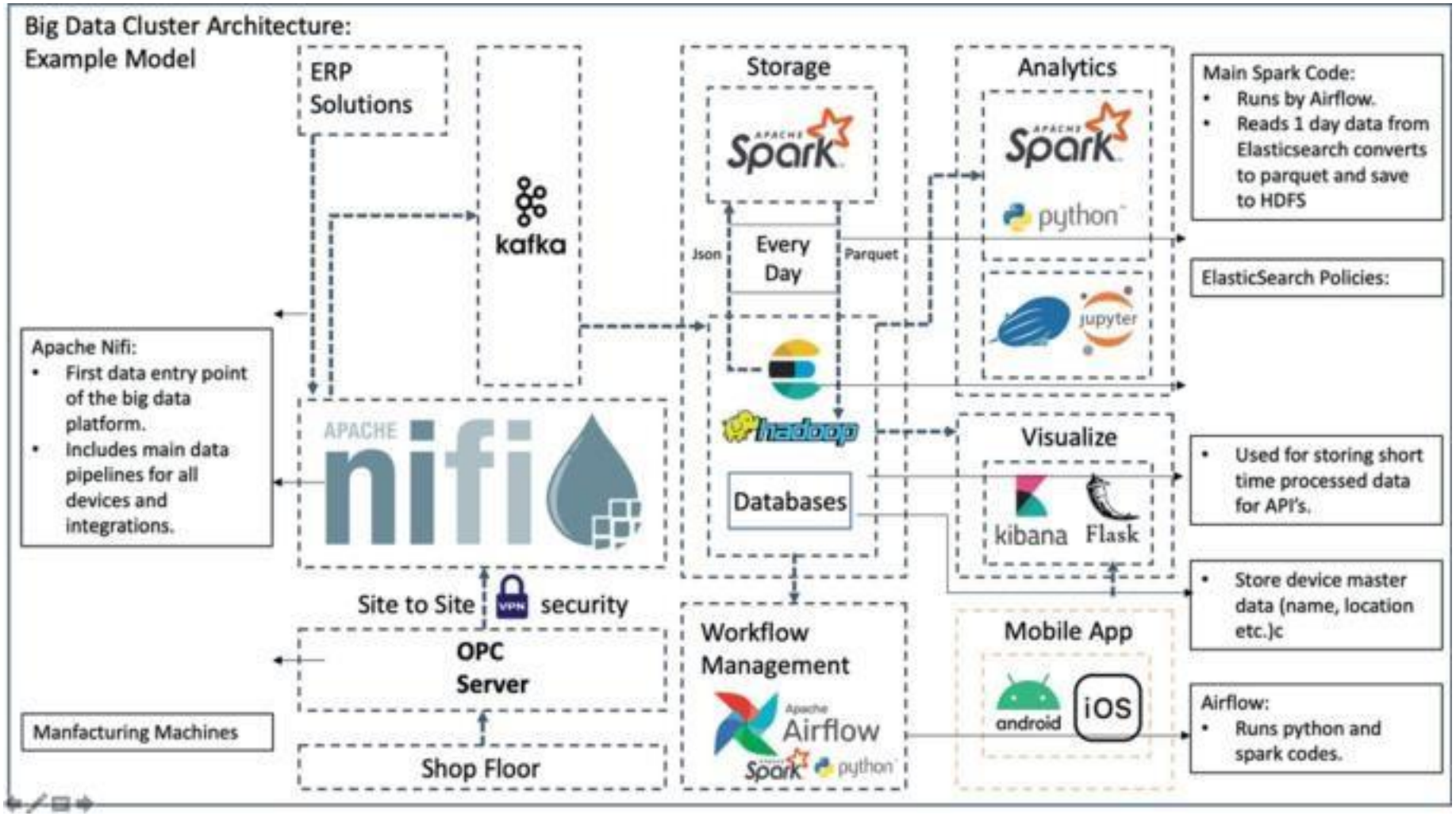
Chapitre 1

Airflow

Master 1

TECHNOLOGIES POUR CHAQUE ÉTAPE DE LA CHAINE DE VALEUR

Exemple d'Architecture Big Data



ORCHESTRATION DES WORKFLOWS

Importance de l'orchestration dans un contexte Big Data

- **Introduction :**
  Dans l'univers du Big Data, les organisations doivent traiter de grandes quantités de données provenant de sources variées comme des bases de données relationnelles, des fichiers log, des capteurs IoT, des services en ligne ou des réseaux sociaux. Ces données ne sont utiles que si elles sont correctement collectées, transformées et mises à disposition pour des analyses ultérieures.
  Cependant, la simple collecte et transformation des données ne suffisent pas. Chaque étape du pipeline Big Data — qu'il s'agisse de la collecte, du traitement ou du stockage — doit être parfaitement coordonnée pour éviter les erreurs et maximiser l'efficacité. Sans une orchestration adéquate, il devient très difficile de gérer ces flux de travail complexes à grande échelle.
- **Objectif de l'orchestration :**
  L'orchestration consiste à automatiser la gestion et la coordination de ces processus. Elle garantit que les tâches sont exécutées dans le bon ordre, de manière optimale, et qu'aucune ressource n'est gaspillée. Grâce à l'orchestration, on peut non seulement automatiser le traitement des données, mais aussi surveiller chaque étape, détecter les erreurs rapidement et reprendre les processus en cas d'échec.
- **Pourquoi est-ce essentiel ?**
  Les pipelines Big Data peuvent inclure des centaines de tâches interdépendantes. Si une seule tâche échoue, cela peut entraîner l'échec de l'ensemble du pipeline. L'orchestration offre la capacité de surveiller, relancer, et réajuster ces processus automatiquement, réduisant ainsi les risques d'erreurs humaines et permettant une exécution fluide à grande échelle.

ORCHESTRATION DES WORKFLOWS

Définition de l'orchestration et son rôle dans Big Data

- L'orchestration des workflows est le processus de gestion automatisée des tâches et des dépendances dans un pipeline. Cela signifie que toutes les étapes nécessaires au traitement des données, depuis la collecte jusqu'à l'analyse, sont organisées, automatisées et exécutées dans le bon ordre, sans intervention humaine.

  Dans un environnement Big Data, où les tâches sont complexes et interdépendantes, l'orchestration est essentielle pour garantir la fluidité et la cohérence des workflows.

- **Importance de l'orchestration dans Big Data :**
  Sans orchestration, les différentes étapes d'un pipeline doivent être gérées manuellement, ce qui est non seulement inefficace mais aussi sujet à des erreurs. L'orchestration assure que chaque tâche est exécutée automatiquement après l'accomplissement de la tâche précédente, en tenant compte des dépendances entre les tâches. Cela permet de gérer des pipelines de traitement de données de manière reproductible et à grande échelle.

  - **Exemple concret :**
    Imaginez un pipeline Big Data qui commence par la collecte de données à partir de capteurs IoT, passe par un traitement des données avec Apache Spark, puis charge les résultats dans une base de données Hive pour l'analyse.

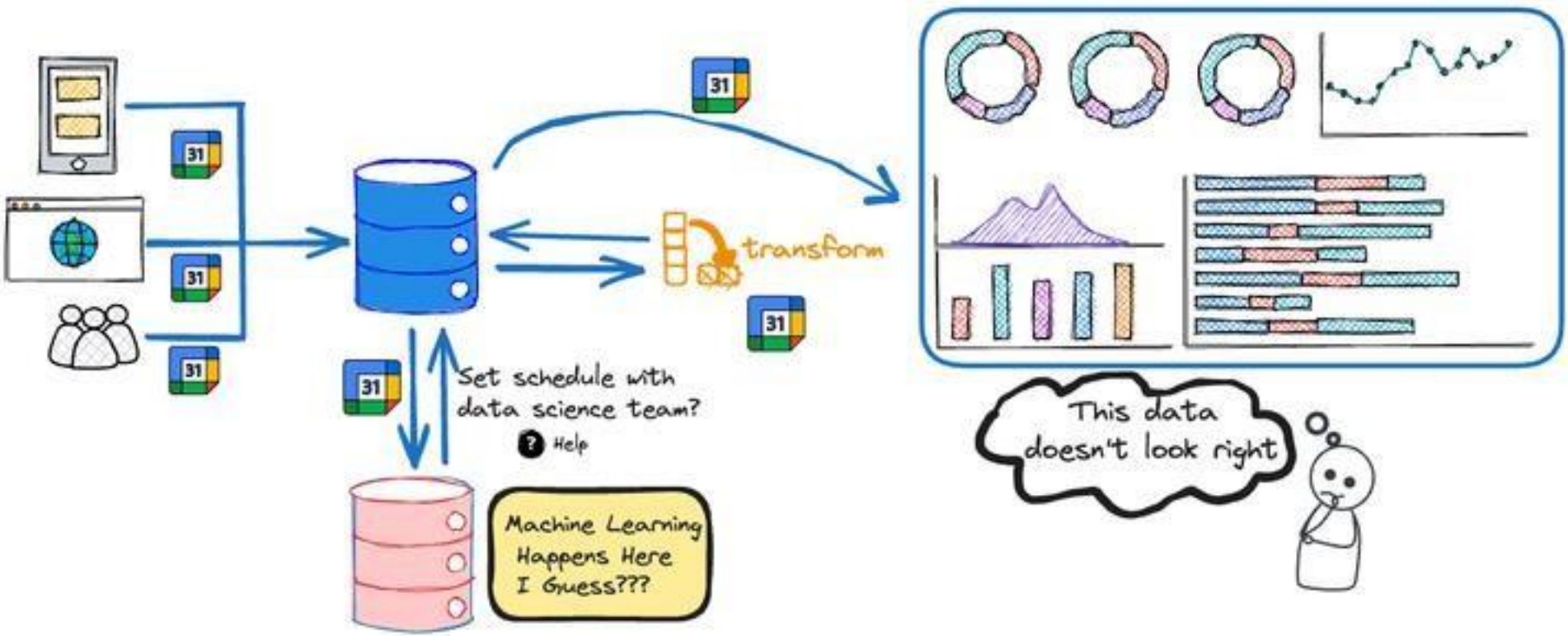
    Chaque étape du pipeline doit attendre la fin de l'étape précédente :

    - D'abord, les données sont collectées.
    - Ensuite, elles sont traitées par Spark pour être nettoyées et agrégées.
    - Enfin, les résultats sont chargés dans Hive pour être analysés ou visualisés.
      Sans orchestration, il faudrait surveiller et déclencher chaque étape manuellement. Avec l'orchestration, tout cela est automatisé.

- **Enjeux sans orchestration :**
  Sans une orchestration efficace, le risque d'erreurs humaines augmente. Des tâches peuvent être oubliées, exécutées dans le mauvais ordre ou simplement échouer sans être redémarrées. L'orchestration apporte également une dimension de surveillance continue, permettant de relancer automatiquement des tâches en cas d'échec, et ainsi de garantir l'intégrité des données traitées.

ORCHESTRATION DES WORKFLOWS

Définition de l'orchestration et son rôle dans Big Data



ORCHESTRATION DES WORKFLOWS

La complexité des pipelines Big Data (1/2)

- **Définition des pipelines Big Data :**
  Un pipeline Big Data est une séquence de processus ou de tâches qui manipule les données brutes pour les transformer en informations exploitables. Ces pipelines incluent généralement plusieurs étapes comme la collecte, l'ingestion, la transformation, le nettoyage, le stockage et l'analyse des données. Chacune de ces étapes peut impliquer des outils et des technologies distinctes, augmentant ainsi la complexité du pipeline.

- **Les différentes étapes d'un pipeline Big Data :**
  - **Collecte des données :** Les données proviennent de différentes sources : systèmes transactionnels, fichiers logs, API, capteurs IoT, etc.
  - **Ingestion des données :** Les données sont transférées dans une infrastructure Big Data (HDFS, S3, bases de données) pour un traitement ultérieur.
  - **Transformation et nettoyage :** Les données brutes sont transformées et nettoyées pour être prêtes à l'analyse (exemple : suppression des doublons, conversion des formats).
  - **Stockage des données :** Les données transformées sont stockées dans des systèmes comme Hive, HBase ou un Data Lake pour être accessibles par d'autres processus.
  - **Analyse des données :** Les données sont analysées via des outils comme Spark, Tableau, ou Superset pour en extraire des informations pertinentes.

ORCHESTRATION DES WORKFLOWS

La complexité des pipelines Big Data (2/2)

- **Les défis des pipelines complexes :**
  - **Multiples sources de données :** Les données peuvent provenir de diverses sources, dans des formats variés (CSV, JSON, XML, etc.).
  - **Dépendances entre les tâches :** Chaque tâche dans un pipeline dépend de la réussite des tâches précédentes. Si une tâche échoue, cela peut interrompre tout le processus.
  - **Gestion des erreurs :** En cas d'erreur, il est difficile de reprendre le processus à la bonne étape sans une orchestration efficace.
  - **Scalabilité :** Les pipelines Big Data doivent être capables de traiter d'énormes volumes de données tout en restant performants.

- **Importance de l'orchestration :**
  La complexité des pipelines Big Data rend indispensable l'utilisation d'un orchestrateur comme Apache Airflow, qui permet de gérer et de surveiller l'exécution de ces workflows. L'orchestration garantit que toutes les tâches du pipeline sont exécutées de manière coordonnée et dans le bon ordre, même si les tâches sont réparties sur plusieurs machines ou clusters.

ORCHESTRATION DES WORKFLOWS

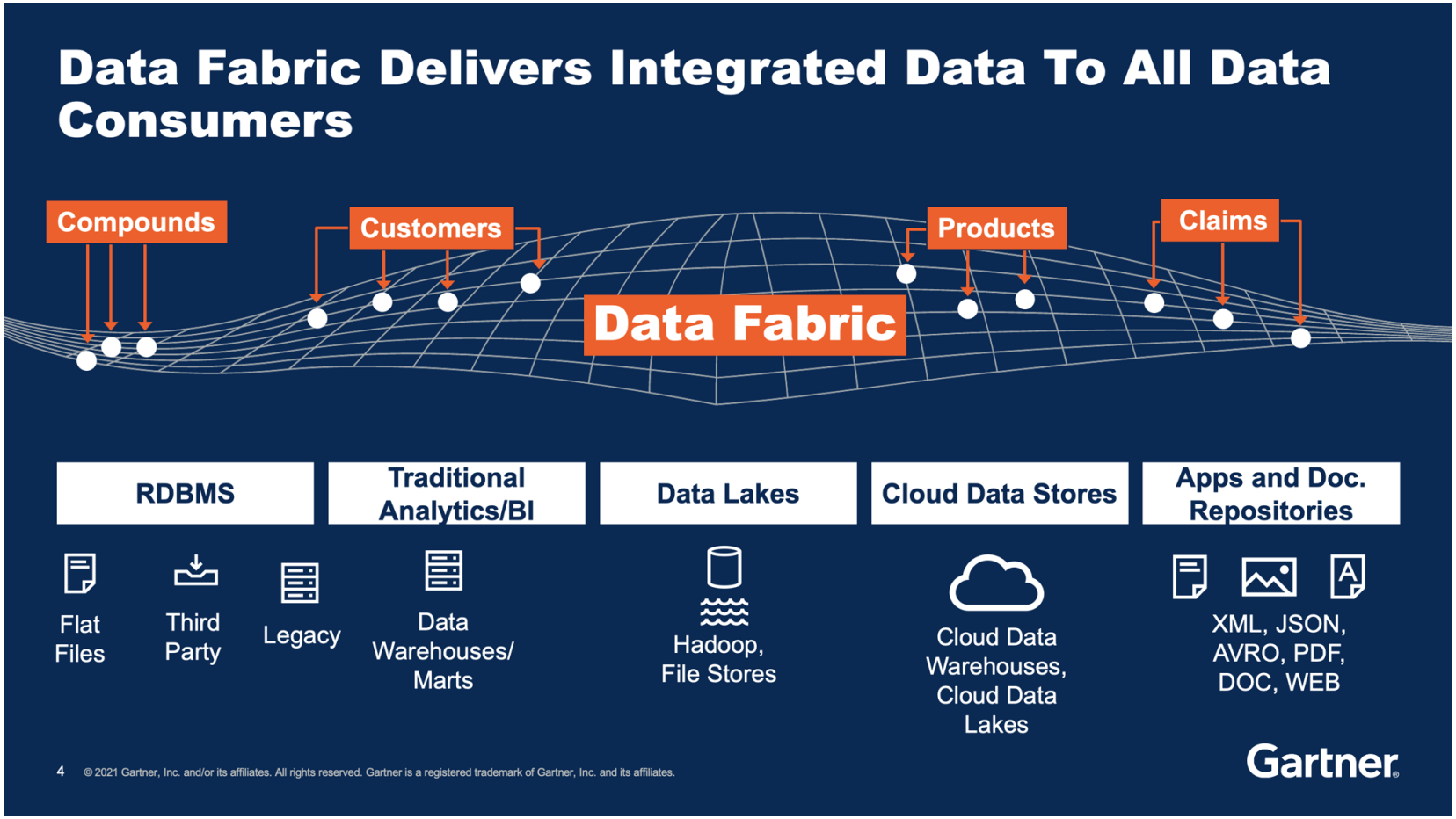
Multiplicité des sources et des outils dans un écosystème Big Data (1/3)

- **Diversité des sources de données dans les environnements Big Data :**
  Les données peuvent provenir de nombreuses sources hétérogènes. Cette diversité rend les pipelines Big Data complexes, car chaque source peut nécessiter une approche différente pour la collecte, la transformation et l'intégration des données. Voici quelques exemples de sources de données courantes :
  - **Bases de données relationnelles (SQL)** : Oracle, MySQL, PostgreSQL, où les données sont stockées dans des tables structurées.
  - **Données semi-structurées** : Formats comme JSON, XML, CSV provenant d'API, de fichiers log ou de services web.
  - **Données non structurées** : Images, vidéos, documents texte stockés dans des systèmes de fichiers distribués comme HDFS ou S3.
  - **Données en temps réel** : Flux de données provenant de capteurs IoT, de systèmes de surveillance en temps réel, ou d'outils comme Apache Kafka et Apache Flink.

- Ces sources de données présentent chacune leurs propres défis, notamment en termes de compatibilité de format, de fréquence de mise à jour et de volume de données.

ORCHESTRATION DES WORKFLOWS

Multiplicité des sources et des outils dans un écosystème Big Data (1/3)



ORCHESTRATION DES WORKFLOWS

Multiplicité des sources et des outils dans un écosystème Big Data (2/3)

- Chaque type de donnée nécessite une méthode spécifique pour être ingéré dans le pipeline. Par exemple, les données en temps réel doivent être traitées de manière asynchrone, tandis que les bases de données SQL peuvent être interrogées périodiquement pour synchroniser les informations.

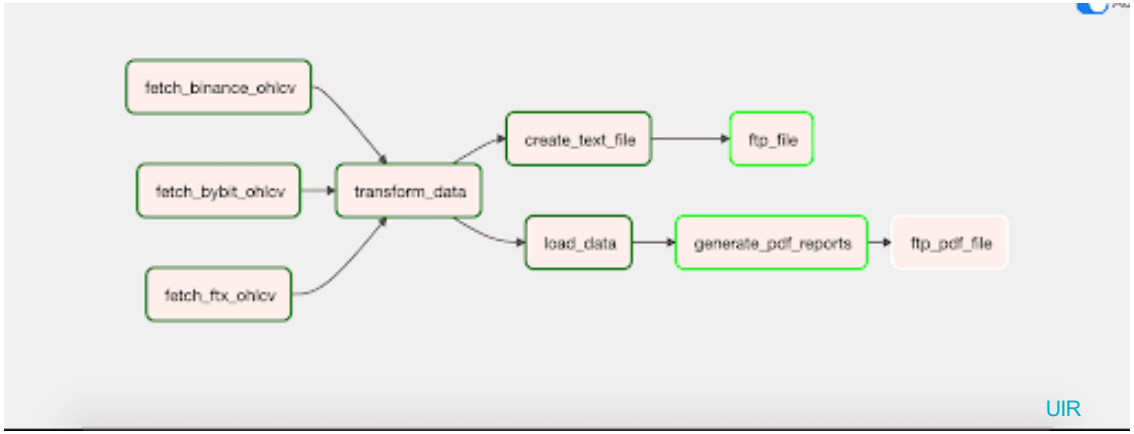
- **Problèmes causés par cette multiplicité de sources :**
  - **Complexité accrue de la gestion des données** : La collecte, la transformation et l'intégration de données provenant de diverses sources augmentent le nombre d'étapes et de dépendances dans le pipeline.
  - **Formats de données variés** : Les données issues de fichiers log (CSV, JSON) ne sont pas aussi facilement manipulables que celles provenant de bases de données relationnelles. Cela requiert des étapes supplémentaires de transformation.
  - **Systèmes hétérogènes** : Chaque source utilise des technologies différentes, ce qui rend l'intégration plus complexe. Par exemple, les données de capteurs IoT peuvent être stockées dans un système temps réel comme Kafka, tandis que les données historiques sont dans une base de données relationnelle.

ORCHESTRATION DES WORKFLOWS

Multiplicité des sources et des outils dans un écosystème Big Data (3/3)

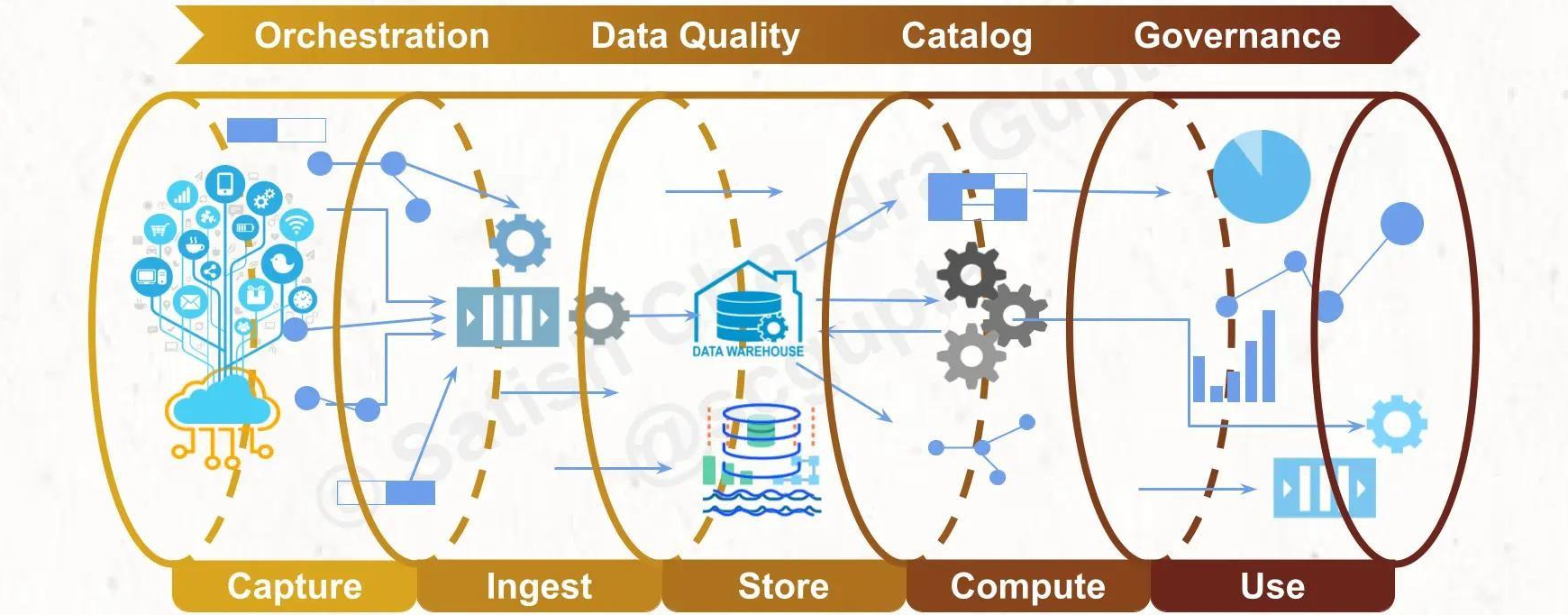
- **Importance de l'orchestration dans ce contexte :**
  L'orchestration permet de centraliser la gestion de ces différentes sources de données et d'automatiser les tâches nécessaires pour les intégrer dans un pipeline unifié. Grâce à des outils comme Airflow, il est possible de créer un workflow qui prend en charge l'ingestion de multiples sources, de gérer les transformations et de garantir que toutes les étapes sont exécutées dans le bon ordre et sans erreur.

- **Exemple avec Apache Airflow :** Un workflow Airflow pourrait être configuré pour :
  - Interroger une base de données SQL toutes les heures.
  - Collecter des flux de données en temps réel depuis Kafka.
  - Nettoyer et transformer les données issues des fichiers logs pour les rendre exploitables.
  - Enregistrer les données nettoyées dans un Data Lake ou une base de données NoSQL comme MongoDB ou HBase.



ORCHESTRATION DES WORKFLOWS

Illustration d'un pipeline Big Data typique



ORCHESTRATION DES WORKFLOWS

Automatisation et réduction des erreurs humaines (1/2)

- **Pourquoi l'automatisation est essentielle dans les pipelines Big Data :**
  Dans un environnement Big Data, les pipelines peuvent contenir des centaines, voire des milliers de tâches interdépendantes. La gestion manuelle de ces processus est non seulement chronophage, mais aussi sujette aux erreurs humaines. L'automatisation permet de déclencher ces tâches de manière autonome, garantissant une exécution fluide et précise.
- **Exemple d'automatisation :**
  - Un pipeline Big Data peut être configuré pour collecter automatiquement des données provenant d'une API externe chaque jour à 2h du matin. Ces données sont ensuite automatiquement traitées par Apache Spark, puis chargées dans une base de données pour l'analyse.
  - Si ce pipeline n'était pas automatisé, il serait nécessaire de surveiller et de déclencher manuellement chaque étape, multipliant les risques d'erreurs et d'oublis.
- **Réduction des erreurs humaines grâce à l'automatisation :**
  L'automatisation permet de réduire considérablement les erreurs liées à l'intervention humaine. Par exemple :
  - **Oubli d'une tâche :** Dans un workflow manuel, il est facile d'oublier d'exécuter une tâche ou d'exécuter une tâche dans le mauvais ordre. L'orchestration automatisée assure que toutes les tâches s'exécutent dans l'ordre correct et selon le calendrier défini.
  - **Mauvaise configuration :** Lors de l'exécution manuelle de tâches répétitives, une mauvaise configuration ou une erreur dans les paramètres peut causer l'échec du pipeline. L'orchestration permet de définir des configurations stables et de les réutiliser, minimisant ainsi les risques d'erreurs.
- **Exemple :**
  - Dans un processus manuel, un opérateur pourrait accidentellement exécuter un script d'analyse avant que la collecte des données soit terminée, entraînant des résultats incomplets. Avec l'orchestration, cette erreur est évitée, car les tâches sont déclenchées automatiquement après la fin de l'étape précédente.

ORCHESTRATION DES WORKFLOWS

Automatisation et réduction des erreurs humaines (2/2)

- **Avantages de l'automatisation dans Big Data :**
  - **Gain de temps :** Les tâches sont exécutées automatiquement, sans besoin d'intervention humaine, ce qui libère du temps pour d'autres activités à plus forte valeur ajoutée.
  - **Consistance :** Les workflows sont exécutés de manière cohérente, à chaque fois, sans variation ou déviation, ce qui garantit des résultats fiables.
  - **Réduction des coûts :** L'automatisation permet de réduire les coûts opérationnels en évitant les erreurs coûteuses et en optimisant l'utilisation des ressources.

- **Outils d'automatisation pour les pipelines Big Data :**
  Apache Airflow est un excellent exemple d'un outil permettant d'orchestrer et d'automatiser des tâches complexes. Il permet de définir des workflows qui s'exécutent à des intervalles réguliers ou en fonction de conditions spécifiques, tout en offrant des mécanismes de reprise automatique en cas d'échec d'une tâche.

ORCHESTRATION DES WORKFLOWS

Les défis liés à la gestion des workflows complexes (gestion des dépendances)

- **Les dépendances entre les tâches :**
  Dans un pipeline Big Data, chaque tâche a souvent des dépendances sur d'autres tâches. Cela signifie qu'une tâche ne peut pas commencer tant que la précédente n'est pas terminée avec succès. Ces dépendances créent une complexité supplémentaire qui doit être gérée de manière rigoureuse.
  - **Exemple :** Une tâche de transformation de données (ex. : nettoyage des données avec Spark) ne peut commencer que lorsque les données ont été collectées et ingérées dans le système de stockage (ex. : HDFS ou un Data Lake).
  - Si cette séquence n'est pas respectée, cela peut entraîner des erreurs de traitement, comme des résultats incomplets ou incohérents.

- **Synchronisation des tâches :**
  Lorsque plusieurs tâches doivent s'exécuter en parallèle ou dans un ordre spécifique, la gestion manuelle devient difficile. Il est crucial de synchroniser les tâches pour garantir que toutes les étapes sont correctement alignées et exécutées selon les priorités établies.
  - **Exemple :** Dans un pipeline où des données doivent être collectées à partir de différentes sources (fichiers CSV, API externes), il faut s'assurer que toutes les données sont bien disponibles avant de lancer la transformation.

ORCHESTRATION DES WORKFLOWS

Les défis liés à la gestion des workflows complexes (gestion des dépendances)

- **Gestion des erreurs et des échecs :**
  Les pipelines Big Data peuvent échouer pour diverses raisons, comme des erreurs réseau, une surcharge de ressources, ou des problèmes de compatibilité de données. La gestion des erreurs est donc essentielle pour garantir que les workflows puissent être relancés sans compromettre l'intégrité des données.
  - **Stratégie de gestion des échecs :** Un pipeline bien conçu doit inclure des mécanismes de reprise automatique pour les tâches échouées. Par exemple, Apache Airflow permet de définir des "retries" (réessais) pour relancer une tâche en cas d'échec, avec un intervalle de temps spécifié entre les tentatives.

- **Problème de scalabilité :**
  Les pipelines Big Data doivent être capables de s'adapter à la croissance des volumes de données. Plus les volumes de données augmentent, plus il devient difficile de gérer manuellement les tâches. Une orchestration efficace permet de gérer ces défis en automatisant les processus et en garantissant une scalabilité à travers l'utilisation de clusters et de ressources distribuées.
  - **Exemple :** Si le volume de données collectées augmente brusquement, un pipeline orchestré peut adapter dynamiquement les ressources de calcul pour traiter ces données sans intervention humaine.

- **Orchestration pour résoudre ces défis :**
  L'utilisation d'un outil d'orchestration comme Airflow permet de gérer automatiquement les dépendances, de synchroniser les tâches, et d'automatiser la gestion des erreurs. Ces outils offrent une interface graphique pour visualiser les workflows et permettent de surveiller l'état d'avancement des tâches en temps réel.
  - **Exemple pratique :** Avec Apache Airflow, il est possible de définir des "DAGs" (Directed Acyclic Graphs) pour orchestrer des workflows complexes avec plusieurs dépendances et conditions. Chaque tâche peut être configurée avec des règles spécifiques pour savoir quand elle doit commencer, en fonction de l'état des autres tâches.

ORCHESTRATION DES WORKFLOWS

Scalabilité et performance dans les environnements Big Data

- **Définition de la scalabilité dans un contexte Big Data :**
  La scalabilité fait référence à la capacité d'un système à gérer une augmentation du volume de données ou du nombre de tâches sans compromettre les performances. Dans un pipeline Big Data, les données peuvent croître de manière exponentielle, et le système doit être capable d'évoluer pour traiter ces volumes supplémentaires de manière efficace.

- **Types de scalabilité :**
  - **Scalabilité horizontale :** Ajouter plus de nœuds dans un cluster pour partager la charge de travail.
  - **Scalabilité verticale :** Ajouter plus de puissance (CPU, RAM) à un nœud unique pour traiter des tâches plus rapidement.

- **Défis de la scalabilité dans Big Data :**
  Les environnements Big Data sont souvent distribués sur plusieurs systèmes, chacun ayant des ressources limitées. Le traitement d'énormes volumes de données requiert une orchestration efficace pour s'assurer que les tâches sont réparties équitablement entre les différents nœuds de calcul et que les ressources disponibles sont utilisées de manière optimale.

- **Exemple concret :**
  - Imaginons un pipeline qui collecte des données en temps réel depuis des milliers de capteurs IoT dans une usine. Si la quantité de données augmente rapidement (par exemple, en raison de l'ajout de nouveaux capteurs), le pipeline doit être capable de s'adapter en ajoutant de nouveaux nœuds au cluster ou en redimensionnant les ressources de calcul.
- Sans orchestration, il serait très difficile de gérer cette augmentation de charge de manière manuelle.

ORCHESTRATION DES WORKFLOWS

Scalabilité et performance dans les environnements Big Data

- **Optimisation des performances grâce à l'orchestration :**
  L'orchestration permet d'optimiser les performances en distribuant les tâches sur plusieurs nœuds, en gérant les ressources de manière intelligente et en veillant à ce que chaque tâche s'exécute avec les ressources nécessaires.
  - **Répartition des tâches :** Dans un environnement distribué, un orchestrateur comme Apache Airflow peut diviser les tâches de traitement des données en plusieurs sous-tâches et les attribuer à différents nœuds pour une exécution parallèle, réduisant ainsi le temps de traitement global.
  - **Gestion des ressources :** Les orchestrateurs peuvent également surveiller l'utilisation des ressources (CPU, mémoire) et ajuster les allocations de manière dynamique pour éviter les goulets d'étranglement ou les surcharges.

- **Exemple d'un pipeline scalable :**
  Un pipeline de traitement de données géospatiales doit ingérer des cartes satellites de plusieurs téraoctets. Grâce à l'orchestration avec Apache Airflow, les données sont découpées en petits segments, traitées en parallèle par des nœuds Spark, puis réassemblées pour une analyse globale. Ce type d'architecture scalable permet de traiter rapidement des volumes massifs de données tout en optimisant l'utilisation des ressources.

ORCHESTRATION DES WORKFLOWS

Scalabilité et performance dans les environnements Big Data

- **Avantages de la scalabilité et de l'optimisation :**
  - **Réduction du temps de traitement :** En exécutant les tâches en parallèle, les orchestrateurs permettent de réduire considérablement le temps nécessaire pour traiter de grands volumes de données.
  - **Meilleure utilisation des ressources :** L'orchestration permet de garantir que les ressources sont utilisées de manière optimale, réduisant ainsi les coûts associés au traitement des données.
  - **Adaptabilité :** Le pipeline peut évoluer en fonction des besoins sans qu'il soit nécessaire de réécrire le code ou de reconfigurer manuellement les systèmes.

- **Importance de l'orchestration pour la scalabilité :**
  Les orchestrateurs comme Apache Airflow jouent un rôle crucial dans la scalabilité des pipelines Big Data en permettant une gestion intelligente des tâches et des ressources. Cela assure que, même lorsque les volumes de données augmentent, le pipeline reste performant et les temps de traitement restent faibles.

ORCHESTRATION DES WORKFLOWS

Gestion des erreurs et reprise des workflows après échec

- **Gestion des erreurs :**
  Les erreurs dans un pipeline Big Data peuvent survenir pour des raisons variées (échec réseau, données corrompues, surcharge de ressources). Une bonne gestion des erreurs est cruciale pour éviter des interruptions prolongées et garantir la continuité du workflow.
- **Stratégies courantes :**
  - **Réessais automatiques :** Airflow permet de définir des tentatives de réexécution (retries) en cas d'échec, avec un intervalle de temps prédéfini.
  - **Gestion des délais (Timeouts) :** Limiter le temps d'exécution d'une tâche avant de la marquer comme échouée et d'envoyer une alerte.
  - **Alertes :** Notifications par email ou autres canaux pour informer rapidement en cas de problème.
- **Reprise après échec :**
  La reprise automatique permet de relancer un workflow là où il s'est arrêté sans devoir tout recommencer.
  - **Exemple :** Si une tâche Spark échoue, Airflow relance cette tâche uniquement après correction sans impacter les autres tâches du pipeline.
- **Avantages :**
  - Réduction des temps d'arrêt et amélioration de la fiabilité.
  - Alertes proactives pour agir rapidement en cas d'échec.

ORCHESTRATION DES WORKFLOWS

- **Résumé de l'orchestration :**
  L'orchestration permet d'automatiser, de synchroniser et de surveiller les tâches dans un pipeline Big Data complexe. Elle garantit que toutes les étapes sont exécutées de manière coordonnée et efficace, minimisant les erreurs humaines et optimisant l'utilisation des ressources.

- **Avantages clés :**
  - **Efficacité :** Réduction des erreurs et des temps d'exécution grâce à l'automatisation.
  - **Conformité et traçabilité :** Suivi détaillé des workflows et des données, essentiel dans des environnements régulés.
  - **Résilience :** Relance automatique des tâches en cas d'échec, garantissant la continuité des opérations.

- **Transition vers Airflow :**
  L'importance de l'orchestration dans le Big Data étant bien établie, Apache Airflow se présente comme une solution robuste pour implémenter ces concepts de manière automatisée et flexible.

INTRODUCTION À APACHE AIRFLOW

Présentation d'Apache Airflow

- **Introduction à Apache Airflow :**
  Apache Airflow est une plateforme open-source qui permet d'orchestrer des workflows complexes. Il aide à planifier, surveiller et automatiser des tâches dans un environnement distribué.

- **Caractéristiques principales :**
  - **DAGs (Directed Acyclic Graphs) :** Airflow structure les workflows en DAGs, où chaque tâche est un nœud et les dépendances entre les tâches sont représentées par des arcs.
  - **Flexibilité :** Airflow permet d'écrire des workflows en Python, facilitant ainsi l'intégration avec d'autres outils et technologies Big Data.
  - **Interface utilisateur :** Offre une vue d'ensemble des workflows, permet de suivre l'avancement, de gérer les erreurs et de relancer des tâches si nécessaire.

- **Pourquoi Airflow ?**
  - Airflow est conçu pour des pipelines dynamiques et évolutifs, avec la possibilité de gérer des tâches de manière distribuée sur plusieurs nœuds.

INTRODUCTION À APACHE AIRFLOW

Historique et rôle dans l'écosystème Big Data

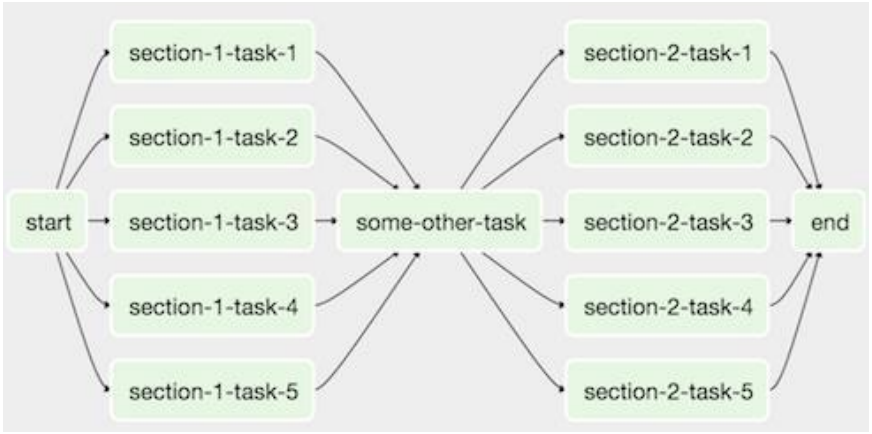
- **Historique d'Apache Airflow :**
  Développé à l'origine par Airbnb en 2014 pour gérer ses propres workflows complexes, Apache Airflow est rapidement devenu une solution populaire dans l'écosystème Big Data pour l'orchestration de pipelines.

- **Son rôle dans Big Data :**
  - Airflow aide à automatiser et organiser les processus Big Data complexes, tels que la collecte de données, le traitement avec Spark ou Hive, et le stockage dans des Data Lakes.
  - Il permet de gérer et de suivre des workflows distribués à travers plusieurs environnements.

- **Adoption :**
  Utilisé par des entreprises comme Airbnb, Uber, et Dropbox, Airflow est devenu un standard de fait pour l'orchestration de pipelines dans des environnements Big Data à grande échelle

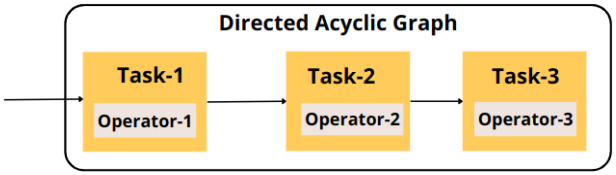
INTRODUCTION À APACHE AIRFLOW

Fonctionnement général d'Airflow



- **DAG (Directed Acyclic Graph) :**
  Le cœur d'Airflow est constitué de DAGs, des graphes orientés acycliques où chaque tâche représente une étape du pipeline, et les flèches représentent les dépendances entre ces tâches. Un DAG définit le flux de travail et le séquencement des tâches.

- **Opérateurs :**
  Airflow utilise des opérateurs pour définir les actions que chaque tâche doit effectuer. Il existe plusieurs types d'opérateurs comme :
  - **BashOperator** : Pour exécuter des scripts shell.
  - **PythonOperator** : Pour exécuter des fonctions Python.
  - **HiveOperator** : Pour interagir avec Hive, etc.



- **Planification et exécution des tâches :**
  Les workflows peuvent être planifiés pour s'exécuter à des intervalles réguliers ou déclenchés manuellement. Airflow permet aussi de surveiller et de redémarrer automatiquement les tâches en cas d'échec.

INTRODUCTION À APACHE AIRFLOW

Principaux composants d'Airflow (DAGs, tasks, opérateurs)

- **DAG (Directed Acyclic Graph)** :
  Le DAG est la structure qui définit l'ordre d'exécution des tâches sans boucles. Chaque nœud du graphe représente une tâche et les flèches indiquent les dépendances entre elles. Cela garantit que les étapes sont suivies dans le bon ordre.
  - **Exemple :** Un DAG peut représenter un workflow commençant par la collecte de données, suivi de leur transformation, et se terminant par leur stockage.
- **Opérateurs** :
  Les opérateurs définissent les actions que chaque tâche doit accomplir. Ils encapsulent des opérations spécifiques comme exécuter une commande bash (BashOperator), lancer une fonction Python (PythonOperator), ou soumettre un job Spark (SparkSubmitOperator).
  - **Exemple :** Un **HiveOperator** peut exécuter des requêtes SQL sur une base de données Hive pour traiter des données volumineuses.
- **Tasks (Tâches)** :
  Ce sont les unités de travail dans un DAG. Chaque opérateur dans Airflow est associé à une tâche spécifique.
  - **Exemple :** Un DAG peut comporter une tâche pour collecter des données d'une API, une autre pour les transformer, et une dernière pour les stocker.
- **XComs** :
  Permet l'échange d'informations entre les tâches dans un même DAG. Une tâche peut déposer des données dans XComs, et une autre tâche peut les récupérer pour les utiliser.
  - **Exemple :** Une tâche peut collecter des données, les passer à une autre tâche via XComs, qui les utilisera pour un traitement supplémentaire.
- **Monitoring et logs** :
  L'interface utilisateur d'Airflow permet de suivre l'exécution des tâches et d'afficher les logs pour chaque tâche. Cela permet de visualiser l'état des workflows (réussite, échec, etc.) et de diagnostiquer rapidement les problèmes.
  - **Exemple :** Un tableau de bord montrant les tâches échouées et les logs associés pour les déboguer.

INTRODUCTION À APACHE AIRFLOW

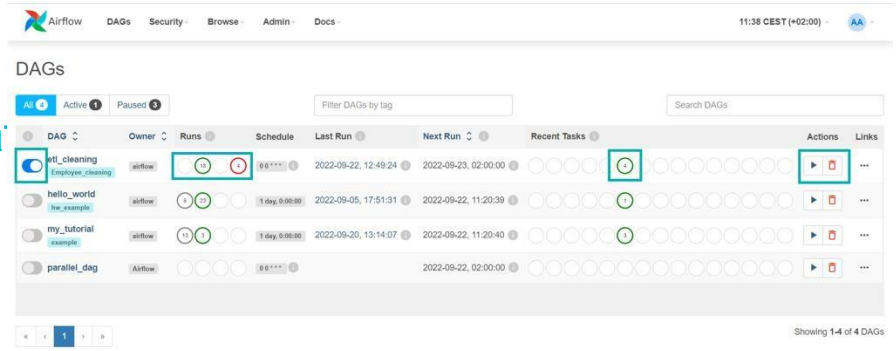
Avantages d'Airflow par rapport aux autres outils d'orchestration

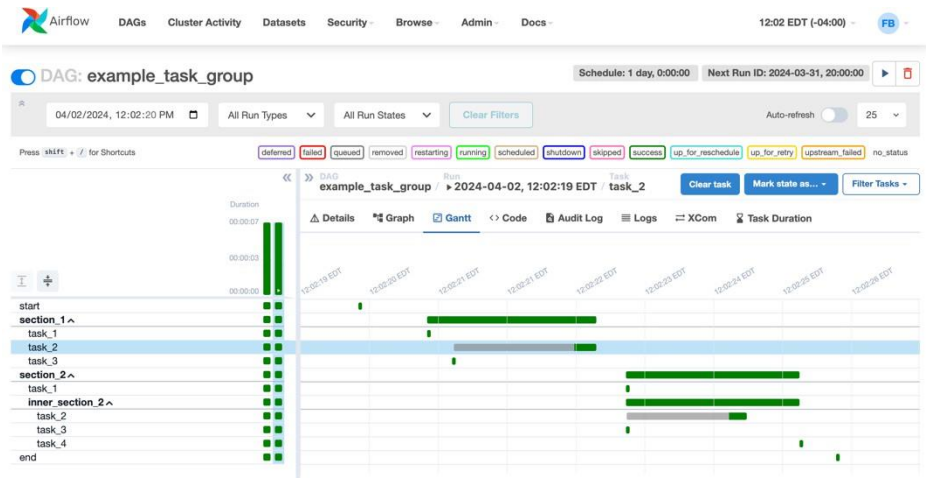
- **Flexibilité** :
  Airflow est écrit en Python, permettant de créer des workflows dynamiques et personnalisés avec une grande souplesse. Cette flexibilité permet d'intégrer Airflow avec pratiquement n'importe quel outil ou service.
  - **Exemple :** Vous pouvez intégrer des services cloud, des bases de données locales, ou des systèmes distribués comme Hadoop et Spark directement dans vos workflows.
- **Visualisation intuitive** :
  L'interface utilisateur permet de visualiser les DAGs, les tâches et leurs états (en attente, en cours, réussie, échouée) en temps réel. Cela simplifie la gestion et le suivi des workflows complexes.
  - **Exemple :** Visualiser un DAG complexe avec des dizaines de tâches exécutées à différents moments pour identifier facilement les points de blocage.
- **Modularité** :
  Airflow dispose de nombreux opérateurs prédéfinis, couvrant une grande variété d'actions comme l'exécution de commandes Bash, Python, Spark, Hive, etc. En plus, il est possible de créer des opérateurs personnalisés pour des besoins spécifiques.
  - **Exemple :** Développer un opérateur personnalisé pour interagir avec un service interne ou une API non supportée.
- **Gestion des échecs** :
  Airflow gère les erreurs via des stratégies comme les "retries" (réessais), les "timeouts" (temps d'exécution limite), et des alertes en cas de défaillance. Cela garantit une exécution plus robuste des workflows, même en cas d'erreurs.
  - **Exemple :** Si une tâche échoue, Airflow peut être configuré pour réessayer automatiquement jusqu'à trois fois avant d'alerter l'utilisateur.
- **Scalabilité** :
  Airflow est conçu pour gérer des pipelines complexes et volumineux. Il permet de répartir les tâches sur plusieurs nœuds dans un cluster, offrant ainsi une solution scalable pour traiter de grandes quantités de données.
  - **Exemple :** Un workflow Airflow avec des tâches distribuées sur un cluster de 10 nœuds Spark pour traiter des pétaoctets de données.

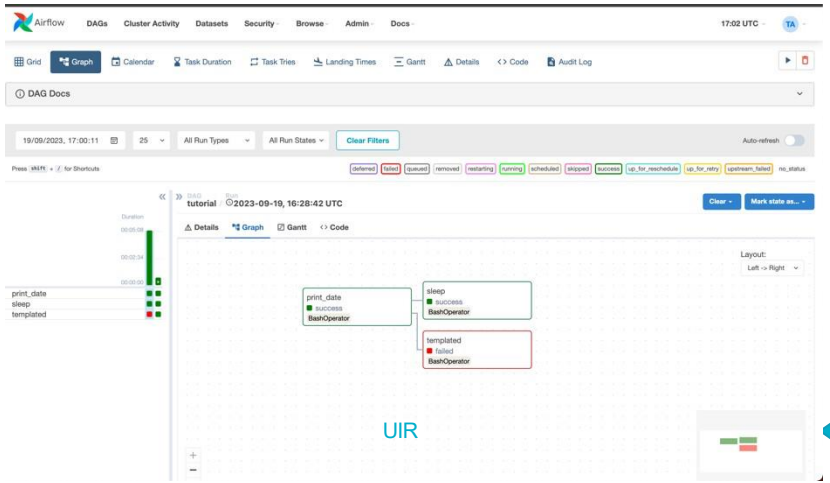
INTRODUCTION À APACHE AIRFLOW

Airflow UI - Introduction à l'interface utilisat

- **Présentation de l'interface utilisateur (UI) :**
  L'interface utilisateur d'Apache Airflow est l'un de ses principaux points forts. Elle permet de gérer et de superviser les DAGs, d'examiner les logs des tâches, de visualiser les flux de travail et de gérer les erreurs de manière intuitive. L'UI simplifie la gestion des workflows en offrant une vue d'ensemble des processus en cours et passés.
- **Tableau de bord des DAGs :**
  La page d'accueil de l'interface présente une liste de tous les DAGs disponibles. Pour chaque DAG, des informations essentielles sont affichées, telles que l'état actuel (en cours, réussi, échoué), le dernier temps d'exécution, et les intervalles de planification.
  - **Exemple :** Sur le tableau de bord, vous pouvez visualiser l'historique d'exécution d'un DAG avec des indicateurs de couleur (vert pour réussi, rouge pour échoué).
- **Vue de graphe :**
  Airflow propose une vue de graphe des DAGs, où vous pouvez visualiser graphiquement les tâches et leurs dépendances. Cela permet de comprendre facilement l'organisation des workflows et de suivre l'avancement de chaque tâche.
  - **Exemple :** Un DAG complexe peut être visualisé sous forme de graphe avec des liens clairs entre chaque étape du pipeline.







UIR

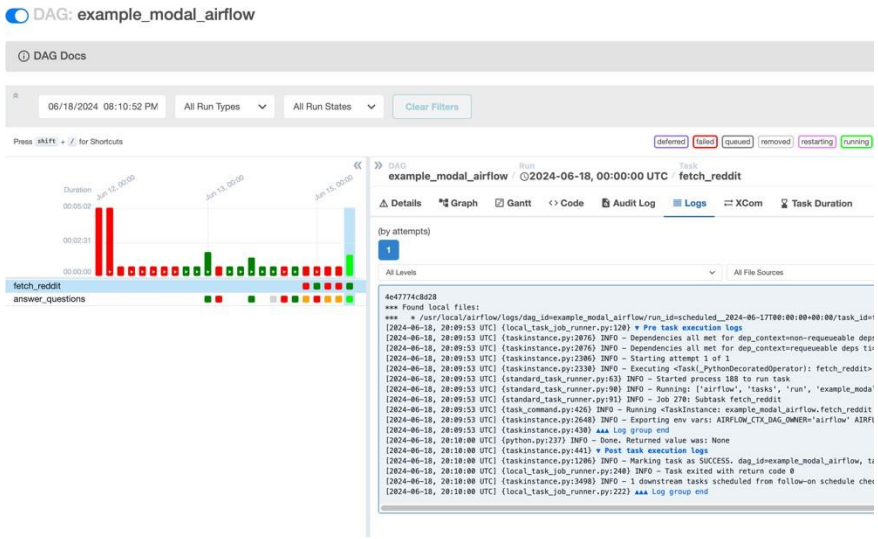
INTRODUCTION À APACHE AIRFLOW

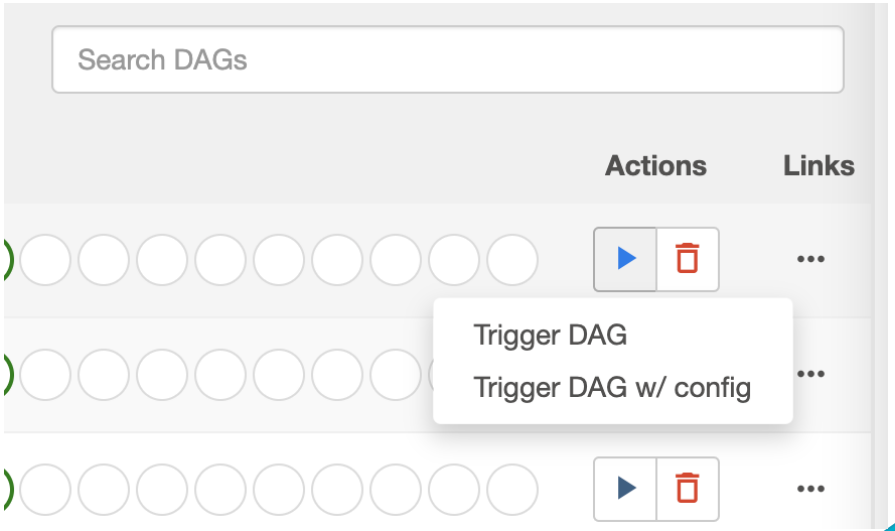
Airflow UI - Introduction à l'interface utilisateur

- **Logs et débogage :**
  L'interface permet également de consulter les logs détaillés de chaque tâche. Vous pouvez accéder aux logs des tâches échouées pour diagnostiquer les erreurs et prendre des mesures correctives.
  - **Exemple :** Si une tâche échoue, vous pouvez consulter les logs via l'UI pour voir la cause exacte (par exemple, un problème de connexion ou de mémoire).

- **Redémarrage des tâches et gestion des erreurs :**
  Depuis l'interface, vous pouvez relancer manuellement des tâches échouées ou suspendues. Vous pouvez également gérer les reprises automatiques ou déclencher des alertes pour les erreurs critiques.
  - **Exemple :** Si une tâche échoue lors d'une exécution programmée, l'UI permet de relancer cette tâche manuellement ou d'ajuster les paramètres d'exécution.

- **Planification des DAGs :**
  L'UI offre des outils pour gérer la planification des DAGs, en permettant de les exécuter immédiatement ou de modifier leur intervalle de planification sans toucher au code Python.
  - **Exemple :** Vous pouvez changer l'intervalle de planification d'un DAG directement dans l'UI, en passant d'une exécution quotidienne à une exécution hebdomadaire.





Exemples d'utilisation d'Airflow dans des environnements Big Data

- **Ingestion des données avec Apache NiFi et Airflow :**
  Dans un environnement Big Data, Airflow peut être utilisé pour orchestrer des pipelines d'ingestion de données en combinaison avec Apache NiFi, un outil dédié à l'ingestion des flux de données. Airflow peut gérer les dépendances et la planification tandis que NiFi assure la collecte des données depuis des sources variées (API, fichiers, bases de données).
  - **Exemple :** Utilisation d'Airflow pour déclencher des workflows NiFi qui collectent des données depuis plusieurs sources, telles que des capteurs IoT ou des API tierces, puis les transmettent vers un Data Lake pour traitement.

- **Traitement des données avec Spark et Airflow :**
  Apache Airflow est souvent utilisé pour orchestrer les tâches de traitement de données à grande échelle, en particulier avec Apache Spark. Après l'ingestion des données, Airflow peut planifier et surveiller des tâches Spark pour transformer, nettoyer et analyser les données.
  - **Exemple :** Un DAG Airflow peut soumettre un job Spark quotidien pour nettoyer et agréger les données stockées dans HDFS, en parallèle sur un cluster distribué, et sauvegarder les résultats dans Hive pour analyse.

Exemples d'utilisation d'Airflow dans des environnements Big Data

- **Chargement des données dans Hive avec Airflow :**
  Après le traitement des données, Airflow peut être utilisé pour orchestrer le chargement des données transformées dans une base de données distribuée comme Apache Hive.
  - **Exemple :** Un workflow Airflow qui exécute un **HiveOperator** pour charger les données traitées dans une table Hive. Cela permet aux analystes de les interroger avec SQL pour des analyses plus avancées.
- **Pipeline d'analyse avec Superset et Airflow :**
  Après le traitement des données, Airflow peut également orchestrer la visualisation de ces données dans des outils comme Apache Superset. En automatisant ces processus, les tableaux de bord et les rapports peuvent être mis à jour régulièrement sans intervention manuelle.
  - **Exemple :** Un DAG Airflow peut être configuré pour exécuter des jobs qui mettent à jour les sources de données dans Superset après chaque traitement des données, garantissant que les utilisateurs ont toujours accès à des visualisations à jour.
- **Automatisation des workflows de bout en bout :**
  Airflow permet de gérer de bout en bout l'ingestion, le traitement, le stockage et l'analyse des données. Il peut coordonner plusieurs outils Big Data (NiFi, Spark, Hive, Superset) dans un seul pipeline unifié, rendant les processus plus fluides et automatisés.
  - **Exemple :** Un pipeline Airflow qui collecte des données avec NiFi, les traite avec Spark, les stocke dans Hive, et les visualise dans Superset, offrant une solution complète pour la gestion des données.

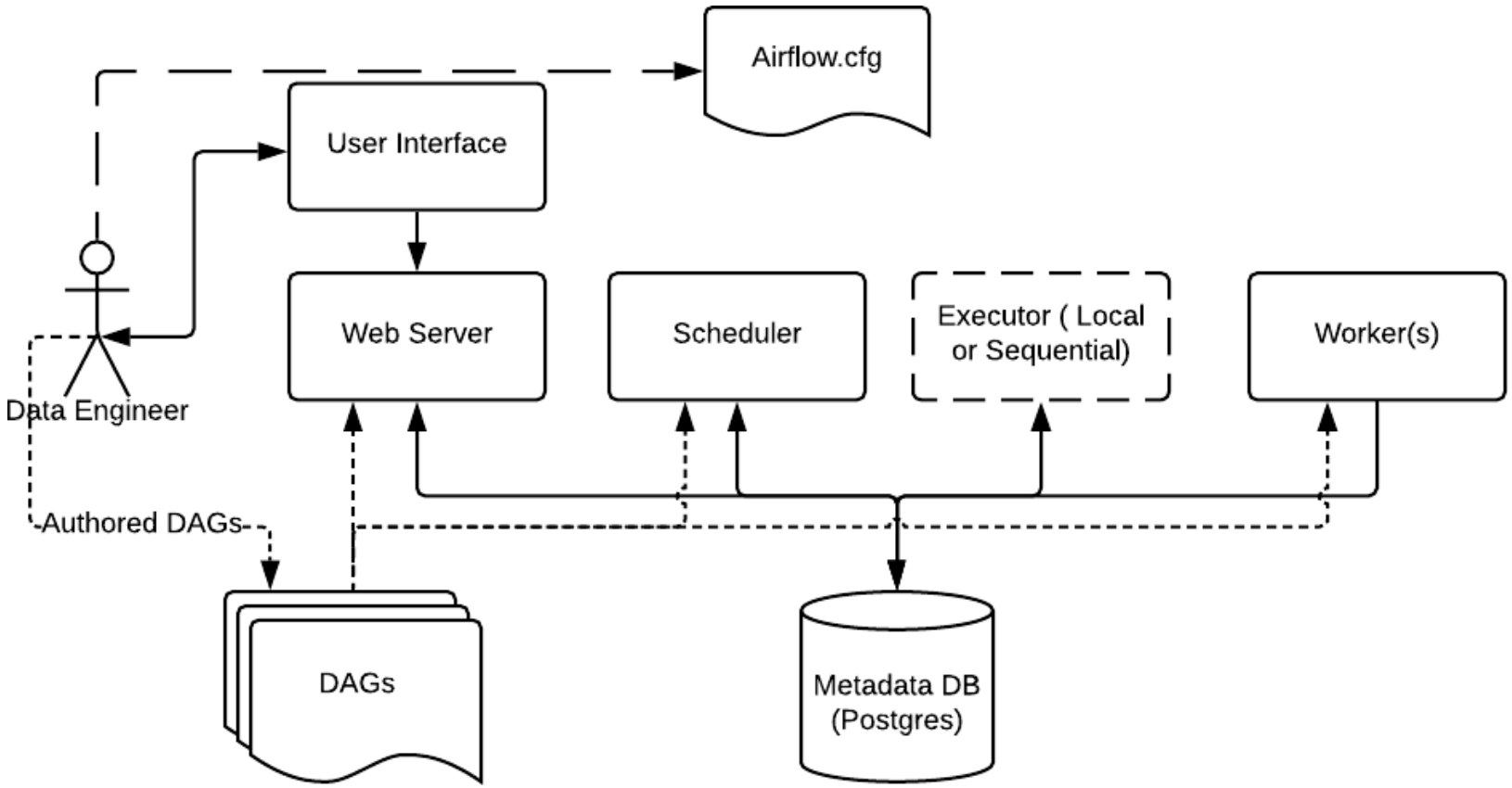
INTRODUCTION À APACHE AIRFLOW

Schéma d'architecture d'Airflow dans un cluster Big Data

- **Introduction à l'architecture d'Airflow :**
  Apache Airflow est conçu pour fonctionner dans des environnements distribués. Il peut être déployé dans un cluster Big Data pour orchestrer les workflows, en exploitant des systèmes distribués comme Hadoop, Spark, ou Hive pour gérer des volumes de données massifs. L'architecture d'Airflow comprend plusieurs composants clés : Scheduler, Web Server, Worker(s), et Metadata Database.
- **Composants principaux :**
  - **Scheduler (Planificateur)** : Responsable de la planification et de l'exécution des tâches définies dans les DAGs. Le Scheduler détermine quelles tâches sont prêtes à être exécutées en fonction des dépendances et des priorités.
  - **Web Server (Serveur Web)** : Fournit l'interface utilisateur d'Airflow, permettant aux utilisateurs de gérer les DAGs, de consulter les logs, de suivre les workflows et de redémarrer les tâches échouées.
  - **Workers (Travailleurs)** : Ce sont les processus qui exécutent les tâches dans le DAG. Dans un cluster, plusieurs workers peuvent être utilisés pour paralléliser l'exécution des tâches sur plusieurs nœuds.
  - **Metadata Database (Base de données de métadonnées)** : Contient les informations sur l'état des DAGs, des tâches, les logs, et les plannings. Cela permet au Scheduler de suivre l'état des workflows et d'ajuster l'exécution des tâches en conséquence.
- **Déploiement dans un cluster :**
  - **Exécution distribuée des tâches :** Les tâches dans un DAG peuvent être distribuées sur plusieurs nœuds pour une exécution parallèle, en utilisant un cluster Hadoop ou Spark pour le traitement des données. Cela permet à Airflow de gérer de grands volumes de données de manière scalable.
  - **Scalabilité** : Airflow peut être configuré pour ajouter ou retirer des workers en fonction de la charge de travail. Cela permet d'optimiser l'utilisation des ressources du cluster, garantissant que chaque tâche est exécutée de manière efficace.
- **Exemple d'architecture avec Airflow et Spark :**
  - **Airflow Scheduler** planifie les tâches et répartit les jobs sur différents workers.
  - Les **Workers** exécutent les tâches distribuées sur un cluster Spark pour traiter les données à grande échelle.
  - Les résultats sont ensuite stockés dans des systèmes de stockage distribués, comme HDFS ou Hive.
  - **Web Server** permet aux utilisateurs de suivre les workflows et de diagnostiquer les erreurs si nécessaire.
- **Avantages de l'architecture distribuée :**
  - **Parallélisme** : Les tâches peuvent être exécutées en parallèle sur plusieurs nœuds, réduisant ainsi le temps global de traitement.
  - **Résilience** : Si un worker échoue, le Scheduler peut redistribuer les tâches à un autre worker, assurant ainsi la continuité du pipeline.
  - **Monitoring centralisé** : Grâce à l'interface d'Airflow, les utilisateurs peuvent superviser l'ensemble des workflows dans un environnement distribué, depuis un point centralisé.

INTRODUCTION À APACHE AIRFLOW

Architecture Airflow



CONCEPTS CLÉS D'AIRFLOW

Les concepts de base d'Airflow

- **Définition d'un DAG en Python :**
  Un DAG dans Apache Airflow est défini par un script Python. Ce script spécifie les tâches à accomplir et la manière dont elles sont liées entre elles. Un DAG ne contient que des définitions et des dépendances, sans exécuter les tâches directement. Chaque tâche est exécutée lorsque les conditions de dépendance sont remplies.

- **Structure de base d'un DAG :**
  Un DAG est composé de plusieurs éléments principaux :
  - **Import des bibliothèques Airflow :** Vous devez d'abord importer les classes nécessaires, telles que DAG, les opérateurs (comme PythonOperator, BashOperator), et d'autres utilitaires pour créer le workflow.
  - **Déclaration des paramètres du DAG :** Le DAG est créé avec des paramètres tels que son identifiant (dag_id), sa fréquence d'exécution (schedule_interval), sa date de début (start_date), et d'autres paramètres (comme le catchup, qui contrôle si les exécutions manquées doivent être récupérées).

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

def my_function():
    print("Hello, Airflow!")

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 1, 1),
    'retries': 1,
}

dag = DAG(
    'my_first_dag',
    default_args=default_args,
    schedule_interval='@daily',
)

task = PythonOperator(
    task_id='print_hello',
    python_callable=my_function,
    dag=dag
)
```

CONCEPTS CLÉS D'AIRFLOW

Les concepts de base d'Airflow

- **Définition des dépendances :**
  Une fois les tâches définies, les dépendances entre elles sont spécifiées par des opérateurs comme >> (dépendance séquentielle) ou << (dépendance inverse).
  task_1 >> task_2 signifie que task_2 ne s'exécutera qu'après la réussite de task_1.

- **Paramètres supplémentaires pour les tâches :**
  Chaque tâche peut être configurée avec des paramètres supplémentaires comme :
  - **retries** : Nombre de tentatives en cas d'échec.
  - **retry_delay** : Intervalle entre les tentatives de réessai.
  - **timeout** : Temps maximal d'exécution d'une tâche avant de la marquer comme échouée.

- **Avantages de la structure Python :**
  - **Simplicité et flexibilité :** Grâce à Python, les utilisateurs peuvent facilement créer des workflows complexes, gérer les dépendances et personnaliser les paramètres.
  - **Lisibilité :** Les DAGs sont faciles à lire et à maintenir, avec des dépendances clairement définies.

CONCEPTS CLÉS D'AIRFLOW

Les tâches dans un DAG - Définition et exemples

- **Dépendances entre les tâches :**
  Chaque tâche dans un DAG dépend d'autres tâches. Les dépendances sont définies pour garantir que les tâches sont exécutées dans l'ordre souhaité. Les opérateurs >> et << permettent de spécifier ces relations.
  - **Exemple :** task_1 >> task_2 signifie que task_2 s'exécute après la fin de task_1.
  - Possibilité d'exécuter des tâches en parallèle si elles ne dépendent pas les unes des autres.
- **Planification des DAGs :**
  La planification (ou scheduling) permet de déclencher les DAGs à des intervalles spécifiques. Airflow offre des options simples comme @daily, @hourly, ou des expressions CRON personnalisées pour des horaires complexes.
  - **Exemple CRON :** '0 12 * * *' pour exécuter un DAG tous les jours à midi.
- **Gestion des exécutions manquées (Catchup) :**
  Airflow exécute les DAGs manqués si le service a été interrompu. Ce comportement peut être désactivé via l'option catchup=False.
- **Flexibilité de la planification :**
  Vous pouvez ajuster dynamiquement la fréquence d'exécution directement dans le code ou l'interface utilisateur d'Airflow pour s'adapter à des besoins spécifiques.

CONCEPTS CLÉS D'AIRFLOW

Les opérateurs - Types et cas d'usage (BashOperator, PythonOperator, etc.)

- **Qu'est-ce qu'un opérateur dans Airflow ?**
  Un **opérateur** dans Airflow est une abstraction d'une tâche. Il encapsule la logique de ce que doit accomplir chaque étape d'un DAG. Les opérateurs sont utilisés pour définir les actions à réaliser dans un workflow, comme l'exécution d'un script, l'appel d'une fonction ou la soumission d'un job.
- **Principaux types d'opérateurs dans Airflow :**
  Airflow offre de nombreux opérateurs prêts à l'emploi pour couvrir une grande variété de cas d'usage :
- **PythonOperator** : Exécute une fonction Python définie par l'utilisateur.
  - **Cas d'usage :** Appeler des fonctions Python pour des traitements spécifiques.
  - **Exemple :**

```
PythonOperator(
    task_id='run_python_task',
    python_callable=my_function,
    dag=dag
)
```

CONCEPTS CLÉS D'AIRFLOW

Les opérateurs - Types et cas d'usage (BashOperator, PythonOperator, etc.)

- **BashOperator** : Exécute des commandes shell ou des scripts Bash.
  - **Cas d'usage :** Lancer des scripts d'automatisation ou des commandes système.
  - **Exemple :**

```
BashOperator(
    task_id='run_bash_command',
    bash_command='echo "Hello Airflow!"',
    dag=dag
)
```

- **HiveOperator** : Exécute des requêtes SQL sur des bases de données Hive.
  - **Cas d'usage :** Traiter ou analyser des données dans un cluster Hadoop.
  - **Exemple :**

```
HiveOperator(
    task_id='hive_query',
    hql='SELECT * FROM table_name',
    dag=dag
)
```

CONCEPTS CLÉS D'AIRFLOW

Les opérateurs - Types et cas d'usage (BashOperator, PythonOperator, etc.)

- **SparkSubmitOperator** : Soumet un job Apache Spark.
  - **Cas d'usage :** Lancer des tâches de traitement massivement parallèle sur un cluster Spark.
  - **Exemple :**

```
SparkSubmitOperator(
    task_id='submit_spark_job',
    application='/path/to/app.py',
    dag=dag
)
```

- **EmailOperator** : Envoie des notifications par email.
- **Cas d'usage :** Envoyer des alertes ou des rapports à la fin de l'exécution d'un DAG.
- **Exemple :**

```
EmailOperator(
    task_id='send_email',
    to='example@example.com',
    subject='DAG Success',
    html_content='<h3>DAG Completed</h3>',
    dag=dag
)
```

CONCEPTS CLÉS D'AIRFLOW

Les opérateurs - Types et cas d'usage (BashOperator, PythonOperator, etc.)

- **Opérateurs personnalisés :**
  Si les opérateurs standards ne répondent pas aux besoins spécifiques d'un projet, il est possible de créer des **opérateurs personnalisés** en héritant de la classe BaseOperator. Cela permet d'étendre les fonctionnalités d'Airflow pour intégrer des outils internes ou des services tiers.
  - **Exemple :** Créer un opérateur pour interagir avec une API spécifique ou pour automatiser des processus internes à une entreprise.

CONCEPTS CLÉS D'AIRFLOW

Sensors - Surveillance et déclenchement des tâches

- **Qu'est-ce qu'un Sensor dans Airflow ?**
  Un **Sensor** est un type d'opérateur dans Airflow qui attend qu'un certain événement ou condition soit rempli avant de continuer l'exécution du DAG. Les Sensors sont utilisés pour surveiller l'état d'une ressource externe (fichier, base de données, API) et déclencher des tâches lorsque cette condition est satisfaite.
- **Types de Sensors :**
  Airflow propose plusieurs types de Sensors pour surveiller différentes conditions. Voici quelques exemples
  - **FileSensor** : Surveille l'existence d'un fichier à un emplacement spécifique.
    - **Cas d'usage :** Attendre qu'un fichier de données soit déposé dans un répertoire avant de lancer un traitement.
    - **Exemple :**

```python
from airflow.sensors.filesystem import FileSensor

file_sensor_task = FileSensor(
    task_id='check_for_file',
    filepath='/path/to/file.txt',
    poke_interval=10,  # Vérifie toutes les 10 secondes
    timeout=600,  # Abandonne après 10 minutes
    dag=dag
)
```

CONCEPTS CLÉS D'AIRFLOW

Sensors - Surveillance et déclenchement des tâches

- **HttpSensor** : Vérifie si une URL ou une API répond avec succès.
  - **Cas d'usage :** Attendre la disponibilité d'une API externe avant de lancer des requêtes.
  - **Exemple :**

```python
from airflow.sensors.http_sensor import HttpSensor

http_sensor_task = HttpSensor(
    task_id='check_api_status',
    http_conn_id='my_api',
    endpoint='status',
    response_check=lambda response: response.status_code == 200,
    poke_interval=10,
    dag=dag
)
```

CONCEPTS CLÉS D'AIRFLOW

Sensors - Surveillance et déclenchement des tâches

- **S3KeySensor** : Surveille l'apparition d'un fichier dans un bucket S3.
  - **Cas d'usage :** Attendre qu'un fichier soit uploadé dans un bucket S3 avant de lancer le traitement.
  - **Exemple :**

```
from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor

s3_sensor_task = S3KeySensor(
    task_id='check_for_s3_file',
    bucket_key='my_bucket/my_key',
    bucket_name='my_bucket',
    poke_interval=10,
    dag=dag
)
```

CONCEPTS CLÉS D'AIRFLOW

Sensors - Surveillance et déclenchement des tâches

- **Paramètres des Sensors :**
  Les Sensors sont configurés avec les paramètres suivants :
  - **poke_interval** : Fréquence à laquelle le sensor vérifie la condition (en secondes).
  - **timeout** : Temps maximal pendant lequel le sensor attend avant de considérer que la condition ne sera pas remplie (en secondes).
  - **soft_fail** : Si ce paramètre est activé, le sensor échoue doucement si le temps de surveillance dépasse le timeout, sans faire échouer tout le DAG.

- **Utilité des Sensors :**
  Les Sensors sont utiles pour les pipelines qui dépendent de ressources externes. Par exemple, si un traitement dépend de la disponibilité d'un fichier ou d'une réponse d'API, le Sensor attendra que cette condition soit remplie avant de déclencher la suite du pipeline.

- **Stratégie de surveillance continue (poke) :**
  Les Sensors "poke" régulièrement pour vérifier si la condition est remplie. Cette approche permet une surveillance constante des ressources externes sans surcharger le système.

CONCEPTS CLÉS D'AIRFLOW

XComs - Communication entre tâches dans un DAG

- **Qu'est-ce que XCom dans Airflow ?**
  Les **XComs** (Cross-Communications) sont des mécanismes dans Airflow qui permettent aux tâches d'échanger des informations entre elles. Cela permet à une tâche de partager des résultats ou des données avec une autre tâche dans un DAG.
- **Fonctionnement des XComs :**
  - **Push** : Une tâche peut "pusher" une donnée (stockée sous forme de clé-valeur) dans Airflow.
  - **Pull** : Une autre tâche peut "puller" (récupérer) cette donnée et l'utiliser dans son exécution.
- Les XComs permettent de maintenir un flux de données entre les différentes étapes d'un pipeline sans avoir à recourir à des fichiers externes ou des bases de données pour stocker des résultats intermédiaires.
- **Exemple d'utilisation de XCom :**
  - Une tâche collecte des données à partir d'une API et les pousse dans XCom.
  - Une tâche suivante récupère ces données via XCom pour les transformer.

```python
# Pusher une donnée dans XCom
def push_function(**context):
    context['ti'].xcom_push(key='data', value='Hello from XCom!')

push_task = PythonOperator(
    task_id='push_task',
    python_callable=push_function,
    dag=dag
)

# Puller une donnée de XCom
def pull_function(**context):
    data = context['ti'].xcom_pull(key='data')
    print(f"Pulled data: {data}")

pull_task = PythonOperator(
    task_id='pull_task',
    python_callable=pull_function,
    dag=dag
)

push_task >> pull_task
```

CONCEPTS CLÉS D'AIRFLOW

XComs - Communication entre tâches dans un DAG

- **Utilisation des XComs :**
  - **Partage de résultats entre tâches :** Par exemple, une tâche de collecte de données peut pusher les résultats dans XCom, et une tâche de transformation peut les puller pour les utiliser.
  - **Conditions et dépendances :** XComs peuvent être utilisés pour influencer la logique des DAGs, comme passer des signaux entre les tâches pour déclencher différentes actions en fonction des résultats.

- **Principaux paramètres :**
  - **Key** : Chaque donnée poussée dans XCom est associée à une clé (key), ce qui permet à d'autres tâches de la puller en utilisant cette clé.
  - **Value** : La donnée ou l'objet qui est stocké dans XCom.
  - **Task Instance (TI)** : XCom est géré via l'instance de la tâche (ti), qui est utilisée pour pusher et puller les valeurs.

- **Limites des XComs :**
  - Les XComs ne sont pas conçus pour échanger de grandes quantités de données. Ils sont mieux adaptés pour le transfert d'informations légères comme des messages ou des petites valeurs intermédiaires.
  - Si une tâche a besoin de partager des fichiers volumineux ou des données massives, il est préférable d'utiliser un stockage externe comme HDFS ou S3.

CONCEPTS CLÉS D'AIRFLOW

Définir des règles de déclenchement pour les tâches

- **Qu'est-ce qu'une Trigger Rule ?**
  Dans Airflow, une **Trigger Rule** définit sous quelles conditions une tâche peut être exécutée, en fonction de l'état des autres tâches du DAG. Par défaut, les tâches s'exécutent uniquement si toutes les tâches précédentes ont réussi. Cependant, les Trigger Rules offrent une plus grande flexibilité pour contrôler les conditions de déclenchement.
- **Types de Trigger Rules dans Airflow :**
  - **all_success (par défaut)** : La tâche s'exécute seulement si toutes les tâches précédentes ont réussi.
  - **all_failed** : La tâche s'exécute seulement si toutes les tâches précédentes ont échoué.
  - **one_success** : La tâche s'exécute si au moins une des tâches précédentes a réussi.
  - **one_failed** : La tâche s'exécute si au moins une des tâches précédentes a échoué.
  - **none_failed** : La tâche s'exécute si aucune des tâches précédentes n'a échoué.
  - **none_skipped** : La tâche s'exécute si aucune des tâches précédentes n'a été ignorée.

- **Utilisation d'une Trigger Rule :**
  Pour définir une Trigger Rule, il suffit de l'ajouter comme paramètre dans l'opérateur.

```python
from airflow.operators.dummy_operator import DummyOperator

task_1 = DummyOperator(
    task_id='task_1',
    dag=dag
)

task_2 = DummyOperator(
    task_id='task_2',
    trigger_rule='one_failed',  # La tâche s'exécute si une des tâches échoue
    dag=dag
)

task_1 >> task_2
```

CONCEPTS CLÉS D'AIRFLOW

Définir des règles de déclenchement pour les tâches

- **Cas d'usage des Trigger Rules :**
  - **Gestion des erreurs** : Les règles comme one_failed ou all_failed sont utiles pour exécuter des tâches de récupération en cas d'échec, comme l'envoi d'une alerte ou l'exécution d'un rollback.
  - **Tâches conditionnelles** : Utiliser one_success permet d'exécuter une tâche dès qu'une branche de votre DAG réussit, sans attendre que toutes les autres branches se terminent.
  - **Scénarios complexes** : Les règles none_failed et none_skipped peuvent être utiles dans des workflows complexes où certaines branches peuvent être ignorées ou échouées sans affecter l'exécution globale.

- **Combinaison des Trigger Rules :**
  Les Trigger Rules permettent d'ajouter de la flexibilité aux DAGs, surtout lorsque des workflows complexes avec plusieurs branches doivent gérer des erreurs ou des réussites partielles. Elles garantissent que certaines tâches peuvent s'exécuter même si une ou plusieurs étapes précédentes ont échoué.

CRÉATION ET GESTION DES DAGS

Hooks dans Airflow - Interactions avec des systèmes externes

- **Qu'est-ce qu'un Hook dans Airflow ?**
  Un **Hook** dans Airflow est une interface qui permet de se connecter à des systèmes externes (bases de données, API, services cloud, etc.) pour interagir avec ces ressources. Les Hooks sont souvent utilisés pour récupérer des données, effectuer des opérations CRUD (Create, Read, Update, Delete), ou interagir avec des services externes pendant l'exécution des tâches.
- **Types de Hooks dans Airflow :**
  Airflow propose plusieurs Hooks pour interagir avec des services et des systèmes externes couramment utilisés. Voici quelques exemples :
- **MySqlHook** : Interagit avec des bases de données MySQL, permettant de lire ou écrire des données via SQL.
  - **Cas d'usage :** Exécuter des requêtes SQL dans une base MySQL.
  - **Exemple :**

```
from airflow.providers.mysql.hooks.mysql import MySqlHook

mysql_hook = MySqlHook(mysql_conn_id='my_mysql_conn')
connection = mysql_hook.get_conn()
cursor = connection.cursor()
cursor.execute("SELECT * FROM my_table")
rows = cursor.fetchall()
```

CRÉATION ET GESTION DES DAGS

Hooks dans Airflow - Interactions avec des systèmes externes

- **PostgresHook** : Connexion et interaction avec des bases de données PostgreSQL.
  - **Cas d'usage :** Extraire ou insérer des données dans une base PostgreSQL.
  - **Exemple :**

```
from airflow.providers.postgres.hooks.postgres import PostgresHook

pg_hook = PostgresHook(postgres_conn_id='my_postgres_conn')
connection = pg_hook.get_conn()
cursor = connection.cursor()
cursor.execute("INSERT INTO my_table (id, name) VALUES (1, 'Airflow')")
```

- **HttpHook** : Pour interagir avec des APIs HTTP (GET, POST, etc.).
  - **Cas d'usage :** Envoyer des requêtes vers des API externes et récupérer des données.
  - **Exemple :**

```
from airflow.providers.http.hooks.http import HttpHook

http_hook = HttpHook(http_conn_id='my_http_conn', method='GET')
response = http_hook.run(endpoint='status')
print(response.text)
```

CRÉATION ET GESTION DES DAGS

Hooks dans Airflow - Interactions avec des systèmes externes

- **S3Hook** : Interagit avec des services S3 pour transférer des fichiers.
  - **Cas d'usage :** Charger ou télécharger des fichiers vers/depuis des buckets S3.
  - **Exemple :**

```
from airflow.providers.amazon.aws.hooks.s3 import S3Hook

s3_hook = S3Hook(aws_conn_id='my_aws_conn')
s3_hook.load_file(filename='/path/to/file.txt', key='my_bucket/my_file.txt'
```

- **Comment utiliser un Hook :**
  Les Hooks sont généralement utilisés dans des opérateurs personnalisés ou dans des tâches Python pour interagir avec les ressources externes.
- Vous devez définir une connexion dans l'interface Airflow pour que le Hook utilise ces informations de connexion (par exemple, des identifiants pour accéder à une base de données ou à une API).
- **Cas d'usage des Hooks :**
  - **Extraction de données :** Un Hook peut être utilisé pour extraire des données d'une base de données externe avant de les transformer dans un DAG.
  - **Interaction avec des services cloud :** Les Hooks facilitent l'intégration avec des services cloud comme AWS, Google Cloud, ou Azure, permettant de lire et d'écrire des données sur ces plateformes directement dans un workflow Airflow.
  - **Automatisation des tâches récurrentes :** Les Hooks permettent d'automatiser l'exécution de tâches courantes, comme l'interrogation d'une API, l'insertion de données dans une base, ou le transfert de fichiers.

CRÉATION ET GESTION DES DAGS

Gestion des erreurs dans les DAGs (gestion des erreurs et alertes)

- **Pourquoi la gestion des erreurs est cruciale dans Airflow ?**
  Dans un workflow complexe, les erreurs peuvent survenir pour diverses raisons : échec de connexion, ressources insuffisantes, données manquantes, etc. Une bonne gestion des erreurs garantit que les DAGs peuvent gérer les échecs sans interrompre l'ensemble du pipeline, tout en alertant les responsables pour des actions correctives rapides.
- **Stratégies de gestion des erreurs dans Airflow :**
  - **Retries (réessais)** : Airflow permet de configurer un nombre de tentatives de réexécution automatique en cas d'échec d'une tâche. Cela permet de gérer les erreurs transitoires, comme une perte de connexion temporaire.
    - **Exemple :** Vous pouvez configurer une tâche pour qu'elle soit réessayée jusqu'à 3 fois, avec un délai de 5 minutes entre chaque tentative.

```
task = PythonOperator(
    task_id='example_task',
    python_callable=my_function,
    retries=3,
    retry_delay=timedelta(minutes=5),
    dag=dag,
)
```

CRÉATION ET GESTION DES DAGS

Gestion des erreurs dans les DAGs (gestion des erreurs et alertes)

- **Timeout (délai d'exécution)** : Vous pouvez définir un temps maximal pour qu'une tâche s'exécute. Si ce délai est dépassé, la tâche échoue et Airflow passe à l'étape suivante, évitant ainsi les blocages prolongés.
  - **Exemple :** Si une tâche prend plus de 10 minutes, elle est interrompue et marquée comme échouée.

```
task = BashOperator(
    task_id='example_timeout_task',
    bash_command='my_long_command.sh',
    execution_timeout=timedelta(minutes=10),
    dag=dag,
)
```

- **Trigger Rules pour la gestion des erreurs** : Vous pouvez utiliser des règles de déclenchement (trigger_rule) comme all_failed ou one_failed pour exécuter certaines tâches en cas d'échec de tâches précédentes, permettant ainsi de mettre en place des mécanismes de récupération ou d'alertes.
  - **Exemple :** Une tâche de notification peut être déclenchée si au moins une tâche précédente échoue.

```
task = EmailOperator(
    task_id='send_failure_email',
    to='admin@example.com',
    subject='Task Failed',
    trigger_rule='one_failed',
    dag=dag,
)
```

CRÉATION ET GESTION DES DAGS

Gestion des erreurs dans les DAGs (gestion des erreurs et alertes)

- **Alertes en cas d'échec :**
  Airflow permet de configurer des alertes pour informer les équipes en cas d'échec d'une tâche ou d'un DAG complet. Vous pouvez envoyer des notifications par email, via Slack, ou intégrer des systèmes d'alertes plus avancés comme PagerDuty.
- **Exemple avec EmailOperator :**

```
from airflow.operators.email_operator import EmailOperator

email_alert = EmailOperator(
    task_id='send_email',
    to='team@example.com',
    subject='Airflow Alert: Task Failed',
    html_content='A task has failed in your DAG!',
    dag=dag
)
```

CRÉATION ET GESTION DES DAGS

Gestion des erreurs dans les DAGs (gestion des erreurs et alertes)

- **Notification sur Slack :** Vous pouvez également utiliser des opérateurs personnalisés ou intégrer Airflow avec des services externes comme Slack pour envoyer des notifications d'alertes dans un canal spécifique.

```
from airflow.providers.slack.operators.slack_webhook import SlackWebhookOperator

slack_alert = SlackWebhookOperator(
    task_id='slack_alert',
    http_conn_id='slack_connection',
    webhook_token='your_slack_webhook_token',
    message=":red_circle: Task Failed in Airflow DAG",
    dag=dag
)
```

- **Surveillance et gestion des erreurs via l'interface utilisateur :**
  Airflow propose un tableau de bord où vous pouvez suivre l'état d'exécution des tâches et des DAGs. Les tâches échouées sont marquées en rouge, et il est possible de consulter les logs détaillés pour diagnostiquer la cause de l'échec.
- **Logs pour le débogage :**
  Chaque tâche dans Airflow produit des logs que vous pouvez consulter directement via l'interface utilisateur pour analyser les erreurs. Cela permet d'identifier la source du problème (erreur de script, dépassement de délai, problème de connexion, etc.).

EXÉCUTION ET MONITORING DES WORKFLOWS

Exécution et suivi des DAGs

- **Exécution des DAGs :**
  - Les DAGs peuvent être exécutés manuellement via l'interface ou la CLI, ou automatiquement selon un intervalle planifié.
  - Chaque exécution génère un **DAG Run** et des **Task Instances**, enregistrant l'état de chaque tâche.
- **Exemple de CLI :**

```
airflow dags trigger my_dag_id
```

- **Suivi des DAGs :**
  - **Vue Gantt** : Affiche le temps d'exécution des tâches.
  - **Vue Graphique** : Visualisation des tâches avec état (réussie, échouée, en attente).
  - **Logs des tâches** : Consultables dans l'interface pour diagnostiquer les erreurs.
- **Redémarrage des tâches échouées :**
  - Airflow permet de relancer les tâches échouées sans redémarrer tout le DAG, depuis l'interface utilisateur.
- **Intégration avec des outils externes** :
  - Airflow peut être surveillé avec des outils comme **Prometheus** et **Grafana** pour des métriques en temps réel.

EXÉCUTION ET MONITORING DES WORKFLOWS

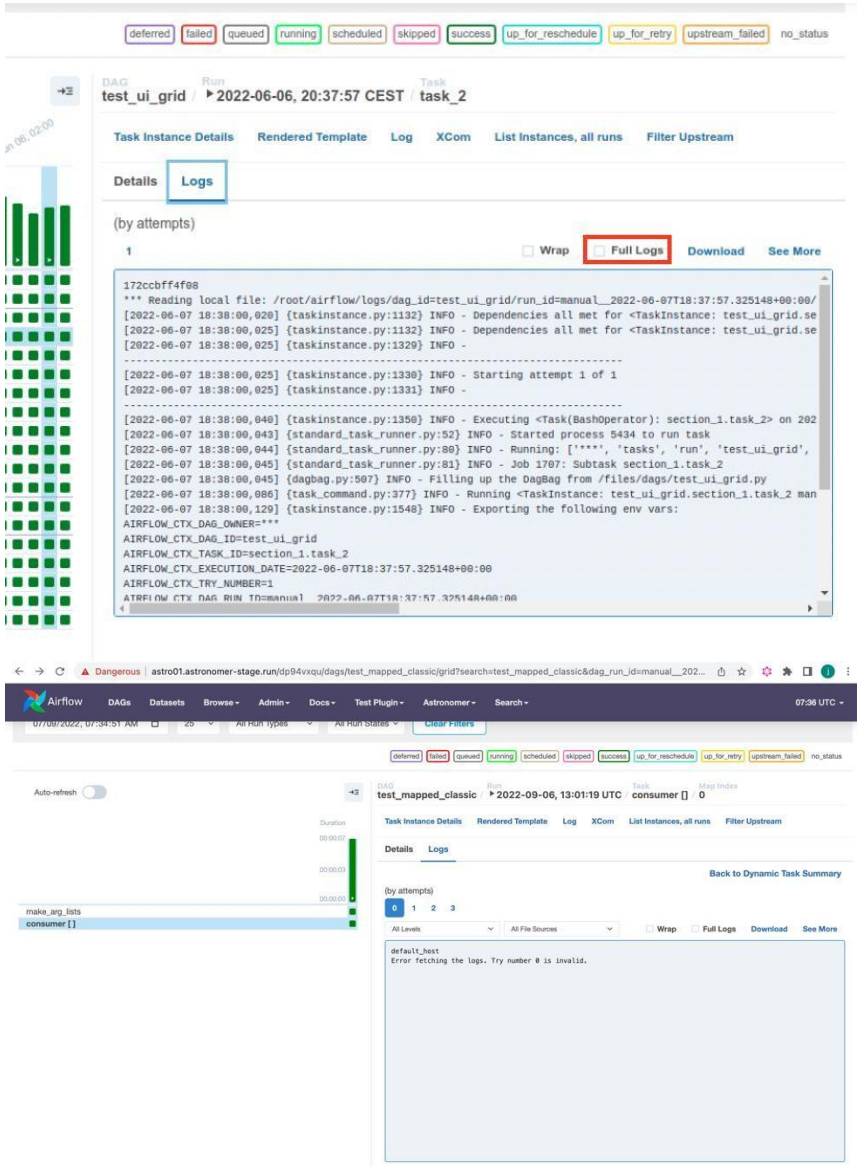
Monitoring via l'interface utilisateur d'Airflow (UI)

- **Tableau de bord principal :**
  - Affiche tous les DAGs actifs avec leur état (en cours, échoué, réussi).
  - Permet de visualiser rapidement l'état d'exécution et les derniers résultats des DAGs.
- **Vue graphique :**
  - Représentation visuelle des tâches et de leurs dépendances dans un DAG.
  - Couleurs pour chaque état : Vert (réussi), Rouge (échoué), Jaune (en cours), Gris (en attente).
- **Logs des tâches :**
  - Chaque tâche génère des logs consultables directement via l'UI.
  - Utiles pour diagnostiquer les échecs ou comprendre les étapes de traitement.
- **Vue Gantt :**
  - Montre la durée d'exécution des tâches sous forme de diagramme Gantt, permettant de suivre le temps de traitement par tâche.
- **Vue Tree :**
  - Affiche l'historique des exécutions d'un DAG sous forme arborescente, avec une vue d'ensemble des succès et échecs.
- **Relancer des tâches via l'UI :**
  - Possibilité de relancer les tâches échouées directement depuis l'interface sans redémarrer tout le DAG.

EXÉCUTION ET MONITORING DES WORKFLOWS

Gestion des logs et des erreurs

- **Logs des tâches :**
  - Chaque tâche dans Airflow génère des logs détaillés qui permettent de suivre l'exécution et d'identifier les erreurs.
  - Ces logs sont accessibles directement via l'interface utilisateur d'Airflow, dans la vue de chaque tâche.
- **Diagnostic des erreurs :**
  - En cas d'échec d'une tâche, les logs offrent des détails sur la cause de l'échec (problèmes de connexion, erreurs de script, etc.).
  - Ils permettent de remonter à l'origine de l'erreur pour effectuer un débogage rapide.
- **Stratégies de gestion des erreurs :**
  - **Retries automatiques** : Configurer des réessais pour réexécuter les tâches échouées.
  - **Alertes** : Configurer des alertes via email ou Slack pour être averti immédiatement en cas d'échec.
  - **Clear & relaunch** : Relancer manuellement les tâches échouées après avoir corrigé le problème.
- **Accès aux logs via l'UI :**
  - Airflow permet d'accéder aux logs de manière centralisée via son interface, avec des options pour les télécharger ou les consulter directement.



OPTIMISATION DES WORKFLOWS ET BONNES PRATIQUES

Optimisation des workflows avec Airflow

- **Parallélisation des tâches :**
  - Airflow permet d'exécuter des tâches en parallèle pour réduire le temps d'exécution global.
  - En définissant des tâches indépendantes, vous pouvez exécuter plusieurs tâches simultanément, améliorant ainsi l'efficacité du workflow.
- **Réutilisation des tâches communes :**
  - Airflow permet de créer des **subDAGs** pour réutiliser des groupes de tâches fréquemment utilisés dans plusieurs DAGs, évitant la duplication de code.
- **Gestion intelligente des ressources :**
  - Utilisez le **Kubernetes Executor** pour créer dynamiquement des pods pour chaque tâche, garantissant une utilisation optimale des ressources et une scalabilité automatique.
- **Optimisation des retries et des timeouts :**
  - Configurer des **timeouts** pour éviter les tâches qui bloquent le DAG et utiliser des **retries** pour gérer les erreurs transitoires.
- **Monitoring et analyse :**
  - Grâce à l'intégration avec des outils comme **Prometheus** ou **Grafana**, vous pouvez surveiller les performances des workflows et ajuster les configurations pour maximiser l'efficacité.
- **Trigger Rules avancées :**
  - Utilisez des règles de déclenchement (Trigger Rules) pour contrôler avec précision quand les tâches doivent s'exécuter, même dans des cas complexes (par exemple, si une tâche échoue, d'autres tâches peuvent tout de même continuer).