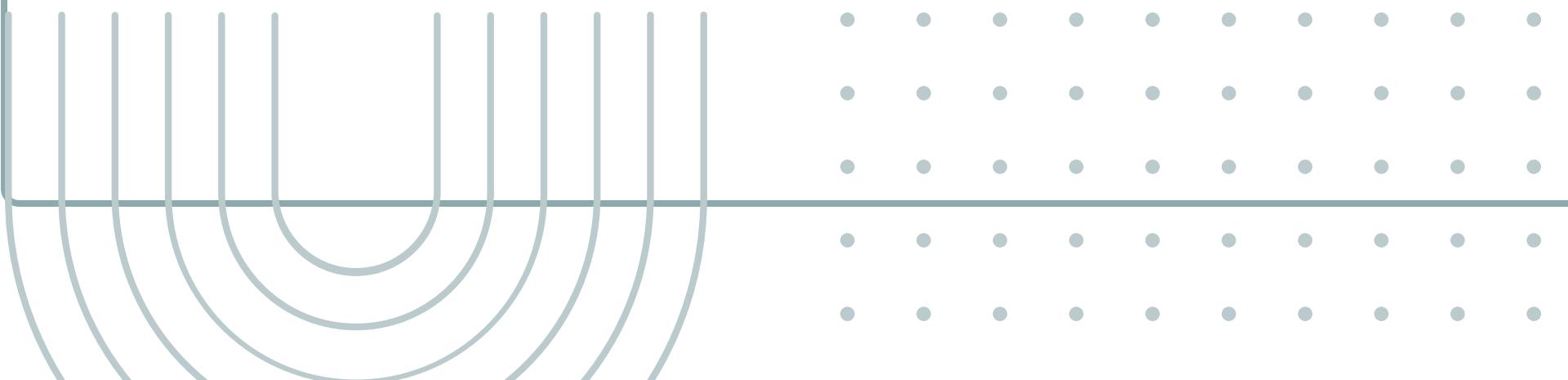
Comme le paramètre de planification est une liste, les DAG peuvent nécessiter plusieurs ensembles de données et le DAG sera planifié une fois que tous les ensembles de données qu'il consomme ont été mis à jour au moins une fois depuis la dernière exécution :

```
with DAG(  
    dag_id="multiple_datasets_example",  
    schedule=[  
        example_dataset_1,  
        example_dataset_2,  
        example_dataset_3,  
    ],  
    ...  
):  
    ...
```

Si un ensemble de données est mis à jour plusieurs fois avant que tous les ensembles de données consommés ne soient mis à jour, le DAG aval sera quand même exécuté une seule fois. C'est parce qu'Airflow suit les mises à jour d'ensemble de données au niveau du DAG, et non au niveau de chaque tâche individuelle.

7.3

TASK FACTORY





Task Factory

- Dans certains cas, vous pouvez vous retrouver à écrire des variations du même DAG à plusieurs reprises.
- Cela se produit souvent dans des situations où vous ingérez des données à partir de sources de données connexes, avec seulement de légères variations dans les chemins sources et les transformations appliquées aux données.
- De même, vous pouvez avoir des processus qui nécessitent de nombreuses étapes / transformations similaires et qui sont donc répétés dans de nombreux DAG différents.
- Un moyen efficace d'accélérer le processus de génération de ces structures DAG courantes est d'écrire une **fonction task factory**.
- L'idée derrière une telle fonction est qu'elle prend toute configuration requise pour les étapes respectives et génère le DAG ou l'ensemble de tâches correspondant (le produisant ainsi, comme une usine).



DYNAMIC TASK MAPPING

- Par exemple, si nous avons un processus courant qui implique de récupérer des données à partir d'une API externe et de prétraiter ces données à l'aide d'un script donné, nous pourrions écrire une fonction factory, comme ceci :

```

File paths used by the different tasks
Parameters that configure the tasks that will be created by the factory function

def generate_tasks(dataset_name, raw_dir, processed_dir,
                   preprocess_script, output_dir, dag):
    raw_path = os.path.join(raw_dir, dataset_name, "{ds_nodash}.json")
    processed_path = os.path.join(
        processed_dir, dataset_name, "{ds_nodash}.json"
    )
    output_path = os.path.join(output_dir, dataset_name, "{ds_nodash}.json")

    fetch_task = BashOperator(
        task_id=f"fetch_{dataset_name}",
        bash_command=f"curl http://example.com/{dataset_name}.json
                      > {raw_path}.json",
        dag=dag,
    )

    preprocess_task = BashOperator(
        task_id=f"preprocess_{dataset_name}",
        bash_command=f"echo '{preprocess_script}' {raw_path}
                      > {processed_path}",
        dag=dag,
    )

    export_task = BashOperator(
        task_id=f"export_{dataset_name}",
        bash_command=f"echo 'cp {processed_path} {output_path}'",
        dag=dag,
    )

    fetch_task >> preprocess_task >> export_task

    return fetch_task, export_task
  
```

Creating the individual tasks

Defining task dependencies

Return the first and last tasks in the chain so that we can connect them to other tasks in the larger graph (if needed).



Task Factory

```
import airflow.utils.dates
from airflow import DAG

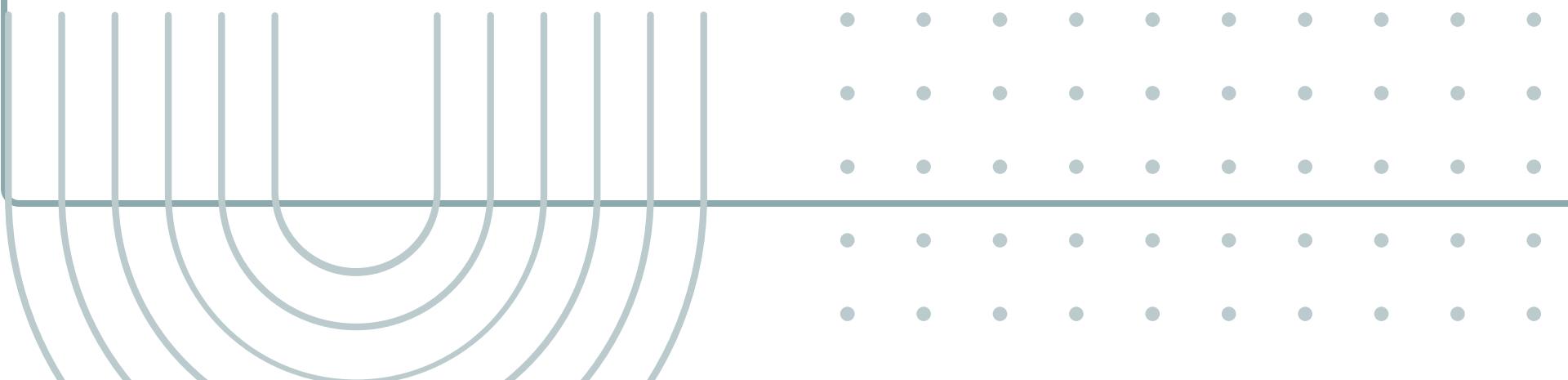
with DAG(
    dag_id="01_task_factory",
    start_date=airflow.utils.dates.days_ago(5),
    schedule_interval="@daily",
) as dag:
    for dataset in ["sales", "customers"]:
        generate_tasks(
            dataset_name=dataset,
            raw_dir="/data/raw",
            processed_dir="/data/processed",
            output_dir="/data/output",
            preprocess_script=f"preprocess_{dataset}.py",
            dag=dag,
)
```

Creating sets of tasks with different configuration values

Passing the DAG instance to connect the tasks to the DAG

7.4

DYNAMIC TASK MAPPING





Dynamic Task Mapping

- La "**Dynamic Task Mapping**" permet de créer des tâches à l'exécution d'un flux de travail en fonction des données actuelles, plutôt que de savoir à l'avance combien de tâches seraient nécessaires par l'auteur du DAG.
- Cela ressemble à la définition de vos tâches dans une boucle for, mais au lieu d'avoir le fichier DAG récupérer les données et le faire lui-même, le planificateur peut le faire en fonction de la sortie d'une tâche précédente.
- Juste avant l'exécution d'une tâche mappée, le planificateur créera **n copies** de la tâche, une pour chaque entrée.
- Il est également possible d'avoir une tâche qui opère sur la sortie collectée d'une tâche mappée, communément appelée "map and reduce".



DYNAMIC TASK MAPPING

Simple Mapping

Dans sa forme la plus simple, vous pouvez cartographier sur une liste définie directement dans votre fichier DAG en utilisant la fonction "**expand()**" au lieu d'appeler directement votre tâche.

```
""" Exemple de DAG démontrant l'utilisation de la "Dynamic Task Mapping". """
from __future__ import annotations

from datetime import datetime

from airflow import DAG
from airflow.decorators import task

with DAG(dag_id="example_dynamic_task_mapping", start_date=datetime(2022, 3, 4)) as dag:

    @task
    def add_one(x: int):
        return x + 1

    @task
    def sum_it(values):
        total = sum(values)
        print(f" Total était de {total}")

    added_values = add_one.expand(x=[1, 2, 3])
    sum_it(added_values)
```

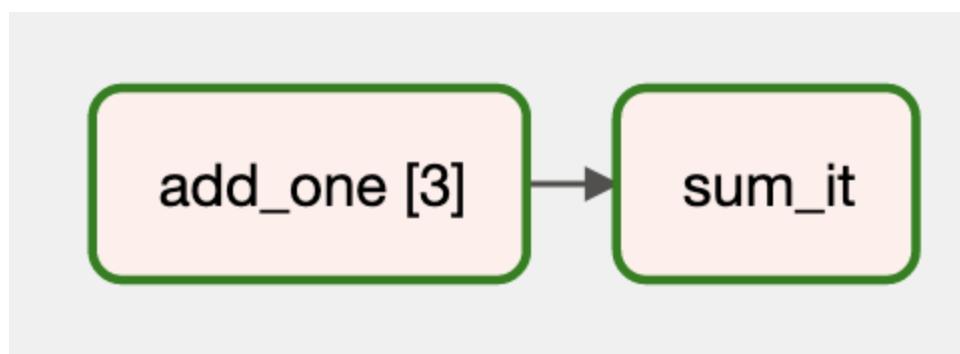


DYNAMIC TASK MAPPING

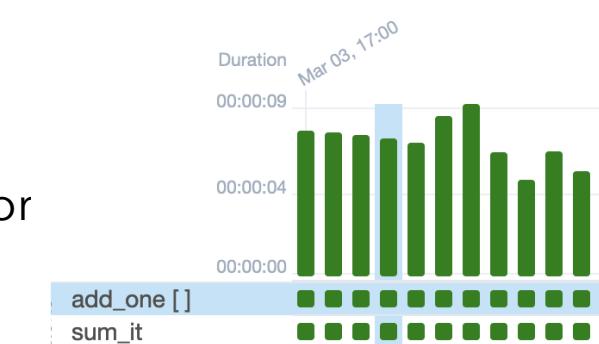
Simple Mapping

Cela affichera "**Total était de 9**" dans les journaux de tâches lors de l'exécution.

Voici la structure du DAG résultant:



La vue en grille fournit également une visibilité sur les tâches cartographiées dans le panneau de détails.



Auto-refresh
[Hide Details Panel](#)

DAG **Run** **simple_mapping / 2022-03-07, 17:00:00 MST / Task add_one**

[All Instances](#) [Filter Upstream](#)

[Ignore All_deps](#) [Ignore Task State](#) [Ignore Task Deps](#)

[Past](#) [Future](#) [Upstream](#) [Downstream](#) [Recursive](#) [Failed](#)

[Past](#) [Future](#) [Upstream](#) [Downstream](#)

[Past](#) [Future](#) [Upstream](#) [Downstream](#)

Status: ■ success Started: 2022-04-08, 17:29:05 MDT
Ended: 2022-04-08, 17:29:06 MDT

3 Tasks Mapped
success: 3

Task Id: add_one
Run Id: scheduled__2022-03-07T00:00:00+00:00
Operator: _PythonDecoratedOperator
Duration: 00:00:01

Mapped Instances

MAP INDEX	STATE	DURATION	START DATE	END DATE	Actions
0	■ success	00:00:00	2022-04-08, 17:29:05 MDT	2022-04-08, 17:29:06 MDT	
1	■ success	00:00:00	2022-04-08, 17:29:05 MDT	2022-04-08, 17:29:06 MDT	
2	■ success	00:00:00	2022-04-08, 17:29:05 MDT	2022-04-08, 17:29:06 MDT	

< > 1-3 of 3



DYNAMIC TASK MAPPING

Repeated Mapping

Le résultat d'une tâche cartographiée peut également être utilisé en entrée de la tâche cartographiée suivante.

```
with DAG(dag_id="repeated_mapping", start_date=datetime(2022, 3, 4)) as dag:  
  
    @task  
    def add_one(x: int):  
        return x + 1  
  
    first = add_one.expand(x=[1, 2, 3])  
    second = add_one.expand(x=first)
```

Cela donnerait comme résultat **[3, 4, 5]**.



DYNAMIC TASK MAPPING

Constant parameters

En plus de transmettre des arguments qui sont développés à l'exécution, il est possible de transmettre des arguments qui ne changent pas - afin de les différencier clairement, nous utilisons des fonctions différentes : "expand()" pour les arguments cartographiés et "partial()" pour ceux qui ne le sont pas.

```
@task
def add(x: int, y: int):
    return x + y

added_values = add.partial(y=10).expand(x=[1, 2, 3])
# Cela a pour résultat que la fonction "add" est étendue à
# add(x=1, y=10)
# add(x=2, y=10)
# add(x=3, y=10)
```

Cela donnerait comme résultat des valeurs de 11, 12 et 13.

Cela est également utile pour transmettre des éléments tels que des identifiants de connexion, des noms de tables de base de données ou des noms de compartiments à des tâches.



DYNAMIC TASK MAPPING

Mapping sur plusieurs paramètres

En plus d'un seul paramètre, il est possible de passer plusieurs paramètres à "expand". Cela aura pour effet de créer un "produit cartésien", en appelant la tâche cartographiée avec chaque combinaison de paramètres.

```
@task
def add(x: int, y: int):
    return x + y

added_values = add.expand(x=[2, 4, 8], y=[5, 10])
# Cela entraîne l'appel de la fonction add avec
# add(x=2, y=5)
# add(x=2, y=10)
# add(x=4, y=5)
# add(x=4, y=10)
# add(x=8, y=5)
# add(x=8, y=10)
```

Cela donnerait comme résultat l'appel de la tâche "add" 6 fois. Veuillez noter, cependant, que l'ordre d'expansion n'est pas garanti.



DYNAMIC TASK MAPPING

Mapping Générée par tâche

Jusqu'à présent, les exemples présentés auraient pu être implémentés avec une boucle "for" dans le fichier DAG, mais la véritable puissance de la cartographie dynamique des tâches réside dans la capacité d'une tâche à générer la liste à parcourir.

```
@task
def make_list():
    # Cela peut également provenir d'un appel API, de la vérification d'une base de données, ou presque tout ce que vous voulez, tant que
    # la liste/dictionnaire résultant peut être stocké dans le backend XCom actuel.
    return [1, 2, {"a": "b"}, "str"]

@task
def consumer(arg):
    print(arg)

with DAG(dag_id="dynamic-map", start_date=datetime(2022, 4, 2)) as dag:
    consumer.expand(arg=make_list())
```

La tâche "**make_list**" s'exécute comme une tâche normale et doit retourner une liste ou un dictionnaire, puis la tâche **consommatrice** sera appelée quatre fois, une fois avec chaque valeur dans le retour de "**make_list**".



DEFERABLE OPERATORS

- **Termes et concepts**

- **asyncio:** Une bibliothèque Python utilisée comme base pour plusieurs frameworks asynchrones. Cette bibliothèque est essentielle à la fonctionnalité des opérateurs reportables et est utilisée lors de l'écriture des déclencheurs.
- **Triggers(Déclencheurs):** De petites sections asynchrones de code Python. En raison de leur nature asynchrone, ils coexistent efficacement dans un seul processus connu sous le nom de déclencheur.
- **Déclencheur:** Un service Airflow similaire à un planificateur ou un worker qui exécute une boucle d'événements asyncio dans votre environnement Airflow. L'exécution d'un déclencheur est essentielle pour utiliser les opérateurs reportables.
- **Deferred (Reporté):** Un état de tâche Airflow indiquant qu'une tâche a mis en pause son exécution, libéré l'emplacement du worker et soumis un déclencheur à être récupéré par le processus de déclencheur.

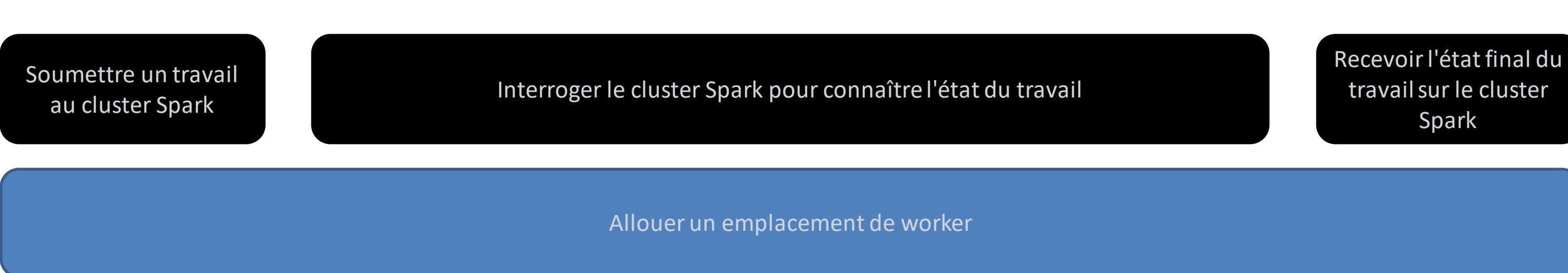
Les termes deferrable, async et asynchrone sont utilisés de manière interchangeable et ont la même signification.



DEFERABLE OPERATORS

- Avec **des opérateurs traditionnels**, une tâche soumet un travail à un système externe tel qu'un cluster Spark, puis interroge l'état du travail jusqu'à ce qu'il soit terminé.
- Bien que la tâche ne réalise pas un travail significatif, elle occupe toujours un emplacement de worker pendant le processus de sondage.
- À mesure que les emplacements de worker sont occupés, les tâches sont mises en file d'attente et les heures de début sont retardées.

Le schéma suivant illustre ce processus :





DEFERABLE OPERATORS

- Avec **Deferrable Operators**, les emplacements de worker sont libérés lorsque la tâche interroge l'état du travail.
- Lorsque la tâche est reportée, le processus de sondage est transféré en tant que déclencheur au déclencheur, et l'emplacement du worker devient disponible.
- Le déclencheur peut exécuter de nombreuses tâches de sondage asynchrones simultanément, ce qui empêche les tâches de sondage d'occuper vos ressources de worker. Lorsque l'état final du travail est reçu, la tâche reprend, occupant un emplacement de worker pendant qu'elle se termine.

Le schéma suivant illustre ce processus :





DEFFERABLE OPERATORS

Les DeferrableOperators disponibles :

Les deferrable operators suivants sont installés par défaut dans Airflow :

- TimeSensorAsync
- DateTimeSensorAsync

```
async_sensor = DateTimeSensorAsync(  
    task_id="async_task",  
    target_time="""{{ macros.datetime.utcnow() +  
    macros.timedelta(minutes=20) }}""",  
)
```



BEST PRACTICES

Créer un nouveau DAG est un processus en trois étapes :

- Écrire du code Python pour créer un objet DAG,
- Tester si le code correspond à nos attentes,
- Configurer les dépendances d'environnement pour exécuter votre DAG.

Écrire un DAG

Créer un nouveau DAG dans Airflow est assez simple. Cependant, il y a de nombreuses choses auxquelles vous devez faire attention pour vous assurer que l'exécution ou l'échec du DAG ne produit pas de résultats inattendus.



BEST PRACTICES

Créer une tâche

Vous devez considérer les tâches dans Airflow comme des transactions dans une base de données. Cela implique que vous ne devriez jamais produire de résultats incomplets à partir de vos tâches.

Airflow peut réessayer une tâche en cas d'échec. Ainsi, les tâches doivent produire le même résultat à chaque nouvelle exécution. Voici quelques moyens d'éviter de produire un résultat différent :

- **N'utilisez pas INSERT** lors d'une nouvelle exécution de la tâche, car cela pourrait entraîner **des doublons** dans votre base de données.

Remplacez-le par UPSERT.

- **Lisez et écrivez dans une partition spécifique.** Ne lisez jamais les données les plus récentes disponibles dans une tâche. Une meilleure façon de procéder est de **lire les données d'entrée à partir d'une partition spécifique**.
- La fonction **datetime now()** de Python donne l'objet datetime actuel. Cette fonction ne doit jamais être utilisée à l'intérieur d'une tâche, en particulier pour effectuer des calculs critiques, car **elle conduit à des résultats différents** à chaque exécution.



BEST PRACTICES

Supprimer une tâche

Il est important d'être prudent lors de la suppression d'une tâche d'un DAG, car **elle ne sera plus visible** dans la vue graphique, la vue en grille, etc., ce qui **compliquera la vérification des journaux de cette tâche** à partir du serveur Web.

Si cela n'est pas souhaité, il est recommandé de créer un nouveau DAG.

Communication

Lorsque **l'exécuteur Kubernetes** ou **Celery** est utilisé, les tâches d'un DAG sont exécutées **sur différents serveurs** par Airflow.

Il est donc **déconseillé de stocker des fichiers ou des configurations sur le système de fichiers local**, car il est possible que la tâche suivante s'exécute sur un serveur différent sans y avoir accès.

Cela peut être illustré par exemple avec une tâche qui télécharge un fichier de données pour que la tâche suivante le traite.

Dans le cas de **l'exécuteur Local**, le stockage d'un fichier sur le disque **peut compliquer** les tentatives de réexécution, comme lorsque qu'une tâche supprime le fichier de configuration dont une autre tâche a besoin pour s'exécuter.





BEST PRACTICES

Code de niveau supérieur en Python

Il est recommandé **d'éviter d'écrire du code de niveau supérieur** qui n'est pas nécessaire pour créer des opérateurs et établir des relations entre eux dans un DAG.

Ceci est dû à la décision de conception pour le planificateur d'Airflow et à l'impact de la vitesse de traitement du code de niveau supérieur sur les performances et la scalabilité d'Airflow.

Dynamic DAG Generation

Parfois, écrire des DAG manuellement n'est pas pratique. Peut-être avez-vous beaucoup de DAG qui font des choses similaires avec juste un paramètre qui change entre eux. Ou peut-être avez-vous besoin d'un ensemble de DAG pour charger des tables, mais vous ne voulez pas mettre à jour manuellement les DAG chaque fois que ces tables changent. Dans ces cas-là et d'autres encore, il peut être plus utile de générer dynamiquement des DAG.



BEST PRACTICES

Dynamic DAG Generation

➤ Single-file methods

Une méthode pour générer dynamiquement des DAG est d'avoir un seul fichier Python qui génère des DAG en fonction de certains paramètres d'entrée. Par exemple, une liste d'API ou de tables. Un cas d'utilisation courant pour cela est une pipeline de type ETL ou ELT où il y a de nombreuses sources ou destinations de données. Cela nécessite la création de nombreux DAG qui suivent tous un modèle similaire.

Certains avantages de la méthode à fichier unique :

- C'est facile à mettre en œuvre.
- Il peut accueillir des paramètres d'entrée provenant de sources très diverses.
- L'ajout de DAG est presque instantané car il ne nécessite que de modifier les paramètres d'entrée.



BEST PRACTICES

Dynamic DAG Generation

➤ Single-file methods

La méthode à fichier unique présente les désavantages suivants :

- Votre visibilité sur le code derrière un DAG spécifique est limitée car aucun fichier DAG n'est créé.
- Le code de génération est exécuté chaque fois que le DAG est analysé car cette méthode nécessite un fichier Python dans le dossier des DAGs. La fréquence à laquelle cela se produit est contrôlée par le paramètre "min_file_process_interval". Cela peut causer des problèmes de performance si le nombre total de DAG est élevé, ou si le code se connecte à un système externe tel qu'une base de données.



BEST PRACTICES

Dynamic DAG Generation

➤ Single-file methods

Dans l'exemple donné, les paramètres d'entrée peuvent provenir de n'importe quelle source accessible au script Python.

Pour générer ces paramètres uniques et les enregistrer en tant que DAG valides auprès du planificateur Airflow, il suffit de définir une boucle simple (par exemple, `range(1, 4)`) et de transmettre les paramètres à la portée globale.

```
from pendulum import datetime
from airflow import DAG
from airflow.operators.python import PythonOperator
def create_dag(dag_id, schedule, dag_number, default_args):
    def hello_world_py(*args):
        print("Hello World")
        print("This is DAG: {}".format(str(dag_number)))
    dag = DAG(dag_id, schedule=schedule, default_args=default_args)
    with dag:
        t1 = PythonOperator(task_id="hello_world",
                            python_callable=hello_world_py)
    return dag
# build a dag for each number in range(10)
for n in range(1, 4):
    dag_id = "loop_hello_world_{}".format(str(n))
    default_args = {"owner": "airflow", "start_date": datetime(2021, 1, 1)}
    schedule = "@daily"
    dag_number = n
    globals()[dag_id] = create_dag(dag_id, schedule, dag_number,
                                   default_args)
```



BEST PRACTICES

Dynamic DAG Generation

➤ Generate DAGs from connections

Une autre manière de définir les paramètres d'entrée pour les DAG générés de manière dynamique consiste à définir des connexions Airflow. Cette option peut être intéressante si chaque DAG doit se connecter à une base de données ou une API. Étant donné que la configuration des connexions est nécessaire de toute façon, créer les DAG à partir de cette source permet d'éviter un travail redondant.



BEST PRACTICES

Dynamic DAG Generation

➤ Generate DAGs from connections

Une autre manière de définir les paramètres d'entrée pour les DAG générés de manière dynamique consiste à définir des connexions Airflow. Cette option peut être intéressante si chaque DAG doit se connecter à une base de données ou une API. Étant donné que la configuration des connexions est nécessaire de toute façon, créer les DAG à partir de cette source permet d'éviter un travail redondant.



BEST PRACTICES

Dynamic DAG Generation

➤ Generate DAGs from connections

Pour implémenter cette méthode, la récupération des connexions depuis la base de données de métadonnées d'Airflow se fait en instantiant la session et en interrogeant la table de connexion. Il est également possible de filtrer cette requête pour ne récupérer que les connexions qui correspondent à des critères spécifiques.

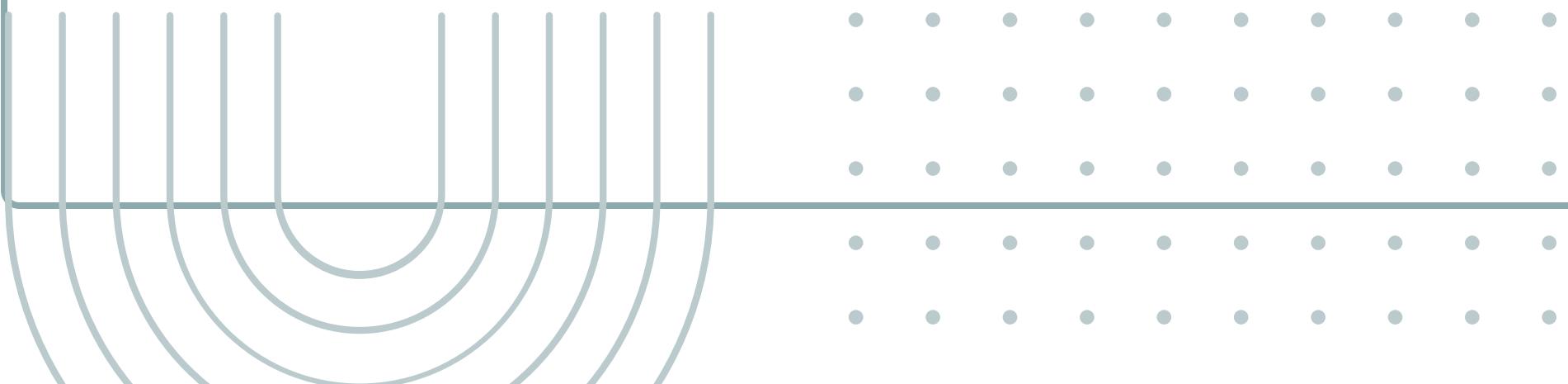
```
from pendulum import datetime
from airflow import DAG, settings
from airflow.models import Connection
from airflow.operators.python import PythonOperator
def create_dag(dag_id, schedule, dag_number, default_args):
    def hello_world_py(*args):
        print("Hello World")
        print("This is DAG: {}".format(str(dag_number)))
    dag = DAG(dag_id, schedule=schedule, default_args=default_args)
    with dag:
        t1 = PythonOperator(task_id="hello_world",
                            python_callable=hello_world_py)
    return dag
session = settings.Session()
conns = (
    session.query(Connection.conn_id)
    .filter(Connection.conn_id.ilike("%MY_DATABASE_CONN%"))
    .all()
)
for conn in conns:
    dag_id = "connection_hello_world_{}".format(conn[0])
    default_args = {"owner": "airflow", "start_date": datetime(2018, 1, 1)}
    schedule = "@daily"
    dag_number = conn
    globals()[dag_id] = create_dag(dag_id, schedule, dag_number, default_args)
```



ATELIER

08.

AIRFLOW OPTIMISATION



BOTTLENECK & DEADLOCKS



Un **bottleneck** se réfère à une situation où les performances d'un système sont limitées par une ressource particulière, telle que le processeur, la mémoire, l'E/S de disque ou la bande passante réseau.

Un bottleneck peut ralentir l'ensemble du système et réduire son débit, mais cela ne se traduit pas nécessairement par une défaillance complète du système.

Un **deadlock** se produit lorsque deux ou plusieurs tâches attendent des ressources détenues par chacune d'entre elles, créant une dépendance circulaire où aucune des tâches ne peut avancer. Une impasse peut entraîner l'arrêt complet du système, car les tâches impliquées dans l'impasse ne peuvent pas progresser.

En résumé :

un bottleneck est une limitation de la capacité du système qui réduit ses performances globales,

un deadlock est une situation où le système est incapable de progresser en raison d'une dépendance circulaire.

Il est important de surveiller et de traiter à la fois les bottlenecks et les deadlock dans Apache Airflow pour garantir que le système fonctionne de manière fluide et efficace.



DEADLOCK : CAUSES



Les **deadlocks** peuvent survenir dans Apache Airflow lorsque deux ou plusieurs tâches attendent des ressources détenues par l'autre. Cela peut entraîner une dépendance circulaire où aucune des tâches ne peut progresser, entraînant un deadlock.

Exemples :

Ressources partagées : Lorsque deux ou plusieurs tâches ont besoin d'accéder à la même ressource, telle qu'une table de base de données, un fichier ou un fichier de verrouillage. Si les tâches ne sont pas coordonnées correctement, elles peuvent finir par attendre l'une de l'autre indéfiniment, provoquant un deadlock.

Dépendances entre tâches : Lorsque deux ou plusieurs tâches sont dépendantes l'une de l'autre et attendent que l'autre se termine. Par exemple, si la tâche A attend que la tâche B se termine et que la tâche B attend que la tâche A se termine, un deadlock peut se produire.



DEADLOCK : CAUSES

Systèmes distribués: Les deadlocks peuvent se produire dans les systèmes distribués lorsque plusieurs tâches ou processus s'exécutent sur différents nœuds, et chaque nœud attend des ressources détenues par d'autres nœuds. Si la coordination entre les nœuds n'est pas effectuée correctement, cela peut entraîner un deadlock.



PRÉVENTION DES DEADLOCK

- **Utiliser une file d'attente** de tâches pour gérer les dépendances entre les tâches et éviter les deadlocks.

Cette file d'attente permet d'exécuter les tâches dans le bon ordre et de s'assurer que toutes les dépendances sont satisfaites avant leur exécution.

- **Mettre en place des délais d'attente** pour les tâches, afin de les terminer et de libérer les ressources qu'elles détiennent si elles attendent trop longtemps.
- **Limiter la concurrence** en définissant un nombre maximum de tâches concurrentes, ce qui permet d'éviter la surcharge du système et les deadlocks.
- **Surveiller le système** pour détecter les deadlocks et les autres problèmes éventuels, et prendre les mesures correctives nécessaires.



BOTTLENECK : CAUSES ET TRAITEMENTS

- **Limitations de ressources**: les jobs **Airflow** peuvent être intensifs en CPU ou en mémoire, et si les ressources disponibles sur les nœuds worker sont limitées, cela peut causer un bottleneck.

Solution

Pour éviter que des limitations de ressources ne causent des bottlenecks, il est possible de prendre les mesures suivantes :

- **Augmenter la capacité des nœuds workers** en ajoutant plus de CPU, de mémoire ou d'espace disque pour gérer des charges de travail plus importantes.
- **Utiliser un gestionnaire de cluster** tel que Kubernetes pour répartir la charge de travail sur plusieurs nœuds workers et ajuster automatiquement leur nombre en fonction des besoins.
- **Définir des limites de ressources** pour chaque tâche individuelle afin d'éviter qu'elles n'utilisent excessivement les ressources et qu'elles empêchent d'autres tâches de fonctionner.



BOTTLENECK : CAUSES ET TRAITEMENTS

Dépendances : les jobs Airflow ont souvent des dépendances sur d'autres tâches ou services externes, et si ces dépendances ne sont pas satisfaites, cela peut causer un bottleneck . Par exemple, si une tâche attend des données provenant d'un service externe, et que ce service est lent ou en panne, cela peut causer un goulot d'étranglement. Pour éviter cela, vous pouvez utiliser des mécanismes de réessayai ou mettre en place des solutions de repli pour les dépendances critiques.

Solution

Pour éviter que les dépendances ne causent des ralentissements, des solutions sont les suivantes :

- Utiliser des mécanismes de réessayai pour relancer automatiquement les tâches échouées qui dépendent de services externes.
- Mettre en place des solutions de secours pour les dépendances critiques, de sorte que si le service principal est indisponible, la tâche puisse basculer vers un service secondaire.
- Surveiller les services externes et alerter l'équipe en cas de problèmes de performance ou d'interruptions.



BOTTLENECK : CAUSES ET TRAITEMENTS

Transfert de données : le déplacement de données entre les tâches ou les étapes du pipeline peut être un goulot d'étranglement si la quantité de données est importante ou si la vitesse de transfert est lente. Pour éviter cela, vous pouvez optimiser le processus de transfert de données en utilisant la compression, la parallélisation ou des systèmes de fichiers distribués comme Hadoop ou S3.

Solution

Pour éviter que le transfert de données ne cause des problèmes de performance, il est possible de :

- Optimiser le processus de transfert de données en utilisant des techniques telles que la compression, la parallélisation ou des systèmes de fichiers distribués comme Hadoop ou S3.
- Surveiller le débit de transfert de données et ajuster la méthode ou les paramètres de transfert si nécessaire.
- Utiliser des techniques de mise en cache ou de préchargement pour réduire la quantité de données à transférer.



BOTTLENECK : CAUSES ET TRAITEMENTS

Planification : si l'intervalle de planification pour une tâche est trop court, cela peut causer un bottleneck en surchargeant les nœuds worker. Alternativement, si l'intervalle est trop long, cela peut causer des retards et ralentir le pipeline. Pour éviter cela, vous pouvez ajuster l'intervalle de planification en fonction de la charge de travail et des ressources disponibles.

Solution

Afin d'éviter que la planification ne provoque des goulots d'étranglement, les mesures suivantes peuvent être prises :

- Établir des intervalles de planification adaptés à la charge de travail et aux ressources disponibles.
- Prioriser les tâches en fonction de leur importance et de leur échéance, de manière à exécuter en premier les tâches critiques.
- Surveiller la file d'attente des tâches et ajuster le nombre de tâches simultanées en fonction des ressources disponibles.

JINJA TEMPLATING



Jinja est un moteur de template rapide, expressif et extensible. Des espaces réservés spéciaux dans le template permettent d'écrire du code similaire à la syntaxe de Python. Ensuite, les données sont transmises au template pour générer le document final.

Pour cet exemple, considérons que la transmission de la date de début de l'intervalle de données sous forme de variable d'environnement à un script Bash est souhaitée, en utilisant l'opérateur BashOperator:

Dans cet exemple, la variable `ds` est utilisée comme variable de template. Comme le paramètre "env" de l'opérateur Bash est également un template Jinja, la date de début de l'intervalle de données sera rendue disponible en tant que variable d'environnement nommée "DATA_INTERVAL_START" dans le script Bash.

```
# The start of the data interval as YYYY-MM-DD
date = "{{ ds }}"
t = BashOperator(
    task_id="test_env",
    bash_command="/tmp/test.sh ",
    dag=dag,
    env={"DATA_INTERVAL_START": date},
)
```

JINJA TEMPLATING



Il est possible d'utiliser le templating Jinja avec tous les paramètres marqués comme "templated" dans la documentation. La substitution des modèles a lieu juste avant l'appel de la fonction pre_executede l'opérateur.

Il est également possible d'utiliser le templating Jinja avec des champs imbriqués, pour autant que ces champs soient marqués comme étant des templates dans la structure à laquelle ils appartiennent.

Les champs enregistrés dans la propriété template_fields seront soumis à la substitution de template, par exemple le champ "path" dans l'exemple :

```
class MyDataReader:  
    template_fields: Sequence[str] = ("path",)  
  
    def __init__(self, my_path):  
        self.path = my_path  
  
        # [additional code here...]  
  
    t = PythonOperator(  
        task_id="transform_data",  
        python_callable=transform_data,  
        op_args=[MyDataReader("/tmp/{{ ds }}/my_file")],  
        dag=dag,  
    )
```



JINJA TEMPLATING



Des champs profondément imbriqués peuvent également être substitués, pour autant que tous les champs intermédiaires soient marqués comme des champs de modèle :

```
class MyDataTransformer:

    template_fields: Sequence[str] = ("reader",)

    def __init__(self, my_reader):
        self.reader = my_reader
        # [additional code here...]

class MyDataReader:

    template_fields: Sequence[str] = ("path",)

    def __init__(self, my_path):
        self.path = my_path
        # [additional code here...]

t = PythonOperator(
    task_id="transform_data",
    python_callable=transform_data,
    op_args=[MyDataTransformer(MyDataReader("/tmp/{{ ds }}/my_file"))],
    dag=dag,
```



JINJA TEMPLATING



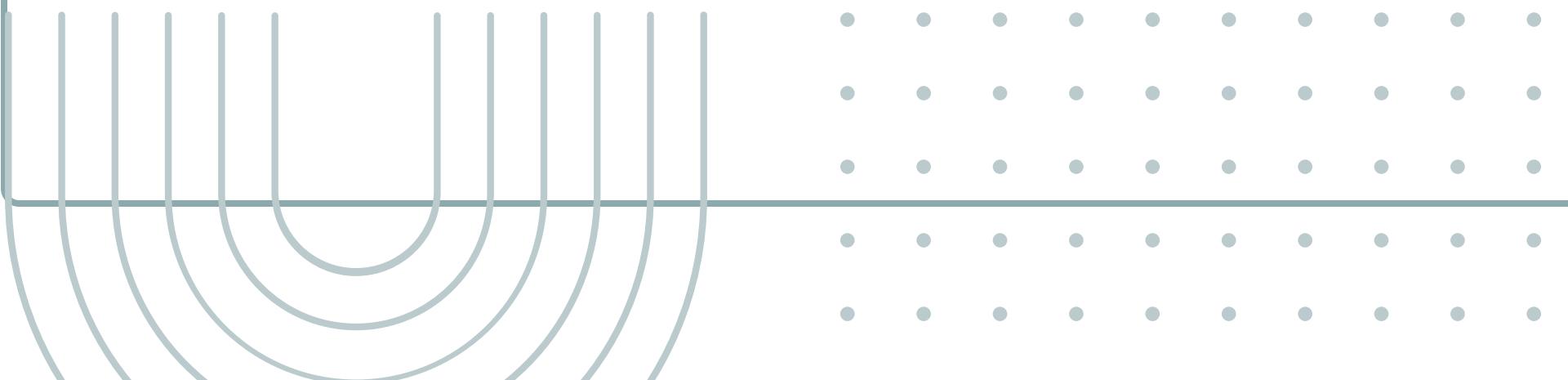
Il est possible de transmettre des options personnalisées à l'environnement Jinja lors de la création de votre DAG.

Une utilisation courante consiste à éviter que Jinja ne supprime une nouvelle ligne en fin de chaîne de modèle :

```
my_dag = DAG(  
  
    dag_id="my-dag",  
  
    jinja_environment_kwargs={  
  
        "keep_trailing_newline": True,  
  
        # some other jinja2 Environment options here  
  
    },  
  
)
```

09.

DAG BRANCHING

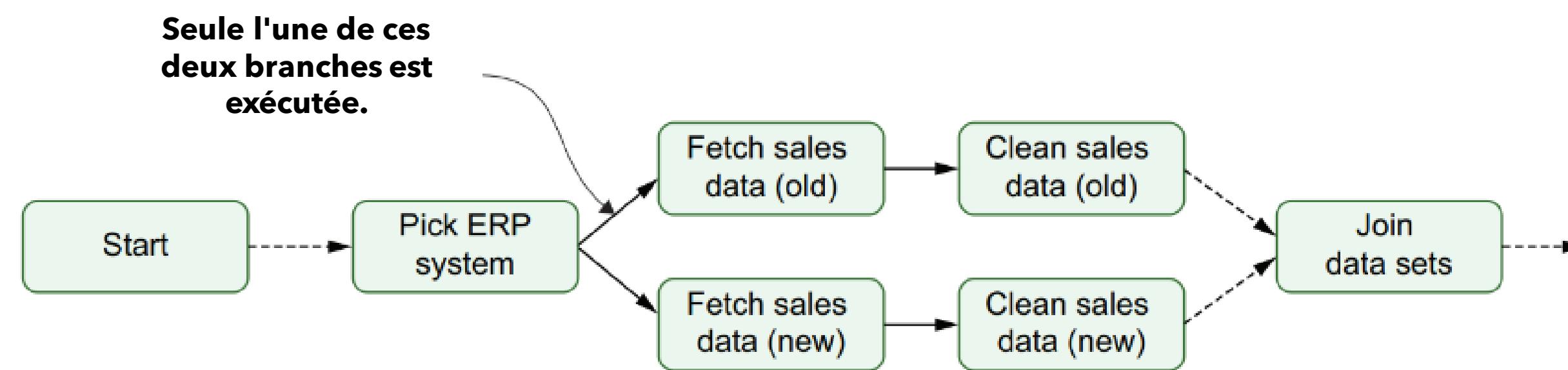




BRANCHING DANS LE DAG

Une autre façon de prendre en charge les deux systèmes ERP différents dans un DAG unique est de développer deux ensembles distincts de tâches (un pour chaque système) et de permettre au DAG de choisir s'il doit exécuter les tâches de récupération de données à partir de l'ancien ou du nouveau système ERP .

La création des deux ensembles de tâches est relativement simple : il suffit de créer des tâches pour chaque système ERP séparément en utilisant les opérateurs appropriés et de lier les tâches respectives.





BRANCHING DANS LE DAG

Ajout de tâches supplémentaires de récupération/nettoyage

Il est maintenant nécessaire de connecter ces tâches au reste du DAG et de s'assurer qu'Airflow sait laquelle exécuter.

Heureusement, Airflow inclut une fonctionnalité intégrée pour choisir entre des ensembles de tâches aval en utilisant l'opérateur

BranchPythonOperator.

Cet opérateur ressemble au **PythonOperator** car il prend un callable Python comme l'un de ses arguments principaux.

```
fetch_sales_old = PythonOperator(...)

clean_sales_old = PythonOperator(...)

fetch_sales_new = PythonOperator(...)

clean_sales_new = PythonOperator(...)

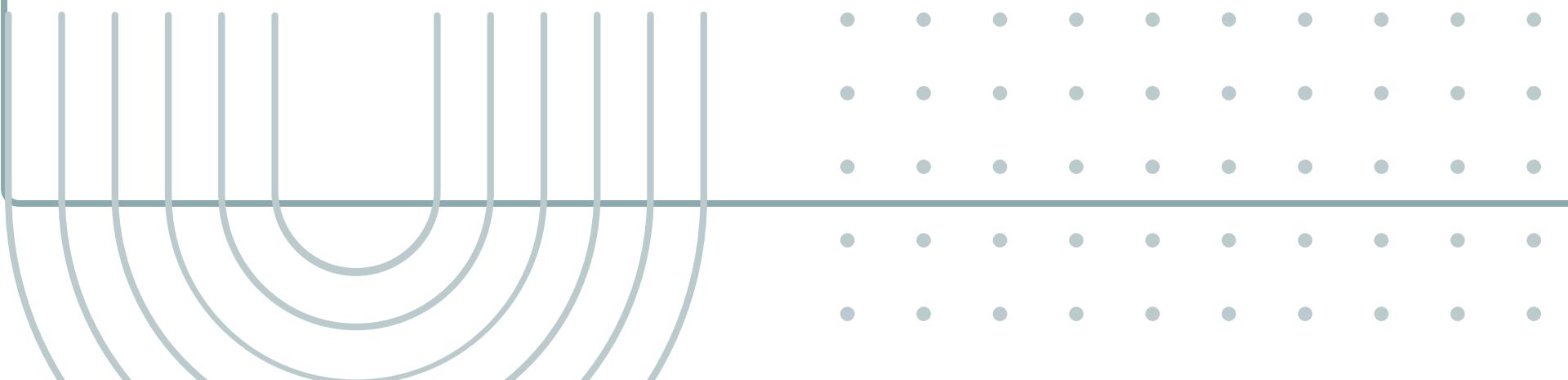
fetch_sales_old >> clean_sales_old

fetch_sales_new >> clean_sales_new

[clean_sales_old, clean_sales_new] >> join_datasets
```

10.

OPTIMISER LE TRAITEMENT DES DONNEES



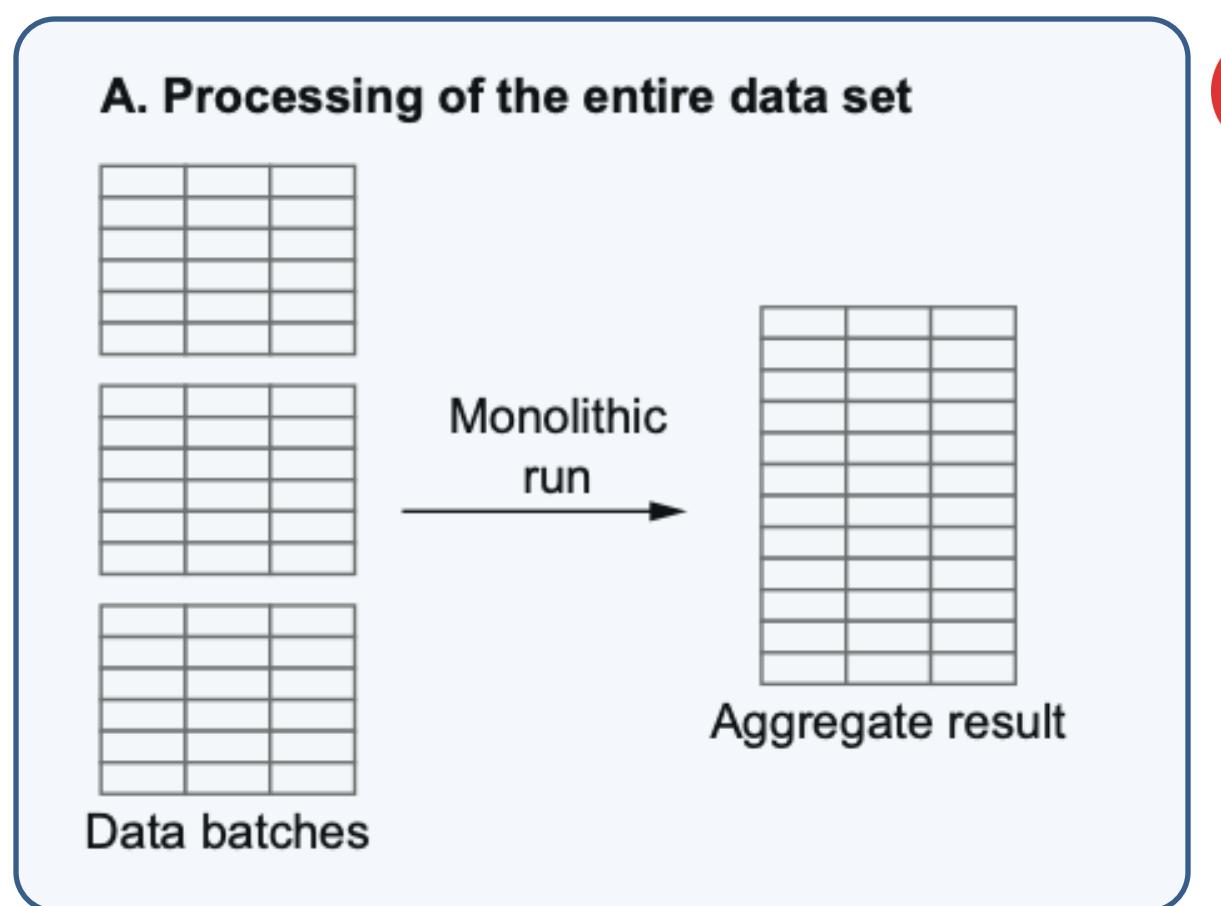


POLLING CUSTOM CONDITIONS

- Un cas d'utilisation courant pour démarrer un flux de travail est l'arrivée de nouvelles données .Example d'un tiers livre un dépôt quotidien de ses données sur un système de stockage partagé entre votre entreprise et ce tiers.
- Prenons l'exemple où une application de coupons mobiles populaire est en cours de développement et que des contacts ont été établis avec toutes les marques de supermarchés pour recevoir une exportation quotidienne de leurs promotions qui seront ensuite affichées dans l'application de coupons.
- Actuellement, les promotions sont principalement un processus manuel : la plupart des supermarchés emploient des analystes de prix pour prendre en compte de nombreux facteurs et fournir des promotions précises. Certaines promotions sont bien réfléchies des semaines à l'avance, et d'autres sont des ventes flash spontanées d'un jour.



EFFICIENT DATA HANDLING





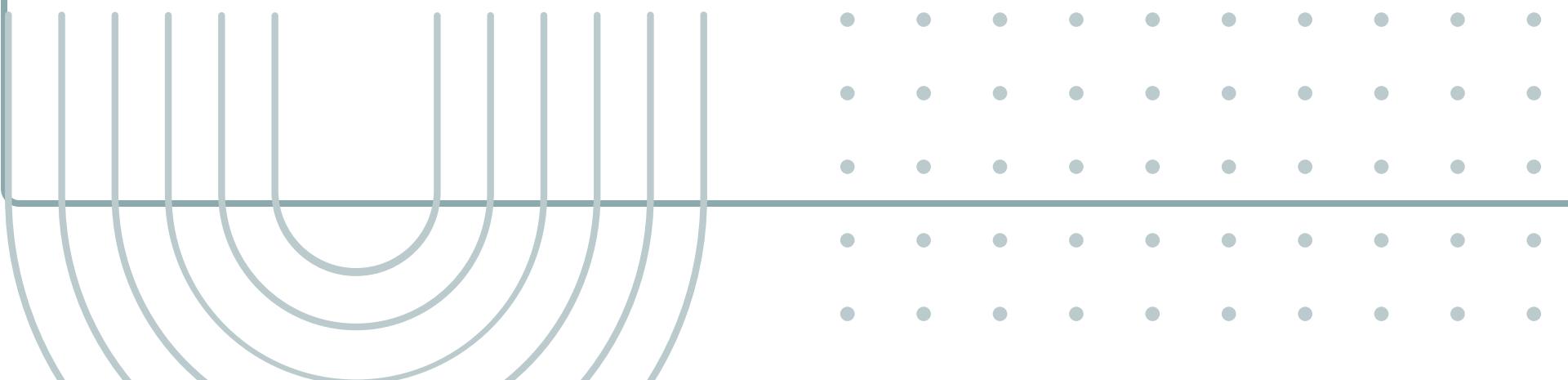
EFFICIENT DATA HANDLING

General Advice

- *Don't store data on local file systems*
- *Offload work to external/source systems*

11.

GESTION DE CONCURRENCE AVEC LES POOLS DE RESSOURCES





POOLS DE RESSOURCES

- Lors de l'exécution de nombreuses tâches en parallèle, vous pouvez rencontrer des situations où plusieurs tâches ont besoin d'accéder au même ressource.
- Cela peut rapidement submerger ladite ressource si elle n'est pas conçue pour gérer ce type de concurrence.
- Exemple : Ressources partagées du style base de données, système GPU, clusters Spark
(si, par exemple, vous souhaitez limiter le nombre de jobs en cours d'exécution sur un cluster donné.)
- Airflow vous permet de contrôler le nombre de tâches ayant accès à une ressource donnée à l'aide de **pools de ressources**,
- Chaque pool contient un nombre fixe de slots, qui accordent l'accès à la ressource correspondante.
- Les tâches individuelles ayant besoin d'accéder à la ressource peuvent être assignées au pool de ressources,
- Cela indique au Scheduler qu'il doit obtenir un slot à partir du pool avant de pouvoir planifier la tâche correspondante.



POOLS DE RESSOURCES

Airflow Admin - List Pool

Record Count: 1

Pool	Slots	Running Slots	Queued Slots
default_pool	128	0	0

Pool names Number of slots in each pool Pool statistics

Airflow Admin - Add Pool

Name of the pool: my_resource_pool

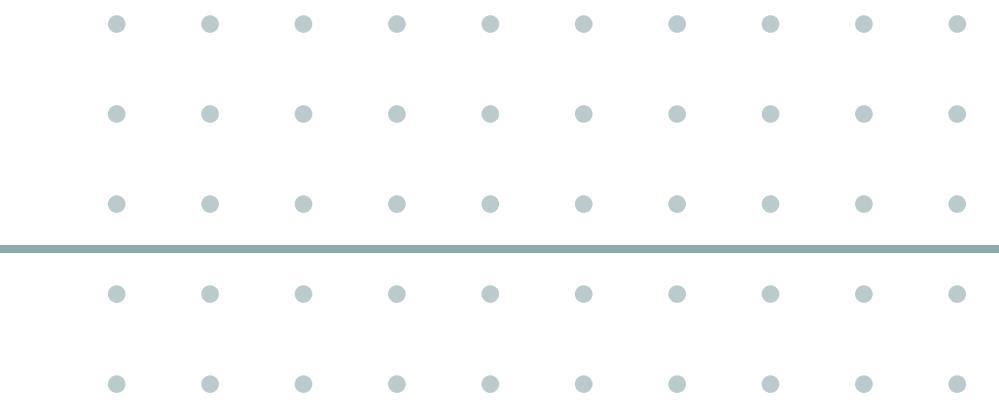
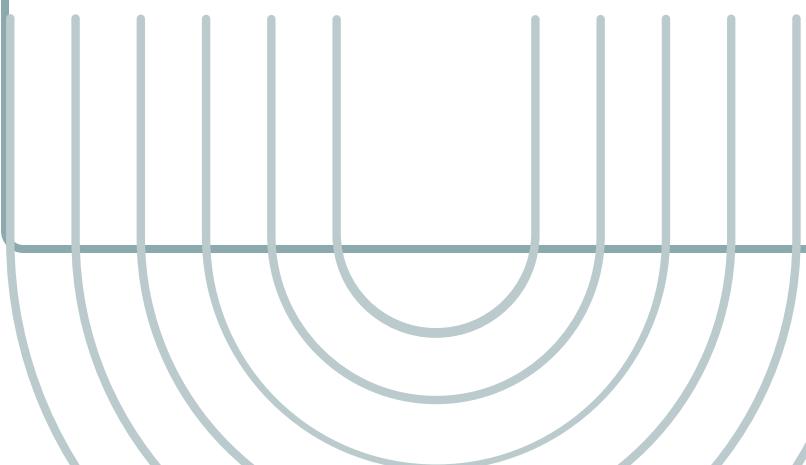
Number of slots for this pool: 10

Description:

Save

12.

MONITORING W/NOTIFICATIONS





ALERT MONITORING

```
from pendulum import datetime
from airflow import DAG

default_args = {
    "owner": "airflow",
    "start_date": datetime(2018, 1, 30),
    "email": ["noreply@astronomer.io"],
    "email_on_failure": True,
}

with DAG(
    "sample_dag", default_args=default_args, schedule="@daily", catchup=False
) as dag:

    ...
```



ALERT MONITORING

```
from pendulum import datetime
from airflow import DAG
from airflow.operators.empty import EmptyOperator

default_args = {
    "owner": "airflow",
    "start_date": datetime(2018, 1, 30),
    "email_on_failure": False,
    "email": ["noreply@astronomer.io"],
    "retries": 1,
}

with DAG(
    "sample_dag", default_args=default_args, schedule="@daily", catchup=False
) as dag:
    wont_email = EmptyOperator(task_id="wont_email")

    will_email = EmptyOperator(task_id="will_email", email_on_failure=True)
```



ALERT MONITORING

```
[smtp]
# If you want airflow to send emails on retries, failure, and you want to use
# the airflow.utils.email.send_email_smtp function, you have to configure an
# smtp server here
smtp_host = your-smtp-host.com
smtp_starttls = True
smtp_ssl = False
# Uncomment and set the user/pass settings if you want to use SMTP AUTH
# smtp_user =
# smtp_password =
smtp_port = 587
smtp_mail_from = noreply@astronomer.io
```

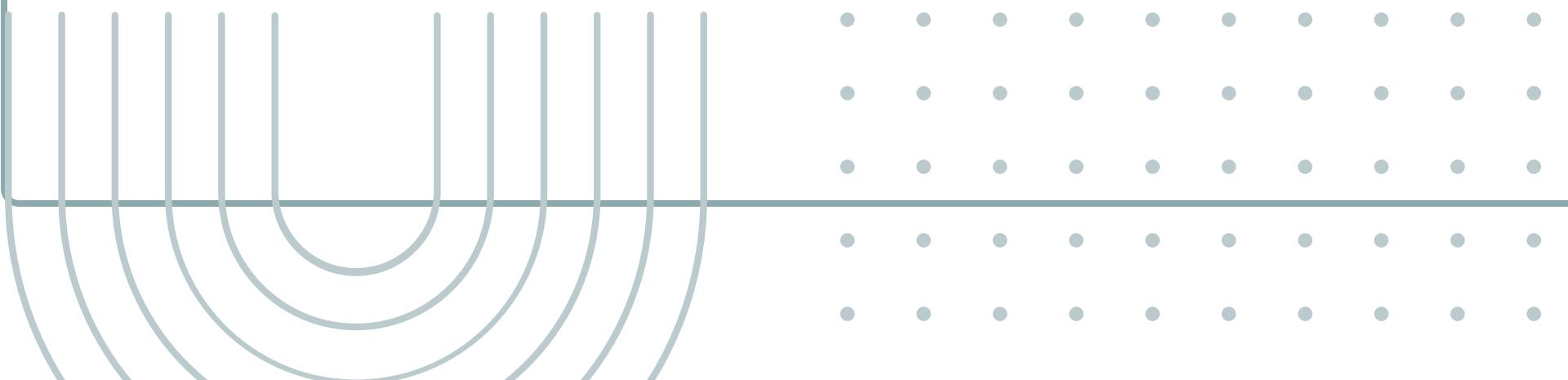


ALERT MONITORING

```
default_subject = 'Airflow alert: {{ti}}'  
# For reporting purposes, the report is based on 1-indexed,  
# not 0-indexed lists (i.e. Try 1 instead of  
# Try 0 for the first attempt).  
default_html_content = (  
    'Try {{try_number}} out of {{max_tries + 1}}<br>'  
    'Exception:<br>{{exception_html}}<br>'  
    'Log: <a href="{{ti.log_url}}>Link</a><br>'  
    'Host: {{ti.hostname}}<br>'  
    'Mark success: <a href="{{ti.mark_success_url}}>Link</a><br>'  
)
```

13.

MONITORING LONG RUNNING TASK W/ SLA





SLA MONITORING

- Dans certains cas, vos tâches ou l'exécution de votre DAG peuvent prendre plus de temps que d'habitude en raison de problèmes imprévus dans les données, de ressources limitées, etc.
- Airflow vous permet de surveiller le comportement de vos tâches à l'aide de son mécanisme **SLA** (service level agreement / accord de niveau de service).
- Cette fonctionnalité vous permet efficacement d'assigner des délais d'attente à vos DAG ou tâches, auquel cas Airflow vous avertira si l'une de vos tâches ou DAG dépasse son SLA (c'est-à-dire prend plus de temps pour s'exécuter que le délai d'attente SLA spécifié).



SLA MONITORING



Airflow

DAGs

Security ▾

Browse ▾

Admin ▾

Docs ▾

Astronomer ▾

17:16 UTC ▾

AU ▾

List Sla Miss

Search ▾



Record Count: 3

Dag Id	Task Id	Logical Date	Email Sent	Timestamp
five_sla_misses	miss	2020-01-01, 03:00:00	False	2021-12-29, 17:12:19
five_sla_misses	miss	2009-12-31, 16:48:00	False	2021-12-29, 17:12:19
five_sla_misses	miss	2000-01-01, 06:36:00	False	2021-12-29, 17:12:19



SLA MONITORING

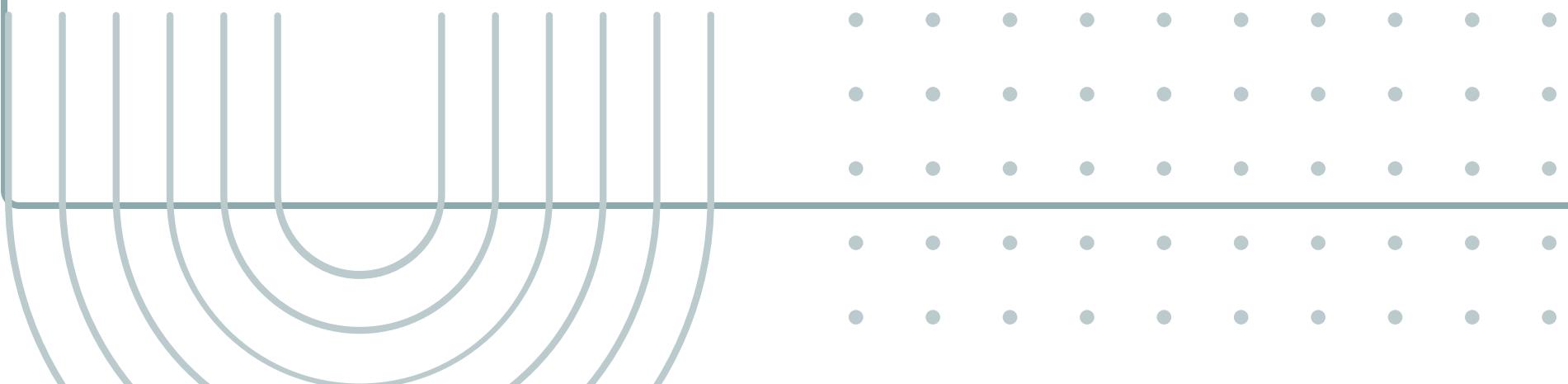
```
from datetime import timedelta  
  
default_args = {  
    "sla": timedelta(hours=2),  
    ...  
}  
  
with DAG(  
    dag_id="...",  
    ...  
    default_args=default_args,  
) as dag:  
    ...
```

```
def sla_miss_callback(context):  
    send_slack_message("Missed SLA!")  
  
    ...  
  
with DAG(  
    ...  
    sla_miss_callback=sla_miss_callback  
) as dag:  
    ...
```

```
PythonOperator(  
    ...  
    sla=timedelta(hours=2)  
)
```

14.

POLLING W/ SENSORS





POLLING W/ SENSORS

- Un cas d'utilisation courant pour démarrer un flux de travail est l'arrivée de nouvelles données .
(Exemple d'un tiers qui livre un dépôt quotidien de ses données sur un système de stockage partagé entre votre entreprise et ce tiers.)
- Prenons l'exemple où une application de coupons est en cours de développement et que des contacts ont été établis avec toutes les marques de supermarchés pour recevoir un export quotidien de leurs promotions, qui seront ensuite affichées dans l'application de coupons.
- Actuellement, les promotions sont principalement un processus manuel :
 - la plupart des supermarchés emploient des analystes de prix pour prendre en compte de nombreux facteurs et fournir des promotions précises. Certaines promotions sont bien réfléchies des semaines à l'avance, et d'autres sont des ventes flash spontanées d'un jour.



POLLING CUSTOM CONDITIONS

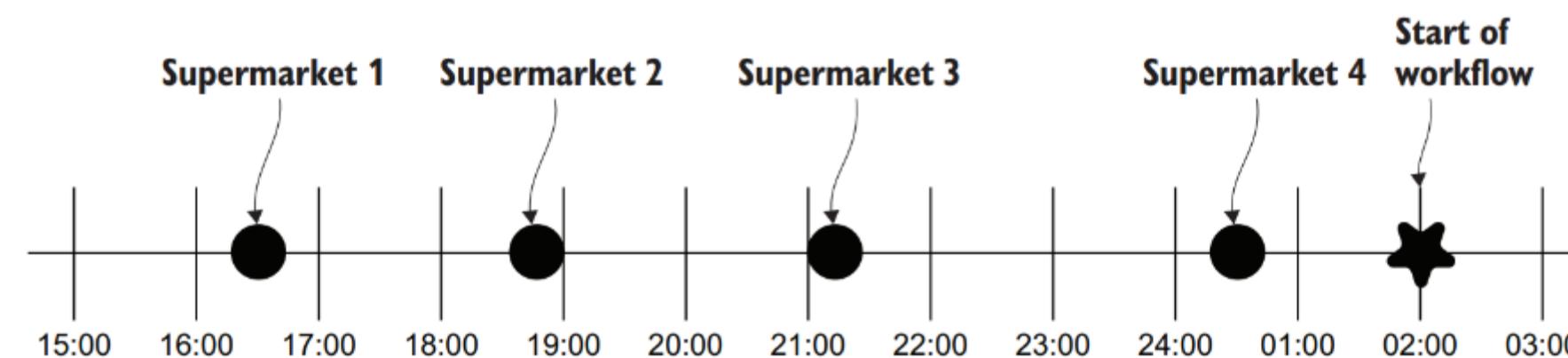
- Les analystes de prix étudient attentivement la concurrence, et parfois les promotions sont faites tard dans la nuit. Par conséquent, les données quotidiennes sur les promotions arrivent souvent à des moments aléatoires. Nous avons vu des données arriver sur le stockage partagé entre 16h00 et 2h00 le lendemain, bien que les données quotidiennes puissent être livrées à n'importe quel moment de la journée. Développons maintenant la logique initiale pour un tel flux de travail.
- Dans ce flux de travail, nous copions les données livrées par les supermarchés(1-4) dans notre propre stockage brut à partir duquel nous pouvons toujours reproduire les résultats.
- Les tâches `process_supermarket_{1,2,3,4}` transforment ensuite toutes les données brutes et les stockent dans une base de données de résultats qui peut être lue par l'application. Enfin, la tâche `create_metrics` calcule et agrège plusieurs métriques qui donnent des insights sur les promotions pour une analyse plus approfondie.





POLLING CUSTOM CONDITIONS

- Avec les données provenant des supermarchés arrivant à des moments différents, la chronologie de ce flux de travail pourrait ressembler à la figure

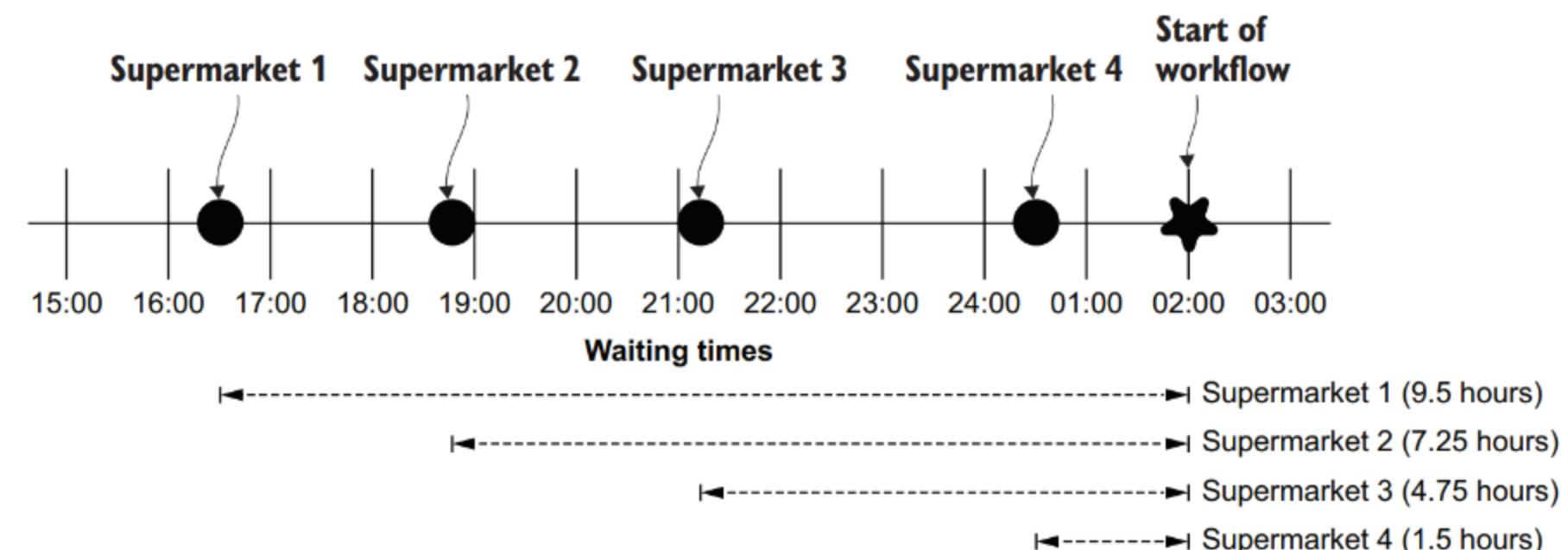


- Ici, nous voyons les heures de livraison des données des supermarchés et l'heure de début de notre flux de travail. Étant donné que nous avons déjà constaté que les supermarchés peuvent livrer des données aussi tard que 2h du matin, il serait prudent de démarrer le flux de travail à 2h pour être sûr que tous les supermarchés ont livré leurs données.



POLLING CUSTOM CONDITIONS

- Cependant, cela entraîne beaucoup de temps d'attente. Le supermarché 1 a livré ses données à 16h30, tandis que le flux de travail commence à traiter à 2h mais ne fait rien pendant 9,5 heures

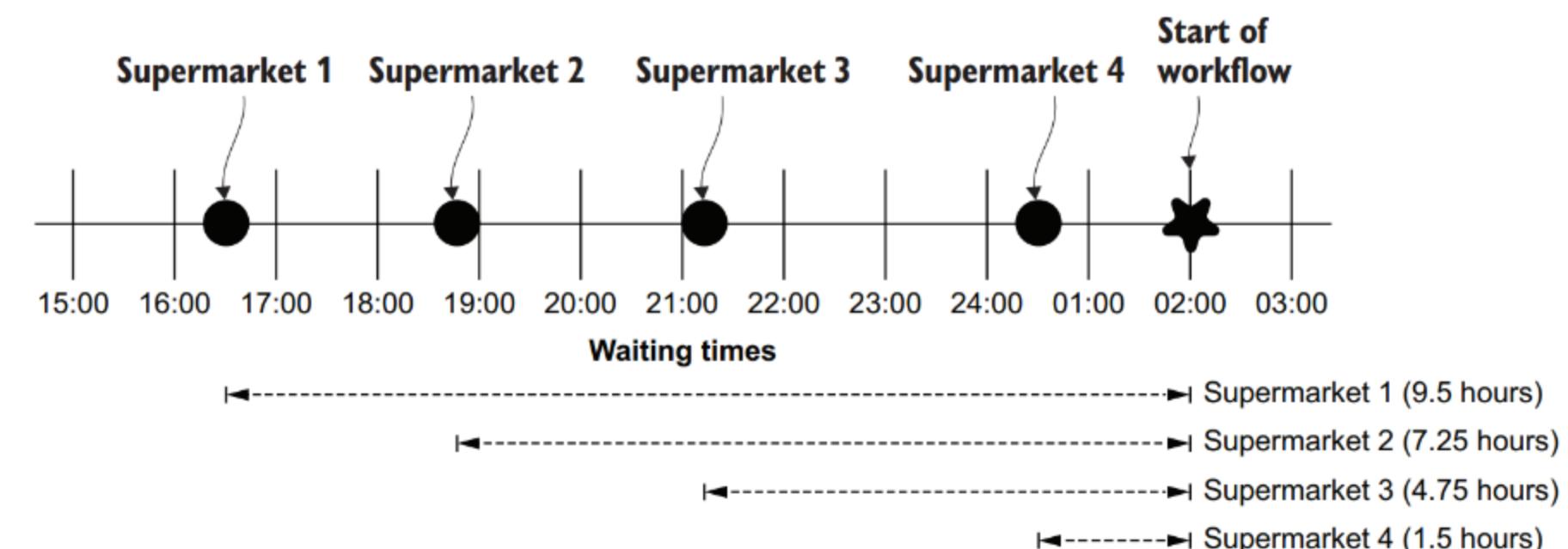


- Une façon de résoudre ce problème dans Airflow est d'utiliser des capteurs, qui sont un type spécial (sous-classe) d'opérateurs. Les capteurs sondent continuellement certaines conditions pour qu'elles soient vraies et réussissent si tel est le cas. Si la condition est fausse, le capteur attend et essaie à nouveau jusqu'à ce que la condition soit vraie ou qu'un délai d'attente soit finalement atteint.



POLLING W/ SENSORS

- Cependant, cela entraîne beaucoup de temps d'attente. Le supermarché 1 a livré ses données à 16h30, tandis que le flux de travail commence à traiter à 2h mais ne fait rien pendant 9,5 heures





POLLING CUSTOM CONDITIONS

- Un FileSensor attend qu'un chemin de fichier existe

```
from airflow.sensors.filesystem import FileSensor  
  
wait_for_supermarket_1 = FileSensor(  
    task_id="wait_for_supermarket_1",  
    filepath="/data/supermarket1/data.csv",  
)
```

- Ce FileSensor vérifiera l'existence de /data/supermarket1/data.csv et renverra true si le fichier existe. Sinon, il renvoie false et le capteur attendra une période donnée (par défaut 60 secondes) et réessayera. Les opérateurs (les capteurs sont également des opérateurs) et les DAG ont des délais d'attente configurables et le capteur continuera de vérifier la condition jusqu'à ce qu'un délai d'attente soit atteint. Nous pouvons inspecter la sortie des capteurs dans le journal de tâches.



POLLING CUSTOM CONDITIONS

- **Un FileSensor attend qu'un chemin de fichier existe**

```
from airflow.sensors.filesystem import FileSensor  
  
wait_for_supermarket_1 = FileSensor(  
    task_id="wait_for_supermarket_1",  
    filepath="/data/supermarket1/data.csv", )
```

- Ce FileSensor vérifiera l'existence de /data/supermarket1/data.csv et renverra true si le fichier existe. Sinon, il renvoie false et le capteur attendra une période donnée (par défaut 60 secondes) et réessayera. Les opérateurs (les capteurs sont également des opérateurs) et les DAG ont des délais d'attente configurables et le capteur continuera de vérifier la condition jusqu'à ce qu'un délai d'attente soit atteint. Nous pouvons inspecter la sortie des capteurs dans le journal de tâches.

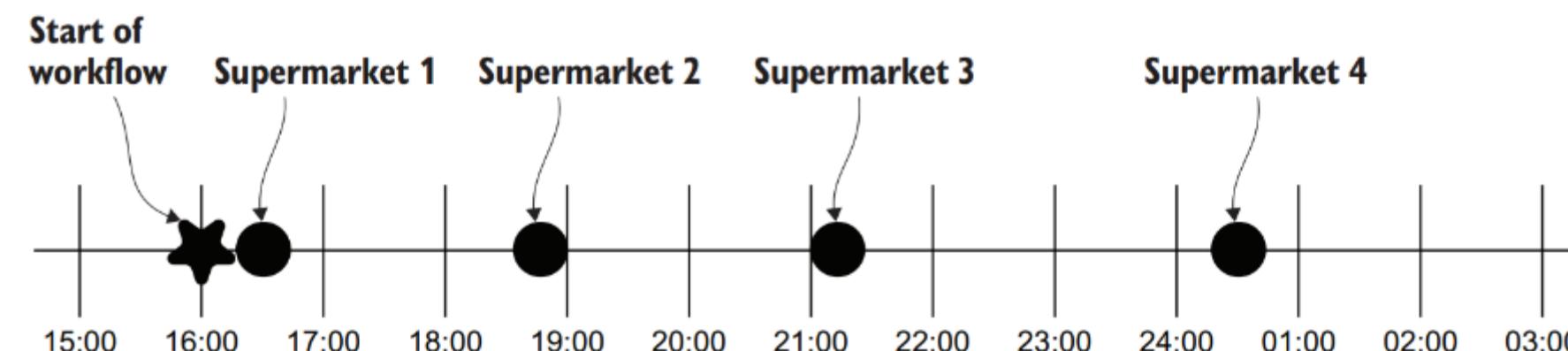
```
{file_sensor.py:60} INFO - Poking for file /data/supermarket1/data.csv  
{file_sensor.py:60} INFO - Poking for file /data/supermarket1/data.csv
```



POLLING CUSTOM CONDITIONS

- **Un FileSensor attend qu'un chemin de fichier existe**

Ici, nous voyons que le sensor vérifie environ une fois par minute si le fichier spécifié est disponible. Le fait de vérifier si une condition est remplie est appelé "poking" dans Airflow. Lorsque l'on incorpore des sensors dans ce workflow, un changement doit être effectué. Maintenant que nous savons que nous ne devons pas attendre jusqu'à 2h00 et supposer que toutes les données sont disponibles, mais plutôt commencer à vérifier en continu si les données sont disponibles, l'heure de début du DAG doit être placée au début des plages d'arrivée des données.



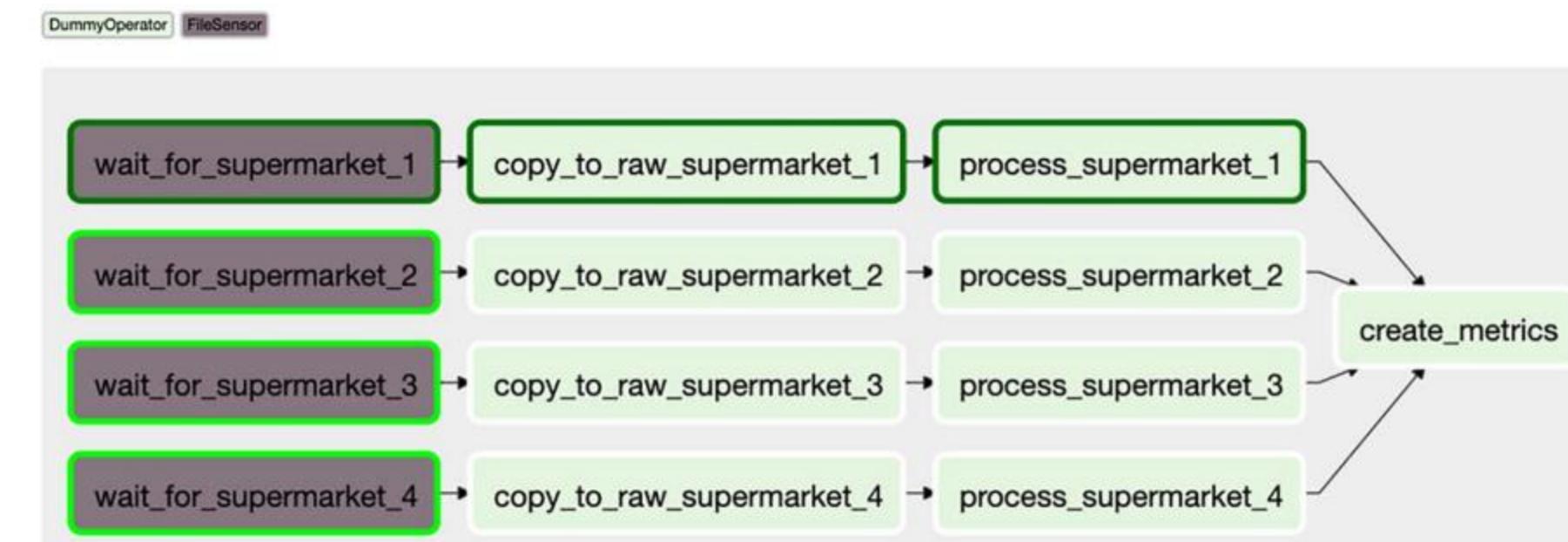


POLLING CUSTOM CONDITIONS

- **Un FileSensor attend qu'un chemin de fichier existe**

La tâche correspondante DAG (FileSensor) sera ajoutée au début du traitement des données de chaque supermarché et

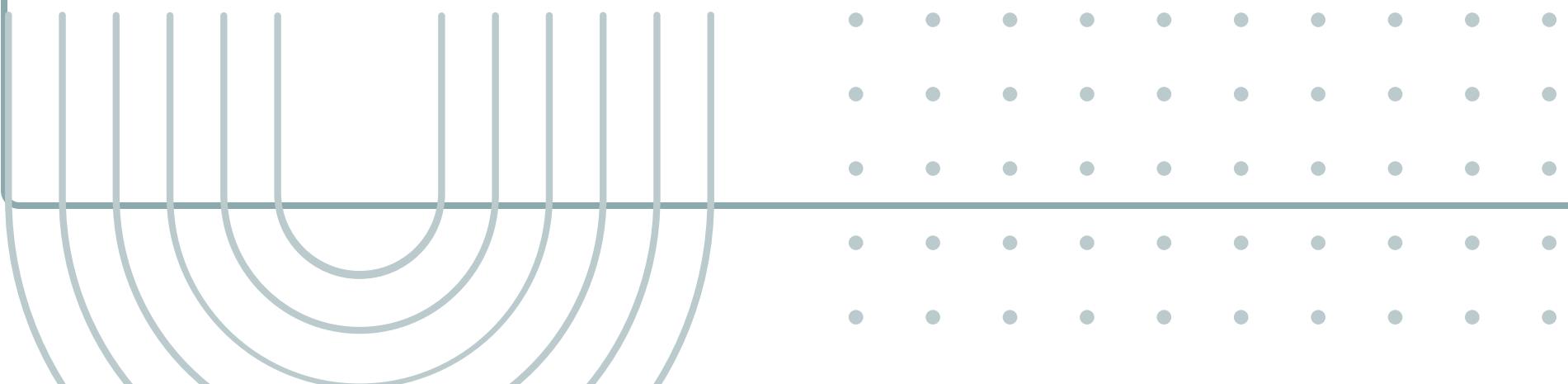
ressemblera à la figure



Dans la figure , des capteurs ont été ajoutés au début du DAG et l'intervalle de planification(schedule_interval) du DAG a été défini pour commencer avant la livraison prévue des données. Ainsi, les capteurs au début du DAG vont continuellement vérifier la disponibilité des données et passer à la tâche suivante une fois que la condition est remplie (c'est-à-dire une fois que les données sont disponibles dans le chemin donné).

15.

BACKFILLING W/ TRIGGER OPERATOR





BACKFILLING WITH THE TRIGGERDAGRUNOPERATOR

Que se passe-t-il si vous modifiez la logique des tâches process_* et que vous souhaitez relancer les DAG à partir de ce point ?

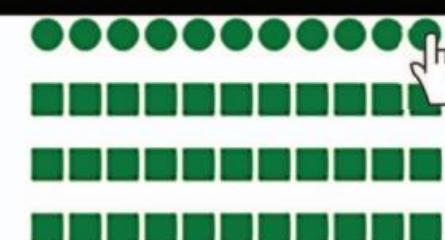
Dans un seul DAG, vous pourriez effacer l'état des tâches process_* et de celles en aval correspondantes. Cependant, effacer les tâches ne s'applique qu'aux tâches au sein du même DAG. Les tâches en aval d'un **TriggerDagRunOperator** dans un autre DAG ne sont pas effacées, il est donc important de bien comprendre ce comportement.

Effacer les tâches d'un DAG, y compris un TriggerDagRunOperator, déclenchera une nouvelle exécution de DAG au lieu de supprimer les exécutions de DAG précédemment déclenchées

Status: success
Run: 2020-12-19, 15:28:39 UTC
Run Id: manual_2020-12-19T15:28:39.682285+00:00
Started: 2020-12-19T15:28:39.698421+00:00
Duration:

UTC:
Started: 2020-12-19, 15:28:39
Ended: 2020-12-19, 15:28:41

Local: UTC (+00:00)
Started: 2020-12-19, 15:28:39
Ended: 2020-12-19, 15:28:41



16.

POLLING W/ EXTERNAL SENSORS





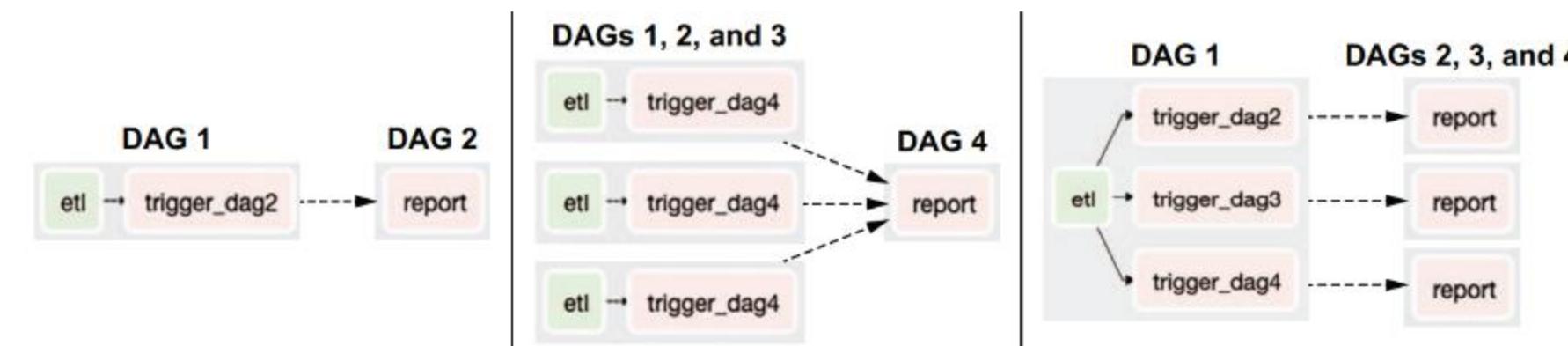
POLLING THE STATE OF OTHER DAG

L'exemple de la figure fonctionne tant qu'il n'y a pas de dépendance des DAG à déclencher vers le DAG déclencheur. En d'autres termes, le premier DAG peut déclencher le DAG aval à tout moment, sans avoir à vérifier des conditions quelconques.



Si les DAG deviennent très complexes, pour plus de clarté, le premier DAG pourrait être divisé en plusieurs DAG et une tâche

TriggerDagRunOperator correspondante pourrait être créée pour chaque DAG correspondant, comme on le voit dans la figure :



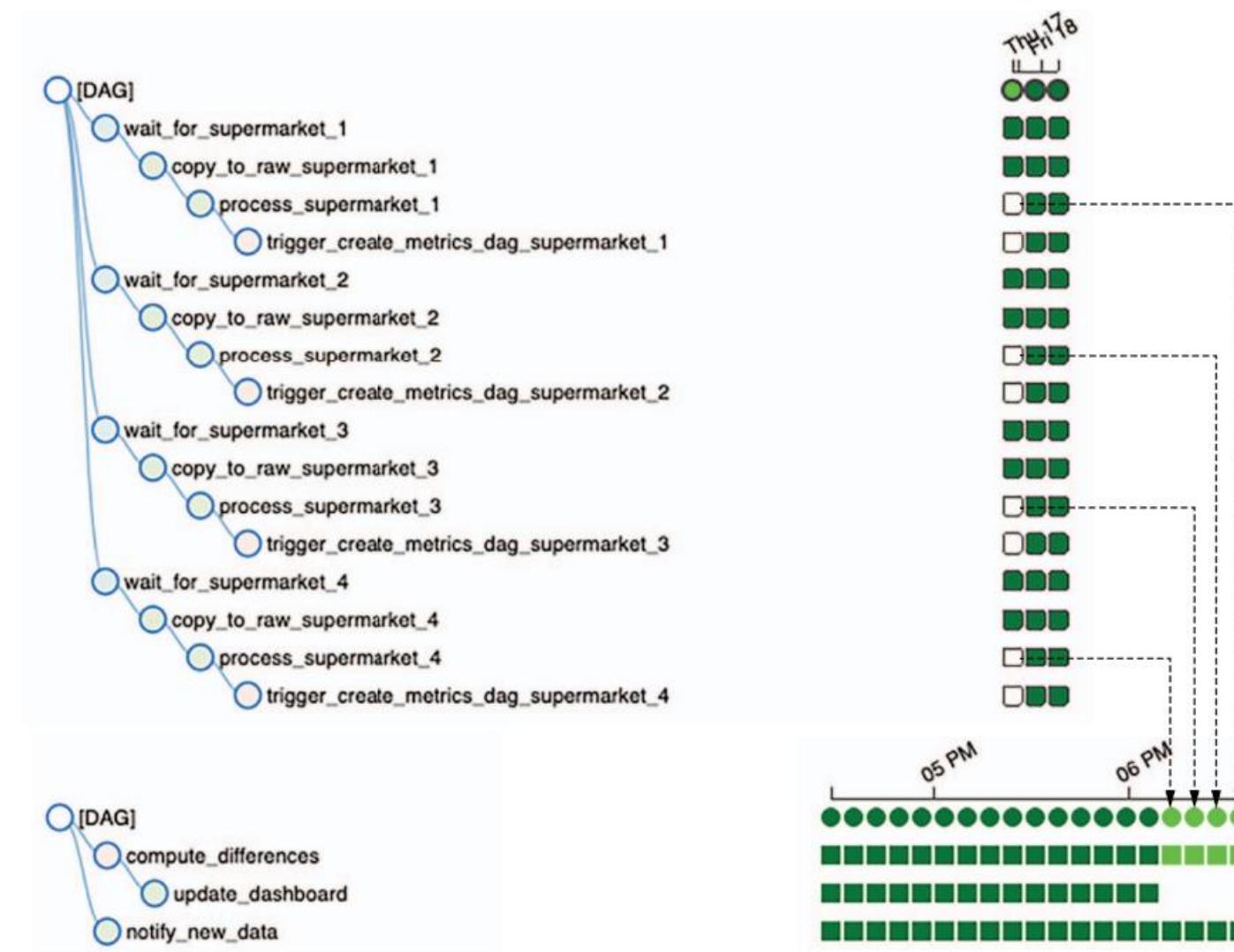
De plus, un DAG déclenchant plusieurs DAG aval est un scénario possible avec **TriggerDagRunOperator**, comme on le voit dans la figure à droite.



POLLING THE STATE OF OTHER DAG

Mais que se passe-t-il si plusieurs DAG déclencheurs doivent être terminés avant qu'un autre DAG puisse commencer à s'exécuter ?

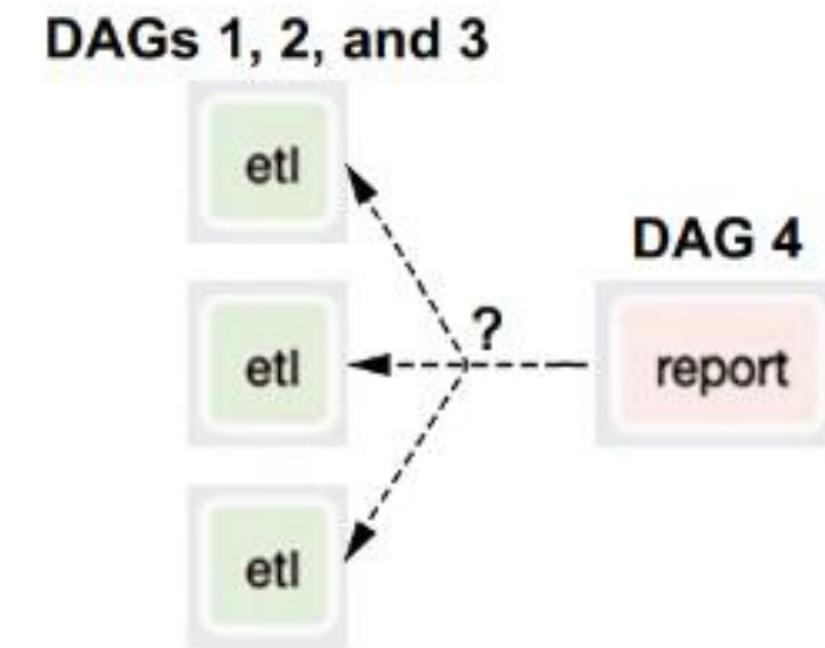
Par exemple, que se passe-t-il si les DAG 1, 2 et 3 extraient, transforment et chargent un ensemble de données, et que vous souhaitez exécuter le DAG 4 une fois que les trois DAG ont été terminés, par exemple pour calculer un ensemble de métriques agrégées ? Airflow gère les dépendances entre les tâches au sein d'un seul DAG ; cependant, il ne fournit pas de mécanisme pour les dépendances inter-DAG





POLLING THE STATE OF OTHER DAG

Pour cette situation, nous pourrions utiliser l'**ExternalTaskSensor**, qui est un capteur qui vérifie l'état des tâches dans d'autres DAG, comme le montre la figure 6.19. Ainsi, les tâches `wait_for_etl_dag{1,2,3}` agissent comme un proxy pour garantir l'état "terminé" des trois DAG avant d'exécuter finalement la tâche de rapport.



Le fonctionnement de l'**ExternalTaskSensor** consiste à le pointer vers une tâche dans une autre DAG pour vérifier son état.

```

import airflow.utils.dates
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from airflow.sensors.external_task import ExternalTaskSensor

dag1 = DAG(dag_id="ingest_supermarket_data", schedule_interval="0 16 * * *", ...)
dag2 = DAG(schedule_interval="0 16 * * *", ...)

DummyOperator(task_id="copy_to_raw", dag=dag1) >> DummyOperator(task_id="process_supermarket", dag=dag1)
wait = ExternalTaskSensor(
    task_id="wait_for_process_supermarket",
    external_dag_id="ingest_supermarket_data",
    external_task_id="process_supermarket",
    dag=dag2,
)
report = DummyOperator(task_id="report", dag=dag2)
wait >> report

```





POLLING THE STATE OF OTHER DAG

- Puisqu'il n'y a pas d'événement provenant de DAG 1 vers DAG 2, DAG 2 interroge l'état d'une tâche dans DAG 1, mais cela présente plusieurs inconvénients. Dans le monde d'Airflow, les DAG n'ont pas de notion d'autres DAG.
- Bien qu'il soit techniquement possible de interroger le stockage de métadonnées sous-jacent (ce que fait l'**ExternalTaskSensor**), ou de lire les scripts DAG du disque et d'inférer les détails d'exécution d'autres workflows, ils ne sont pas couplés de quelque manière que ce soit dans Airflow.
- Cela nécessite un peu d'alignement entre les DAG dans le cas où l'**ExternalTaskSensor** est utilisé. Le comportement par défaut est tel que l'**ExternalTaskSensor** vérifie simplement l'état d'une tâche avec la même date d'exécution que lui-même.



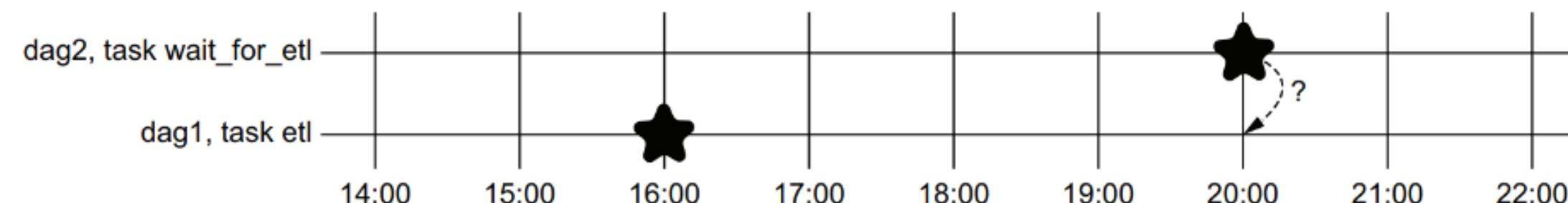
POLLING THE STATE OF OTHER DAG

- Ainsi, si un **ExternalTaskSensor** s'exécute avec une date d'exécution de 2019-10-12T18:00:00, il interrogerait le stockage de métadonnées d'Airflow pour la tâche donnée, également avec une date d'exécution de 2019-10-12T18:00:00.
- Maintenant, supposons que les deux DAG ont un intervalle de planification différent; alors ceux-ci ne s'aligneraient pas et donc l'**ExternalTaskSensor** ne trouverait jamais la tâche correspondante !

```
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from airflow.sensors.external_task import ExternalTaskSensor

dag1 = DAG(dag_id="dag1", schedule_interval="0 16 * * *")
dag2 = DAG(dag_id="dag2", schedule_interval="0 20 * * *") |-----|
DummyOperator(task_id="etl", dag=dag1)
ExternalTaskSensor(task_id="wait_for_etl", external_dag_id="dag1", external_task_id="etl", dag=dag2)
```

**schedule_intervals do not align,
thus ExternalTaskSensor will
never find corresponding task.**





POLLING THE STATE OF OTHER DAG

- En cas de non-alignement des intervalles de planification, nous pouvons effectuer un décalage, par lequel l'**ExternalTaskSensor** doit rechercher la tâche dans l'autre DAG.
- Ce décalage est contrôlé par l'argument `execution_delta` de l'**ExternalTaskSensor**. Il attend un objet `timedelta` et il est important de savoir qu'il fonctionne de manière contre-intuitive par rapport à ce que l'on attend.
- Le `timedelta` donné est soustrait de la date d'exécution, ce qui signifie qu'un `timedelta` positif regarde en arrière dans le temps.



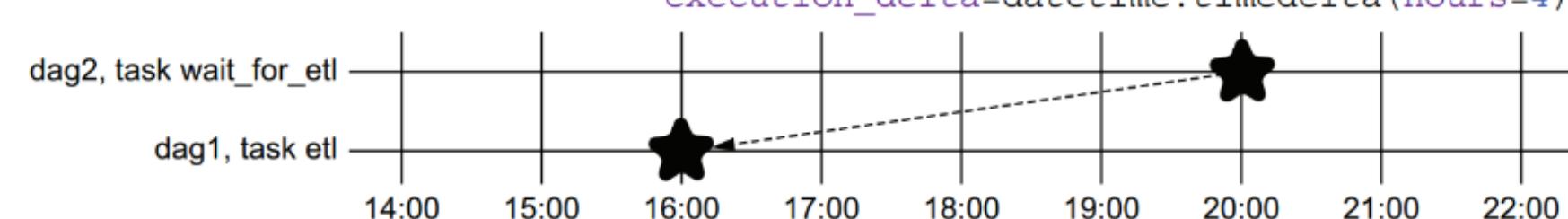
POLLING THE STATE OF OTHER DAG

- Notez que vérifier une tâche à l'aide de l'**ExternalTaskSensor** lorsque l'autre DAG a une période d'intervalle différente, par exemple, le DAG 1 s'exécute une fois par jour et le DAG 2 s'exécute toutes les cinq heures, complique le réglage d'une bonne valeur pour `execution_delta`. Pour ce cas d'utilisation, il est possible de fournir une fonction retournant une liste de `timedeltas` via l'argument `execution_date_fn`. Référez-vous à la documentation d'Airflow pour plus de détails.

```

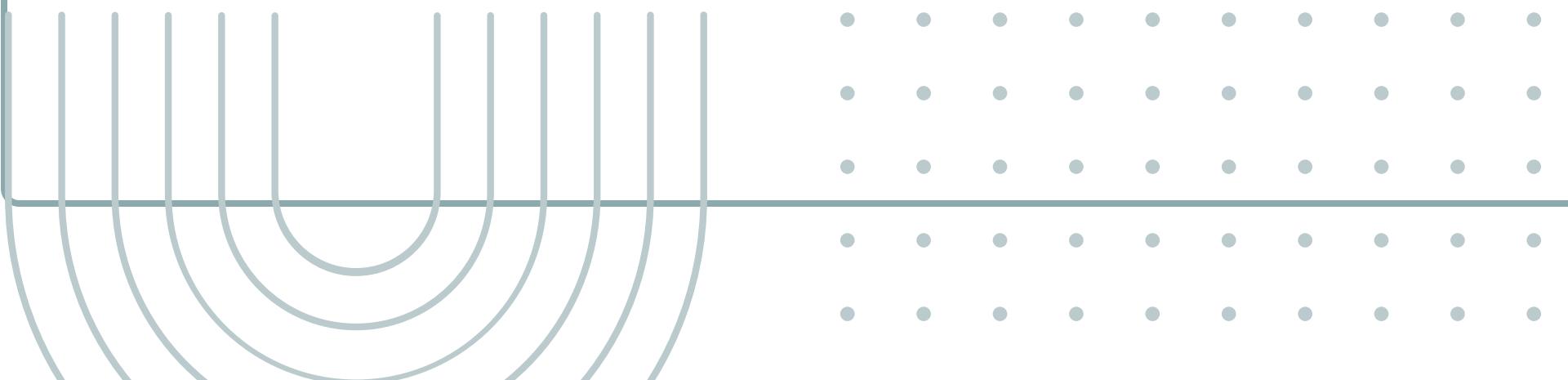
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from airflow.sensors.external_task import ExternalTaskSensor
dag1 = DAG(dag_id="dag1", schedule_interval="0 16 * * *")
dag2 = DAG(dag_id="dag2", schedule_interval="0 20 * * *")
DummyOperator(task_id="etl", dag=dag1)
ExternalTaskSensor(
    task_id="wait_for_etl",
    external_dag_id="dag1",
    external_task_id="etl",
    execution_delta=datetime.timedelta(hours=4),
    dag=dag2,
)

```



17.

STARTING WORKFLOWS WITH REST/CLI





POLLING THE STATE OF OTHER DAG

En plus de déclencher des DAGs à partir d'autres DAGs, ils peuvent également être déclenchés via l'API REST et la CLI. Cela peut être utile si vous souhaitez démarrer des workflows en dehors d'Airflow (par exemple, dans le cadre d'un pipeline CI/CD). Ou, les données arrivant à des moments aléatoires dans un compartiment AWS S3 peuvent être traitées en définissant une fonction Lambda pour appeler l'API REST, déclenchant ainsi un DAG, au lieu de devoir exécuter des capteurs en continu. En utilisant la CLI Airflow, nous pouvons déclencher un DAG comme suit.

- **Déclencher un DAG en utilisant la CLI d'Airflow**

Cela déclenche la DAG1 avec la date d'exécution définie sur la date et l'heure actuelles. L'identifiant de l'exécution de la DAG est prefixé par "manual__" pour indiquer qu'elle a été déclenchée manuellement, ou depuis l'extérieur d'Airflow. La CLI accepte des configurations supplémentaires pour la DAG déclenchée.

```
airflow dags trigger dag1
→ [2019-10-06 14:48:54,297] {cli.py:238} INFO - Created <DagRun dag1 @ 2019-10-06 14:48:54+00:00: manual__2019-10-06T14:48:54+00:00, externally triggered: True>
```



POLLING THE STATE OF OTHER DAG

- Déclencher un DAG avec une configuration supplémentaire

```
airflow dags trigger -c '{"supermarket_id": 1}' dag1
airflow dags trigger --conf '{"supermarket_id": 1}' dag1
```

Cette configuration est ensuite disponible dans toutes les tâches du DAG déclenché via les variables de contexte de tâche.

- Appliquer une configuration à l'exécution de DAG

Ces tâches impriment la configuration fournie à l'exécution

du DAG, qui peut être utilisée comme une variable dans

toutes les tâches:

```
{cli.py:516} INFO - Running <TaskInstance: print_dag_run_conf.process 2019-10-15T20:01:57+00:00 [running]> on host ebd4ad13bf98
{logging_mixin.py:95} INFO - {'supermarket': 1}
{python_operator.py:114} INFO - Done. Returned value was: None
{logging_mixin.py:95} INFO - [2019-10-15 20:03:09,149]
{local_task_job.py:105} INFO - Task exited with return code 0
```

```
import airflow.utils.dates
from airflow import DAG
from airflow.operators.python import PythonOperator

dag = DAG(
    dag_id="print_dag_run_conf",
    start_date=airflow.utils.dates.days_ago(3),
    schedule_interval=None,
)

def print_conf(**context):
    print(context["dag_run"].conf)

process = PythonOperator(
    task_id="process",
    python_callable=print_conf,
    dag=dag,
```

Configuration supplied when triggering DAGs is accessible in the task context



POLLING THE STATE OF OTHER DAG

En conséquence, si vous avez un DAG dans lequel vous exécutez des copies de tâches simplement pour prendre en charge différentes variables, cela devient beaucoup plus concis avec la conf de DAG en cours d'exécution, car elle vous permet d'insérer des variables dans le pipeline. Cependant, notez que le DAG de la liste 6.8 n'a pas d'intervalle de planification (c'est-à-dire qu'il ne s'exécute que lorsqu'il est déclenché). Si la logique de votre DAG repose sur une conf de DAG en cours d'exécution, il ne sera pas possible de l'exécuter selon un calendrier, car cela ne fournit pas de conf de DAG en cours d'exécution.



De même, il est également possible d'utiliser l'API REST pour le même résultat (par exemple, si vous n'avez pas accès à l'interface en ligne de commande mais que votre instance Airflow peut être atteinte via HTTP).



POLLING THE STATE OF OTHER DAG

- Déclencher un DAG en utilisant l'API REST d'Airflow

Cela peut être pratique pour déclencher des DAG depuis l'extérieur d'Airflow, par exemple depuis votre système CI/CD.

```
# URL is /api/v1
curl \
-u admin:admin \
-X POST \
"http://localhost:8080/api/v1/dags/print_dag_run_conf/dagRuns" \
-H "Content-Type: application/json" \
-d '{"conf": {}}
```

Sending a plaintext username/password is not desirable; consult the Airflow API authentication documentation for other authentication methods.

```
{
  "conf": {},
  "dag_id": "print_dag_run_conf",
  "dag_run_id": "manual_2020-12-19T18:31:39.097040+00:00",
  "end_date": null,
  "execution_date": "2020-12-19T18:31:39.097040+00:00",
  "external_trigger": true,
  "start_date": "2020-12-19T18:31:39.102408+00:00",
  "state": "running"
}
```

The endpoint requires a piece of data, even if no additional configuration is given.

```
curl \
-u admin:admin \
-X POST \
"http://localhost:8080/api/v1/dags/print_dag_run_conf/dagRuns" \
-H "Content-Type: application/json" \
-d '{"conf": {"supermarket": 1}}'
```

```
{
  "conf": {
    "supermarket": 1
  },
  "dag_id": "listing_6_08",
  "dag_run_id": "manual_2020-12-19T18:33:58.142200+00:00",
  "end_date": null,
  "execution_date": "2020-12-19T18:33:58.142200+00:00",
  "external_trigger": true,
  "start_date": "2020-12-19T18:33:58.153941+00:00",
  "state": "running"
}
```