

Chapitre 3:

Dagster

4DATA

Benjamin Jurczak



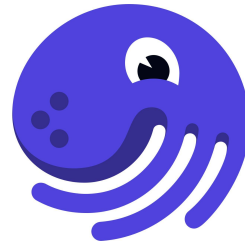
Sommaire

1. Introduction
2. Concepts clés
3. Bonnes pratiques
4. Exemples



1. Introduction à Dagster

1. Introduction à Dagster

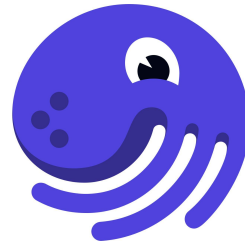


Un outil moderne pour l'orchestration de workflows complexes

Objectifs du chapitre :

- Comprendre ce qu'est Dagster et pourquoi l'utiliser.
- Explorer ses concepts clés et l'appliquer à des cas concrets.

1. Introduction à Dagster



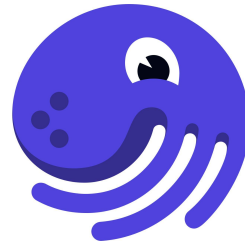
Un outil moderne pour l'orchestration de workflows complexes

Dagster est un framework open source conçu pour orchestrer des pipelines de données. Il permet de concevoir, exécuter, surveiller et maintenir des workflows.

Caractéristiques principales :

- Centré sur les **données** et les **dépendances**.
- Intégration native avec des outils populaires comme Pandas, SQL, dbt.
- Adapté aux workflows modernes : ETL, ML, analyses.

1. Introduction à Dagster



Bref historique et évolution

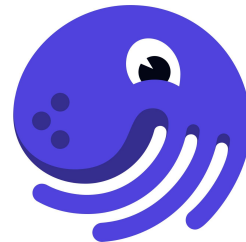
Origines : Projet open-source lancé en 2018 par Dagster Labs.

- Principales versions : Évolution vers une approche "*software-defined assets*" (2022).
- Adoption : Croissance rapide dans l'industrie (startups, scale-ups, entreprises).

pour qui ?

- Data Engineers, Data Scientists, Analystes.
- Équipes DevOps/DataOps cherchant à industrialiser des pipelines.

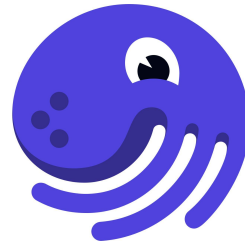
1. Introduction à Dagster



Les défis de l'orchestration

- Problématiques :
 - Dépendances complexes.
 - Débogage laborieux.
 - Manque de traçabilité.
- Solutions apportées par l'orchestration :
 - Automatisation, monitoring, scalabilité.

1. Introduction à Dagster



Un outil moderne pour l'orchestration de workflows complexes

Avantages :

- **Modularité** : Les pipelines sont construits avec des composants réutilisables (ops notamment).
- **Clarté** : Gestion explicite des entrées, sorties et dépendances.
- **Observabilité** : UI très riche → Monitoring.
- **Scalabilité** : Gestion des workflows locaux et en production.

Cas d'usage :

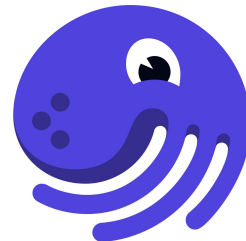
- Automatisation d'ETL.
- Pipelines d'apprentissage automatique.
- Pipelines complexes nécessitant une surveillance accrue.

1. Introduction à Dagster

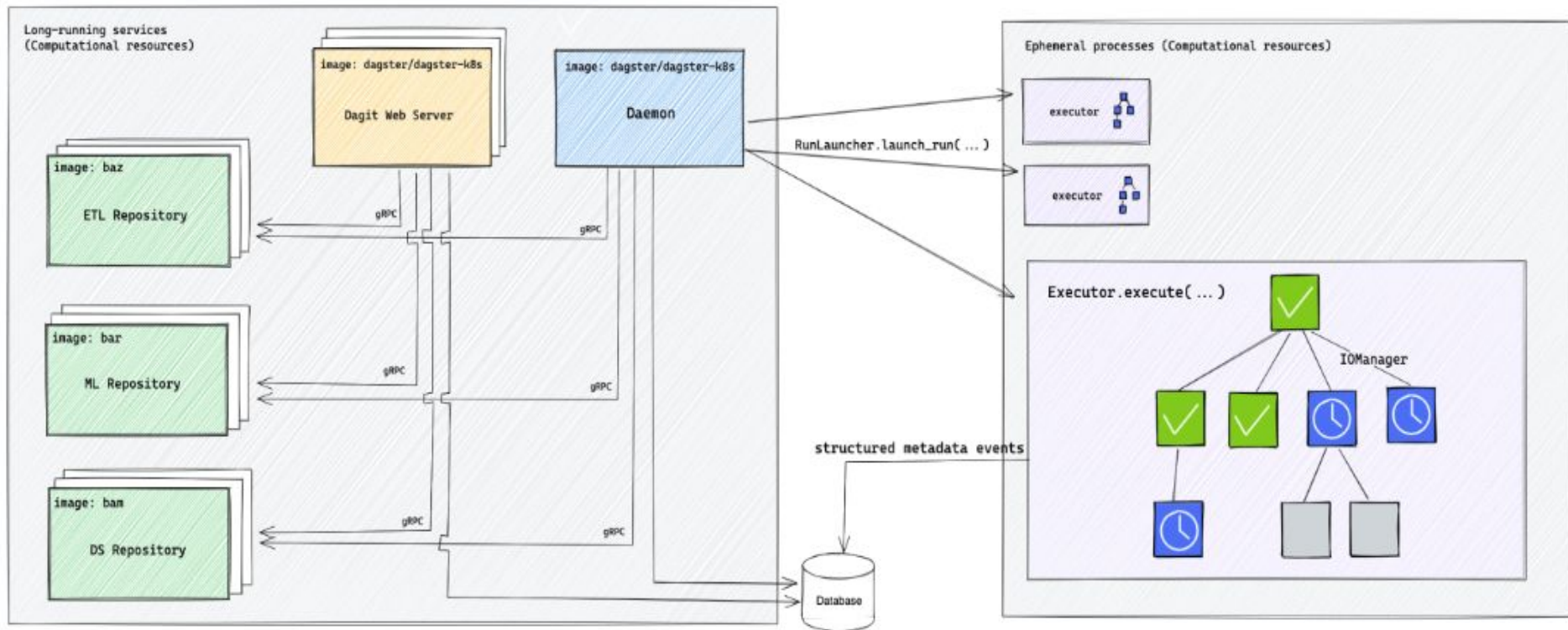


2. Concepts clés

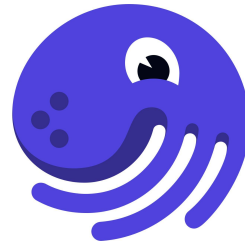
1. Concepts clés



Architecture complète



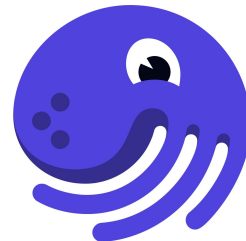
1. Concepts clés



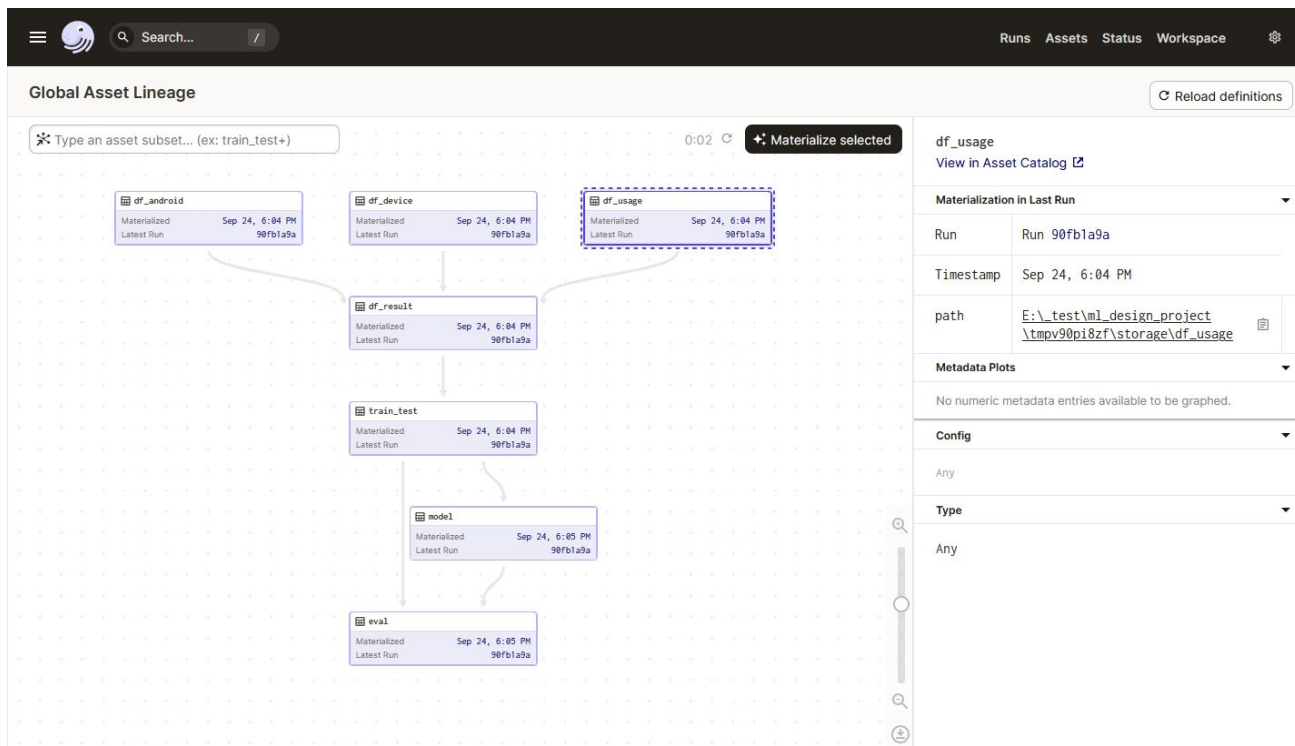
Interface Web de Dagster : Dagit & Dagster Webserver

- ◆ Dagster propose une interface web puissante pour gérer et superviser les workflows de données.
- ◆ **Dagit** interface historique, remplacée depuis par **Dagster Webserver**.
- ◆ **Rôle principal** : visualisation, exécution et debugging des jobs, assets et partitions.
- ◆ Permet d'exécuter manuellement des pipelines et d'analyser les dépendances entre les composants.

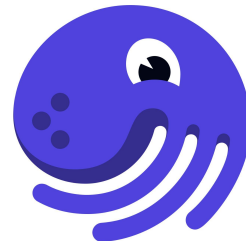
1. Concepts clés



Interface Web de Dagster : Dagit & Dagster Webserver



1. Concepts clés

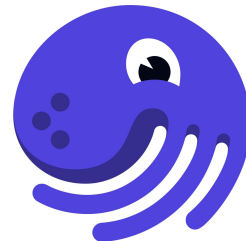


Navigation dans Dagster Webserver

Éléments clés de l'interface

- **Jobs** : Liste et visualisation des jobs
- **Runs** : Historique des exécutions
- **Logs** : Suivi des étapes et erreurs
- **Assets** : Vue sur les assets générés
- **Schedules & Sensors** : Gestion des exécutions automatiques
- et Open Source

1. Concepts clés



Navigation dans Dagster Webserver

Screenshot of the Dagster Webserver interface showing the **Deployment** tab.

The interface includes a top navigation bar with links: Overview, Runs, Catalog, Jobs, Automation, Insights, and Deployment. The current view is **Deployment**, which displays a list of alert policies.

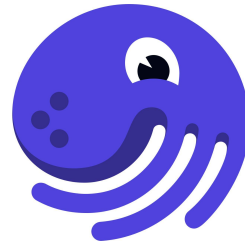
Deployment Tab:

- Filter:
- Connected to dagsterlabs
- Reload definitions

Alert Policies List:

Name	Notification service	Events	Applies to...	Actions
<input checked="" type="checkbox"/> Schedule Sensors Failing	#sales-eng	Tick failure	All jobs	<input type="button" value="v"/>
<input checked="" type="checkbox"/> alert_ops_hooli_agent_down	#eng-cloud-ops-errors	Agent downtime	All jobs	<input type="button" value="v"/>
<input checked="" type="checkbox"/> alert_ops_hooli_code_location_down	#eng-cloud-ops-errors	Code location error	All jobs	<input type="button" value="v"/>
<input checked="" type="checkbox"/> check_users	lopp@dagsterlabs.com	Check failed (WARNING) Check failed (ERROR) Check execution failed	RAW_DATA/users	<input type="button" value="v"/>
<input checked="" type="checkbox"/> email-failure-alert	nelson.bighetti@hooli.co.. richard.hendricks@hooli.co..	Job failure	notify: email	<input type="button" value="v"/>
<input checked="" type="checkbox"/> hooli code location failure	#sales-eng	Code location error	All jobs	<input type="button" value="v"/>
<input checked="" type="checkbox"/> hooli_agent_failure	#sales-eng	Agent downtime	All jobs	<input type="button" value="v"/>
<input checked="" type="checkbox"/> ml-failure	ml@hooli.com	Job failure	alert_team: ml	<input type="button" value="v"/>
<input checked="" type="checkbox"/> weekly_summary_alert	lopp@dagsterlabs.com	Materialization failure	ANALYTICS_r_summary	<input type="button" value="v"/>

1. Concepts clés



Une UI très (très) riche !

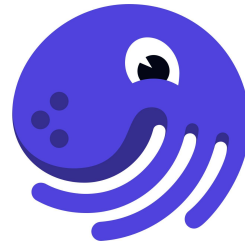
Historique des runs

- Affichage des runs réussis et échoués
- Statut des exécutions (Success, Failed, Canceled)
- Possibilité de rejouer une exécution avec de nouvelles configurations

◆ Versioning et Reproductibilité

- Stockage des exécutions passées
- Comparaison des configurations entre plusieurs runs
- Matérialisation des assets pour assurer la traçabilité

1. Concepts clés



Une UI très (très) riche !

Configuration et Paramétrage via l'Interface directement

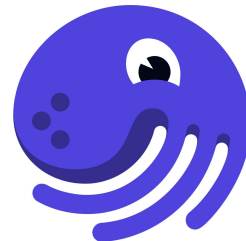
Configurer un job avant exécution

- Définition des paramètres (**ConfigSchema**)
- Chargement de fichiers de configuration
- Sélection des partitions et des ressources utilisées





◆ **Personnalisation et intégrations**

- Configuration des notifications en cas d'échec
- Connexion avec des systèmes externes (Prometheus, Grafana...)
- Exécution sur des backends spécifiques (Kubernetes, Celery, Dask...)

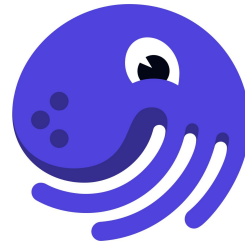
1. Concepts clés



Une UI (très) riche ! Abusez-en !

-  Interface intuitive et riche en fonctionnalités
-  Exécution et monitoring en temps réel des workflows
-  Suivi des exécutions passées et gestion avancée des logs
-  Intégration facile avec les infrastructures modernes

1. Concepts clés

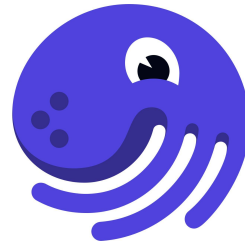


Les unités de base d'un pipeline Dagster

Dagster repose sur une approche modulaire pour construire des workflows **réutilisables** et **maintenables**.

- ◆ 3 éléments clés :
 - **Ops** : unités fondamentales d'exécution
 - **Graphes** : structuration des ops en workflows logiques
 - **Jobs** : orchestration et exécution d'un pipeline

1. Concepts clés

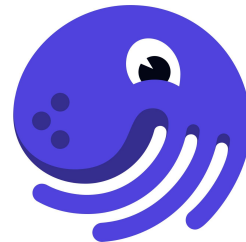


Les unités de base d'un pipeline Dagster

Les Ops – Briques fondamentales d'exécution

- ◆ Qu'est-ce qu'un **@op** ?
 - Une unité de traitement dans Dagster
 - Encapsule une fonction avec des entrées et sorties bien définies
 - Exécution indépendante avec typage statique

1. Concepts clés



Les unités de base d'un pipeline Dagster

Les Ops – Briques fondamentales d'exécution

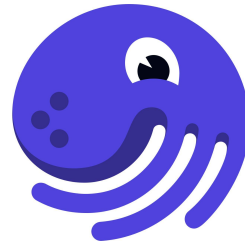
◆ Caractéristiques clés

- Peut recevoir et retourner des données
- Peut être combiné avec d'autres ops dans un pipeline
- Permet de tracer et déboguer les exécutions

```
from dagster import op

@op
def hello():
    return "Hello, Dagster!"
```

1. Concepts clés



Les unités de base d'un pipeline Dagster

Gestion Entrées-Sorties des Ops

Entrées (**In**) et Sorties (**Out**)

- Permettent de structurer le passage de données
- Fortement typées pour éviter les erreurs
- Peuvent être multiples (multi-inputs, multi-outputs)

◆ Exemple avec **In** et **Out**

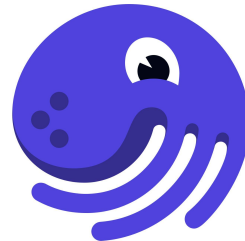
Gestion des dépendances

- Un **op** peut recevoir les résultats d'un autre **op**
- Utilisation de **@graph** pour organiser la logique

```
from dagster import In, Out, op

@op(ins={"name": In(str)}, out=Out(str))
def greet(name):
    return f"Hello, {name}!"
```

1. Concepts clés



Les unités de base d'un pipeline Dagster

Graphes (@graph) – Organisation des Ops

◆ Pourquoi utiliser des graphes ?

- Permet de structurer plusieurs **ops** en un ensemble logique
- Facilite la réutilisation et la modularité
- Permet de composer des workflows plus complexes

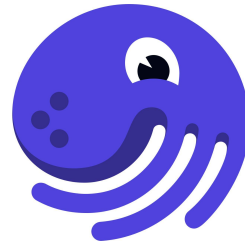
◆ Différence entre un **graph** et un **job**

- **Graph** = regroupe des **ops**, sans exécution propre
- **Job** = exécute un **graph** avec une configuration et un scheduler

```
from dagster import graph

@graph
def pipeline():
    greet(hello())
```

1. Concepts clés



Les unités de base d'un pipeline Dagster

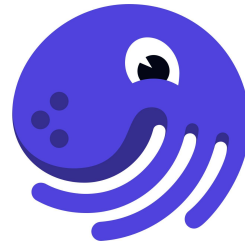
Jobs – Exécution et Orchestration d'un Pipeline

- ◆ Qu'est-ce qu'un **@job** ?
 - Un **job** orchestre l'exécution des **ops** d'un **graph**
 - Peut être configuré avec des paramètres
 - Peut être exécuté localement ou dans un scheduler
- ◆ Différence entre **op** et **job**
 - **op** : unité de traitement
 - **graph** : structure plusieurs **ops**
 - **job** : exécute un **graph** avec des paramètres
 - **my_job** orchestre l'exécution de **pipeline()**

```
from dagster import job

@job
def my_job():
    pipeline()
```


1. Concepts clés



Les unités de base d'un pipeline Dagster

Exécution d'un Job – Locale vs Scheduler

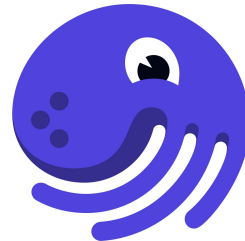
◆ Exécution locale

- Peut être lancée via Python ou l'interface Dagster Webserver
- Utile pour tester des pipelines avant le déploiement

◆ Exécution via un Scheduler

- Utilisation d'un **scheduler** ou de **sensors** pour déclencher automatiquement des jobs
- Permet d'intégrer Dagster dans des workflows de production

1. Concepts clés

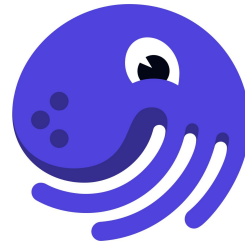


Les unités de base d'un pipeline Dagster

Introduction aux Software Defined Assets (SDA)

- ◆ Qu'est-ce qu'un Asset dans Dagster ?
 - Une **entité persistante** générée par un job (ex : fichier, table de base de données, modèle ML...)
 - Décrit dans le code avec `@asset` pour un suivi et une gestion améliorés
 - Permet un **suivi précis des transformations de données**

1. Concepts clés



Les unités de base d'un pipeline Dagster

◆ Software Defined Assets (SDA)

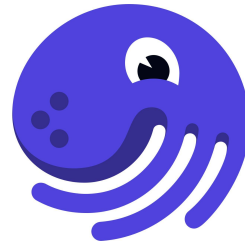
- Nouvelle approche orientée **données** plutôt que **tâches**
- Les **dépendances sont explicites** entre assets, facilitant la traçabilité
- Permet un **monitoring natif** de la qualité et de l'historique des assets



Pourquoi utiliser les SDA ?

- ✓ Traçabilité améliorée
- ✓ Dépendances explicites
- ✓ Monitoring et validation des assets intégrés

1. Concepts clés



Les unités de base d'un pipeline Dagster

Différences entre Ops et Assets

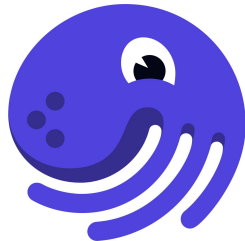
◆ **Ops** (tâches unitaires)

- Un **op** exécute une **fonction unique**
- Il **ne garde pas de trace des résultats** après exécution
- Il **ne peut pas être directement interrogé ou versionné**

◆ **Assets** (orientés données)

- Un **asset** **produit une donnée identifiable**
- Il **peut être stocké, versionné et interrogé** dans Dagster
- Possède des **dépendances explicites**, visibles dans l'interface

1. Concepts clés



Assets

An **asset** is an object in persistent storage, such as a table, file, or persisted machine learning model. An **asset definition** is a description, in code, of an asset that should exist and how to produce and update that asset.

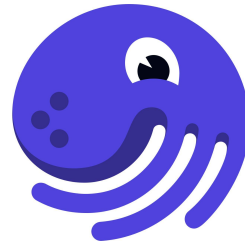
Asset definitions enable a declarative approach to data management, in which code is the source of truth on what data assets should exist and how those assets are computed. To learn how to define assets in code, see "[Defining assets](#)".

Materializing an asset is the act of running its function and saving the results to persistent storage. You can materialize assets from the Dagster UI or by invoking [Python APIs](#).

ASSETS VS OPS

Behind the scenes, the Python function in an asset is an **op**. A crucial distinction between asset definitions and ops is that asset definitions know about their dependencies, while ops do not. Ops aren't connected to dependencies until they're placed inside a graph. You do not need to use ops to use Dagster.

1. Concepts clés



Les unités de base d'un pipeline Dagster

Construire un Pipeline avec `@asset`

- ◆ Définir un asset simple

```
from dagster import asset

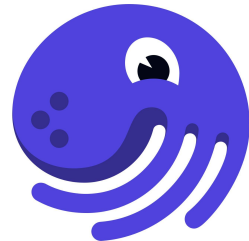
@asset
def raw_data():
    return {"name": "Alice", "age": 30}
```

Pour créer des dépendances entre assets

```
@asset
def transformed_data(raw_data):
    return {key: str(value).upper() for key, value in raw_data.items()}
```

💡 **Dagster Webserver** visualise automatiquement les liens entre `raw_data` et `transformed_data` !

1. Concepts clés



Les unités de base d'un pipeline Dagster

Dépendances et Traçabilité des Assets

◆ Pourquoi suivre les dépendances ?

- Visualisation des **données en entrée et sortie** de chaque asset
- Identification rapide des assets impactés en cas d'erreur
- Exécution automatique des **seuls assets nécessaires**

◆ Comment les gérer dans Dagster ?

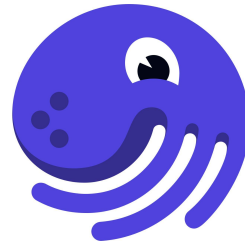
- Les dépendances sont déclarées **explicitement dans le code**
- L'interface Web permet d'afficher **le graphe des assets**
- Possibilité d'utiliser **des partitions pour optimiser l'exécution**

```
@asset
def final_dataset(transformed_data):
    return transformed_data
```



Dagster sait **qu'il doit d'abord exécuter** **transformed_data** avant **final_dataset**.

1. Concepts clés



Les unités de base d'un pipeline Dagster

Gestion des Assets avec Materializations et Observations

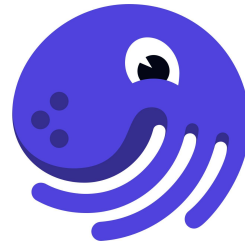
◆ Materializations (**@asset**)

- Permet de **générer et stocker un asset**
- Traçabilité des différentes versions d'un asset
- Monitoré directement dans **Dagster Webserver**

◆ Observations (**@observable_source_asset**)

- Permet de **suivre un asset externe** (ex : un fichier S3, une table SQL, etc.)
- Ne génère pas l'asset, mais permet **de surveiller ses mises à jour**

1. Concepts clés



Les unités de base d'un pipeline Dagster

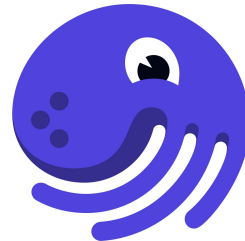
Gestion des Assets avec Materializations et Observations

- ◆ Exemple d'un asset avec Materialization

```
from dagster import AssetMaterialization

@asset
def processed_data():
    yield AssetMaterialization(asset_key="processed_data", description="Données nettoyées")
    return {"cleaned": True}
```

1. Concepts clés



Les unités de base d'un pipeline Dagster

Relations entre Assets, Partitions et Schedules

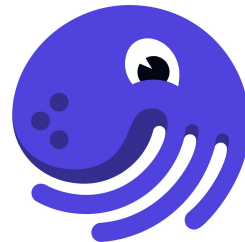
◆ Combiner Assets et Partitions

- Partitions = Diviser les assets par **jour, mois, région...**
- Permet **d'optimiser l'exécution** et de **réduire les coûts**
- Exécuter seulement les partitions modifiées

◆ Planification des exécutions

- **Schedules** : Déclenchent automatiquement la mise à jour des assets
- **Sensors** : Écoutent les changements d'assets et déclenchent une exécution si nécessaire

1. Concepts clés



Les unités de base d'un pipeline Dagster

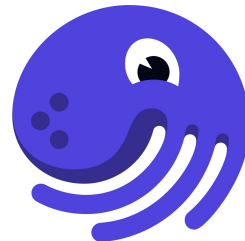
Exemple d'asset partitionné

```
from dagster import AssetKey, DailyPartitionsDefinition

@asset(partitions_def=DailyPartitionsDefinition(start_date="2023-01-01"))
def daily_report():
    return f"Report for {AssetKey.date_str}"
```

💡 Chaque exécution ne traite que la partition du jour, optimisant ainsi l'utilisation des ressources.

1. Concepts clés

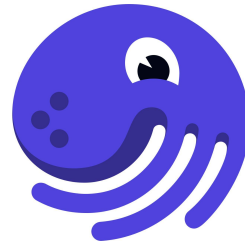


Les unités de base d'un pipeline Dagster

Pourquoi utiliser les Assets dans Dagster ?

- ✓ **Approche orientée données** pour une meilleure traçabilité
- ✓ **Monitoring natif et dépendances explicites**
- ✓ **Gestion des versions** et exécutions optimisées avec partitions
- ✓ **Planification automatique** grâce aux schedules et sensors

1. Concepts clés



Configuration dans Dagster

Pourquoi configurer les exécutions ?

- Permet de **personnaliser le comportement** des **ops** et **jobs**
- Facilite l'**exécution flexible** avec des paramètres dynamiques
- Simplifie la **gestion des environnements (dev, staging, prod)**

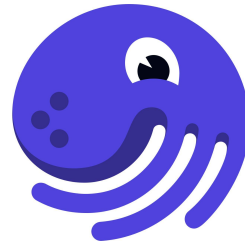
◆ Outils de configuration dans Dagster

- **ConfigSchema** : Définit les paramètres des **ops**
- **dagster.yaml** : Gère les environnements
- Overriding via **Dagster Webserver**
- Exécution conditionnelle avec **DynamicOutput** et **DynamicPartitions**



Une bonne configuration rend les pipelines plus robustes et réutilisables.

1. Concepts clés



Utilisation de ConfigSchema pour les paramètres des ops

Pourquoi utiliser ConfigSchema ?

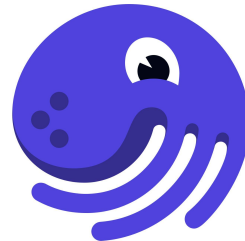
- Permet d'**ajouter des paramètres** aux **ops**
- Applique un **typage strict** pour éviter les erreurs
- Permet une **validation automatique** avant l'exécution

```
from dagster import op, Config

class GreetingConfig(Config):
    name: str

@op
def greet(config: GreetingConfig):
    return f"Hello, {config.name}!"
```

1. Concepts clés



Passage de Variables Dynamiques lors de l'Exécution

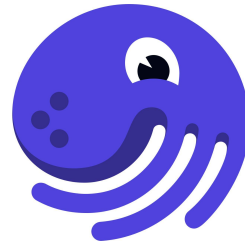
Pourquoi utiliser des variables dynamiques ?

- Adapter l'exécution en fonction d'un **contexte spécifique**
- Charger des **paramètres externes** (API, fichiers, bases de données)
- Permettre des exécutions **reproductibles et flexibles**

```
@op(config_schema={"threshold": int})
def filter_data(context):
    threshold = context.op_config["threshold"]
    return [x for x in range(10) if x > threshold]
```

Le seuil (**threshold**) peut être modifié à chaque exécution.

1. Concepts clés



Configuration des Environnements avec **dagster.yaml**

◆ Pourquoi utiliser **dagster.yaml** ?

- Centralise la **gestion des configurations** pour tous les jobs
- Permet de **définir les ressources** (base de données, stockage, logs...)
- Adapte l'exécution selon l'environnement (**dev, staging, prod**)

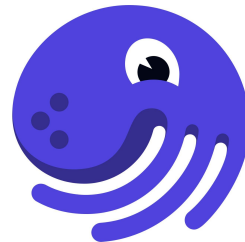
Définit un pipeline multi-processus avec stockage sur disque.

◆ Types de stockage et exécution

- In-memory (défaut)
- Filesystem (local)
- PostgreSQL, S3, GCS (pour production)

```
execution:  
  multiprocessing:  
    config:  
      max_concurrent: 4  
storage:  
  filesystem:  
    config:  
      base_dir: "/mnt/data"
```


1. Concepts clés



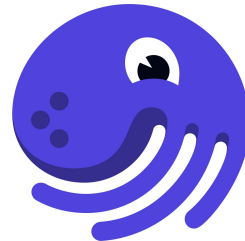
Configuration des Environnements avec **dagster.yaml**

```
! dagster.yaml
scheduler:
  module: dagster.core.scheduler
  class: DagsterDaemonScheduler

run_coordinator:
  module: dagster.core.run_coordinator
  class: QueuedRunCoordinator

run_launcher:
  module: dagster_docker
  class: DockerRunLauncher
  config:
    env_vars:
      # Make sure to use the same environment vars defined in the dagster_pipelines service
      - DAGSTER_POSTGRES_USER
      - DAGSTER_POSTGRES_HOSTNAME
      - DAGSTER_POSTGRES_PASSWORD
      - DAGSTER_POSTGRES_DB
    network: dagster_network
```

1. Concepts clés



Overriding de la Configuration dans Dagster Webserver

◆ Pourquoi modifier la config via l'interface ?

- Permet d'**ajuster les paramètres sans modifier le code**
- Utile pour **tester des exécutions** avec différentes valeurs
- Facilite le **debugging et l'expérimentation**

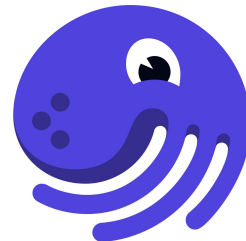
◆ Étapes pour override la configuration

- 1 Aller dans **Dagster Webserver**
- 2 Sélectionner un **Job**
- 3 Cliquer sur "**Launchpad**"
- 4 Modifier la **configuration YAML** et exécuter



Permet d'adapter rapidement un pipeline sans toucher au code source !

1. Concepts clés



Utilisation de ConfigSchema pour les paramètres des ops

The screenshot displays the DVC web interface for a job named `job_using_config`. The interface is divided into several sections:

- Header:** Includes a search bar and navigation links for `Runs`, `Assets`, `Status`, and `Workspace`.
- Job Information:** Shows the job name `job_using_config` and its location `repo@config...resource.py`.
- Overview / Launchpad / Runs:** The `Launchpad` tab is active, showing a `New Run` button and a `+ Add...` button.
- Configuration Editor:** A code editor showing the configuration for the job. The left pane contains the `ops` configuration, and the right pane shows the `config` schema.
- Configuration Schema:** The schema defines the `ops` configuration with a `config` section containing a `person_name` field of type `String`.
- Configuration Actions:** A section at the bottom left showing the status of the configuration. It indicates that there are no missing configurations and no extra configurations to remove.
- Runtime and Resources:** A section at the bottom right showing the runtime and resources for the job. It includes a `Launch Run` button.

```
ops:
  op_using_config:
    config:
      person_name: Alice
```

```
{
  person_name: String
}
```

Use Ctrl+Space to show auto-completions inline.

ERRORS
No errors

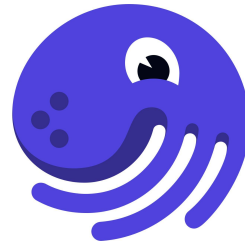
CONFIG ACTIONS:
No missing config
No extra config to remove

RUNTIME
execution
loggers
io_manager

OPS
op_using_config

Launch Run

1. Concepts clés



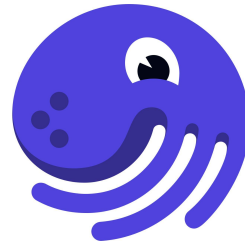
Exécution Conditionnelle avec **DynamicOutput** et **DynamicPartitions**

- ◆ Pourquoi utiliser une exécution conditionnelle ?
 - Exécuter des branches différentes en fonction d'un paramètre
 - Générer **dynamiquement des partitions** en fonction des données
 - Optimiser l'exécution en **traitant uniquement ce qui est nécessaire**

```
from dagster import DynamicOutput, op

@op
def generate_numbers():
    for i in range(3):
        yield DynamicOutput(i, mapping_key=str(i))
```

1. Concepts clés



Exécution Conditionnelle avec **DynamicOutput** et **DynamicPartitions**

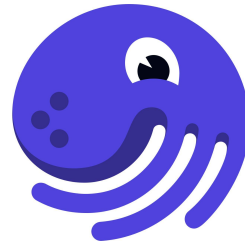
```
from dagster import DynamicPartitionsDefinition

my_partitions = DynamicPartitionsDefinition(name="dates")

@op
def process_partitioned_data(partition_key: str):
    return f"Processing data for {partition_key}"
```

Les partitions sont créées à la volée selon les besoins du job.

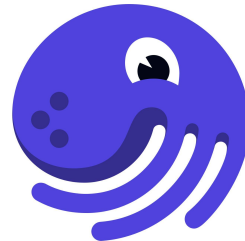
1. Concepts clés



Optimiser la Configuration des Pipelines

- ✓ **ConfigSchema** : Structure les paramètres des `ops`
- ✓ **Variables dynamiques** : Permettent une exécution flexible
- ✓ **dagster.yaml** : Standardise la gestion des environnements
- ✓ **Override via Dagster Webserver** : Facilite les tests et ajustements
- ✓ **Exécution conditionnelle (`DynamicOutput`, `DynamicPartitions`)** : Rend les pipelines plus intelligents et efficaces

1. Concepts clés



Structuration des Dépendances entre Ops

- ◆ Pourquoi structurer les dépendances ?
 - Définir l'ordre d'exécution des ops
 - Optimiser l'exécution en ne lançant que les tâches nécessaires
 - Faciliter le debugging en visualisant le graphe des dépendances
- ◆ Définition des entrées et sorties (**ins**, **out**)
 - **ins**: définit les dépendances en entrée
 - **out**: définit les sorties d'un **op**

process_data attend le résultat de **fetch_data** avant de s'exécuter.

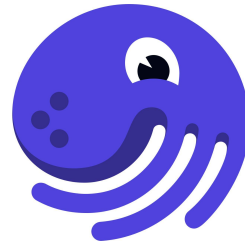
```
from dagster import op, job

@op
def fetch_data():
    return {"value": 42}

@op
def process_data(data):
    return data["value"] * 2

@job
def pipeline():
    process_data(fetch_data())
```

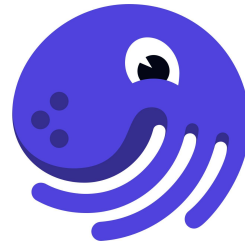
1. Concepts clés



Exécution Conditionnelle dans Dagster

- ◆ Pourquoi exécuter conditionnellement un op ?
 - Éviter d'exécuter des tâches inutiles (optimisation)
 - Gérer les workflows dynamiques (ex : validation, transformation conditionnelle)
- ◆ Utilisation des **Optional Types** pour une exécution conditionnelle

1. Concepts clés



Exécution Conditionnelle dans Dagster

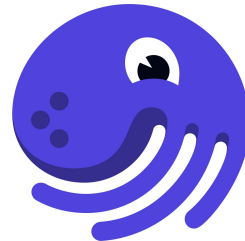
Si `maybe_return_value` renvoie `None`, `process_value` s'adapte.

```
from typing import Optional
from dagster import op

@op
def maybe_return_value():
    return 42 if True else None # Simule une condition

@op
def process_value(value: Optional[int]):
    if value is None:
        return "No value provided"
    return f"Processed value: {value * 2}"
```

1. Concepts clés



Utilisation des Hooks pour Gérer des Événements

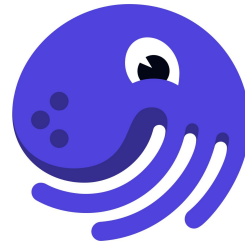
◆ Pourquoi utiliser des hooks ?

- Exécuter des actions **automatiques** avant/après un **op**
- **Envoyer des notifications** en cas de succès ou d'échec
- **Déclencher des audits et des logs** pour améliorer le monitoring

◆ Types de hooks dans Dagster

- **SuccessHook** : Exécute une action si un **op** réussit
- **FailureHook** : Exécute une action en cas d'échec
- **EventMetadataEntry** : Ajoute des métadonnées dans les logs

1. Concepts clés



Lorsqu'un op échoue, une alerte est envoyée (ex : email, Slack).

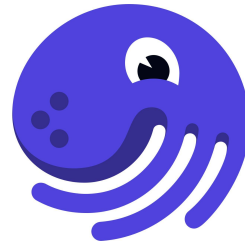
```
from dagster import op, failure_hook

@failure_hook
def notify_failure(context):
    print(f"Op {context.op.name} failed!")

@op
def risky_op():
    raise Exception("Something went wrong!")

@job
def monitored_job():
    risky_op()
```

1. Concepts clés



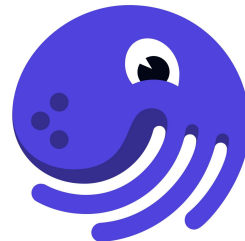
Retrying et Gestion des Erreurs avec **RetryPolicy**

- ◆ Pourquoi gérer les erreurs et retries ?
 - Certaines erreurs sont **temporaires** (ex : API indisponible)
 - **Réessayer l'exécution** au lieu d'échouer directement
 - Définir une **stratégie de gestion des erreurs**
- ◆ Utilisation de **RetryPolicy** pour automatiser les retries

```
from dagster import RetryPolicy, op

@op(retry_policy=RetryPolicy(max_retries=3, delay=2))
def unstable_task():
    if random.choice([True, False]):
        raise Exception("Temporary failure")
    return "Success"
```

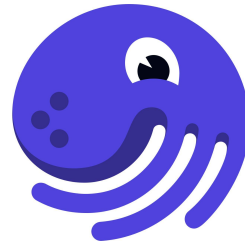
1. Concepts clés



Optimisation des Dépendances et Exécutions Conditionnelles

- ✓ Structurer les dépendances (**ins**, **out**) pour organiser le workflow
- ✓ Gérer les exécutions conditionnelles avec **Optional**, **if/else**
- ✓ Automatiser les notifications et logs avec **Hooks**
- ✓ Optimiser la fiabilité des pipelines avec **RetryPolicy**

1. Concepts clés



Introduction aux Partitions

♦ Pourquoi utiliser des partitions ?

- Diviser un job en **sous-exécutions indépendantes**
- Traiter uniquement les **données nouvelles ou modifiées**
- Optimiser l'utilisation des ressources en évitant les traitements inutiles

♦ Exemples d'utilisation

- Traitement quotidien d'une **table de base de données**
- Génération de rapports **mensuels**
- Mise à jour incrémentale d'un modèle **Machine Learning**

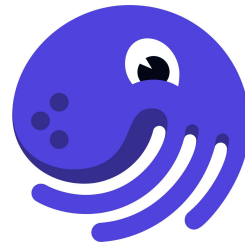
♦ Définition d'une partition dans Dagster

- **Partition temporelle** : découpée par jour, heure, semaine...
- **Partition personnalisée** : selon une liste définie (ex : régions, catégories)



Les partitions permettent de rendre les pipelines plus scalables et efficaces.

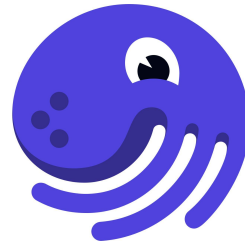
1. Concepts clés



Partitions Temporelles

- ◆ Pourquoi utiliser des partitions temporelles ?
 - Exécuter un job **régulièrement** (quotidien, horaire...)
 - Faciliter la **reprise après échec** en ne rejouant que les dates concernées
 - Simplifier l'**historisation et la gestion des données**

1. Concepts clés



Définition d'une partition quotidienne

```
from dagster import DailyPartitionsDefinition, asset

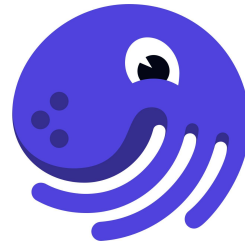
@asset(partitions_def=DailyPartitionsDefinition(start_date="2024-01-01"))
def daily_report():
    return f"Report for {AssetKey.date_str}"
```

💡 Chaque exécution ne traite que la partition du jour !

♦ Autres types de partitions temporelles

- **HourlyPartitionsDefinition** (par heure)
- **WeeklyPartitionsDefinition** (par semaine)

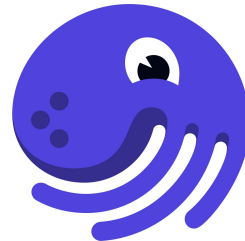
1. Concepts clés



Partitions Personnalisées et Cas Avancés

- ◆ Pourquoi créer des partitions personnalisées ?
 - Adapter les partitions **aux besoins métiers**
 - Exécuter des jobs **par région, client, type de données...**
 - Optimiser l'exécution en ne traitant **que les segments concernés**

1. Concepts clés



Partitions Personnalisées et Cas Avancés

```
from dagster import StaticPartitionsDefinition, asset

regions = StaticPartitionsDefinition(["Europe", "US", "Asia"])

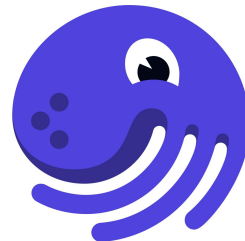
@asset(partitions_def=regions)
def sales_report():
    return f"Report for {AssetKey.region}"
```

💡 Chaque exécution génère un rapport pour une région spécifique.

◆ Cas d'usage avancés

- Traiter les données **par client ou produit**
- Partitionner par **année ou trimestre** pour les analyses financières
- Intégration avec des systèmes externes (S3, bases SQL)

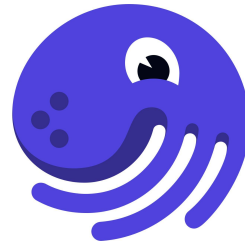
1. Concepts clés



Définition des Schedules et Exécution Automatique

- ◆ Pourquoi utiliser un scheduler ?
 - Planifier l'exécution des jobs à **intervalles réguliers**
 - Éviter d'avoir à lancer les workflows **manuellement**
 - Automatiser l'**ingestion et le traitement des données**

1. Concepts clés



Définition des Schedules et Exécution Automatique

- ◆ Création d'un scheduler avec `@schedule`

```
from dagster import schedule

@schedule(cron_schedule="0 6 * * *", job=my_job)
def daily_job_schedule():
    return {}
```

Ce job s'exécutera automatiquement chaque jour à 6h du matin !

- ◆ Configuration des Schedules

- Basé sur un **cron schedule** ("0 6 * * *" → tous les jours à 6h)
- Peut être défini sur **des partitions spécifiques**
- Compatible avec **Dagster Daemon** pour l'exécution continue

1. Concepts clés



Sensors – Déclenchement Auto. Basé sur des Événements

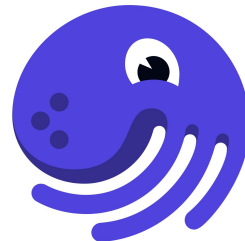
◆ Différence entre **Schedule** et **Sensor**

- **Schedule** → exécute un job à **une heure précise**
- **Sensor** → exécute un job **quand un événement se produit**

◆ Pourquoi utiliser un **Sensor** ?

- Déclencher un job **quand un fichier est ajouté à un dossier**
- Lancer un traitement **quand une table SQL est mise à jour**
- Réagir dynamiquement à **l'arrivée de nouvelles données**

1. Concepts clés



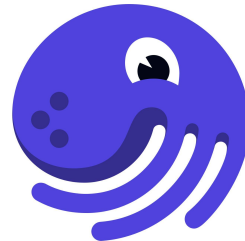
Sensors – Déclenchement Automatique Basé sur des Événemen

```
from dagster import sensor, RunRequest

@sensor(job=my_job)
def file_sensor(context):
    if check_new_file("/data/input/"):
        return RunRequest(run_key="new_file_detected")
```

Ce job ne s'exécutera que lorsqu'un nouveau fichier sera détecté !

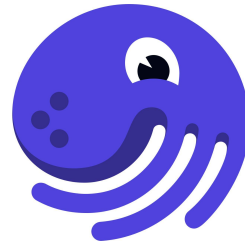
1. Concepts clés



Optimiser la Planification des Jobs

- ✓ **Utiliser les partitions** pour traiter uniquement les nouvelles données
- ✓ **Planifier les exécutions** avec `@schedule` pour automatiser les workflows
- ✓ **Utiliser des sensors** pour déclencher un job **uniquement quand c'est nécessaire**
- ✓ **Combiner partitions et planification** pour optimiser la charge de travail

1. Concepts clés



Introduction aux Ressources (@resource)

◆ Qu'est-ce qu'une Resource dans Dagster ?

- Une ressource (@resource) est un **service externe** utilisé par les **ops** et **jobs**
- Peut être une **base de données, une API, un stockage cloud...**
- Permet de centraliser **les connexions et les configurations**

◆ Pourquoi utiliser les Resources ?

- ✓ Réutilisation des connexions **sans duplication**
- ✓ Gestion dynamique des **identifiants et secrets**
- ✓ Amélioration de la **modularité et du testabilité**

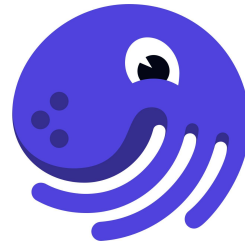
```
from dagster import resource

@resource
def database_connection():
    import sqlite3
    return sqlite3.connect("data.db")
```



Cette ressource est injectée dans les **ops** pour centraliser l'accès à la base de données.

1. Concepts clés



Connexion aux Bases de Données, APIs et Stockage Cloud

◆ Les types de ressources les plus courants

- **Bases de données** : PostgreSQL, MySQL, MongoDB...
- **APIs externes** : Services REST, GraphQL, AWS, OpenAI...
- **Stockage cloud** : S3, Google Cloud Storage, Azure Blob

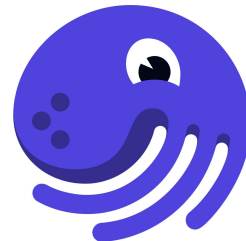
◆ Passage de configuration dynamique aux ressources

```
from dagster import resource, Config

class APIConfig(Config):
    api_key: str

@resource(config_schema=APIConfig)
def api_client(context):
    return f"https://api.example.com?key={context.resource_config['api_key']}"
```

1. Concepts clés



Introduction aux IOManagers

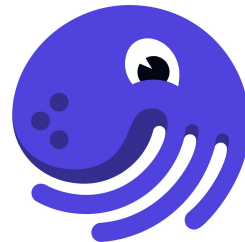
◆ Qu'est-ce qu'un IOManager ?

- Un IOManager gère **l'écriture et la lecture des outputs** entre les **ops**
- Permet de **persister les résultats** d'un job (mémoire, fichier, base de données...)
- Sépare la logique métier du stockage

◆ Types d'IOManagers intégrés

IOManager	Stockage	Utilisation
<i>mem_io_manager</i>	Mémoire vive	Données temporaires
<i>fs_io_manager</i>	Système de fichiers	Sauvegarde locale
<i>db_io_manager</i>	Base de données	Stockage SQL des résultats

1. Concepts clés



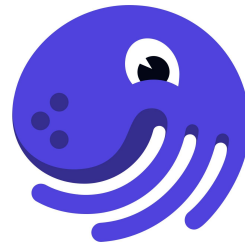
Introduction aux IOManagers

💡 Les résultats seront stockés sur le disque au lieu d'être perdus après exécution.

```
from dagster import fs_io_manager

@job(resource_defs={"io_manager": fs_io_manager})
def my_pipeline():
    process_data()
```

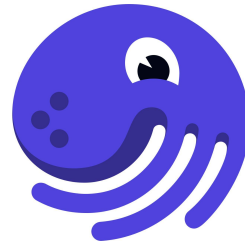
1. Concepts clés



Création d'un IOManager Personnalisé

- ◆ Pourquoi créer un IOManager personnalisé ?
 - ✓ Sauvegarder les résultats dans **un stockage spécifique** (ex : S3, BigQuery)
 - ✓ Adapter la logique de lecture/écriture aux **besoins métier**
 - ✓ Gérer **les formats spécifiques** (Parquet, JSON, etc.)

1. Concepts clés



💡 Les données des jobs sont automatiquement stockées et récupérées depuis AWS S3.

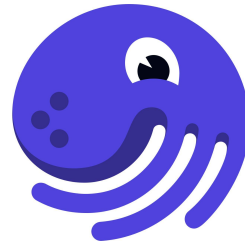
```
from dagster import IOManager
import boto3

class S3IOManager(IOManager):
    def handle_output(self, context, obj):
        s3 = boto3.client("s3")
        s3.put_object(Bucket="my-bucket", Key=context.step_key, Body=str(obj))

    def load_input(self, context):
        s3 = boto3.client("s3")
        response = s3.get_object(Bucket="my-bucket", Key=context.upstream_output.step_key)
        return response['Body'].read()

@io_manager
def s3_io_manager():
    return S3IOManager()
```

1. Concepts clés



Optimisation de la Gestion des Données avec Dagster

- ✓ Les **@resource** facilitent l'accès aux services externes (DB, API, stockage)
- ✓ Les **IOManagers** permettent de persister les résultats et d'améliorer la scalabilité
- ✓ Utiliser un **IOManager personnalisé** permet d'adapter le stockage aux besoins métiers
- ✓ Combiner **Resources** et **IOManagers** assure un pipeline robuste et flexible

2. Concepts clés



3. Bonnes pratiques

3. Bonnes pratiques

Gestion des Logs dans Dagster (**dagster-log**)

- ◆ Pourquoi surveiller les exécutions ?
 - Identifier les erreurs **rapidement**
 - Comprendre l'**ordre et le temps d'exécution** des **ops**
 - Faciliter le **debugging** et l'optimisation des workflows

3. Bonnes pratiques

Gestion des Logs dans Dagster (**dagster-log**)

Le **context.log.info()** permet d'enregistrer des événements dans les logs.

◆ Types de logs dans Dagster

- **INFO** : Information générale sur l'exécution
- **WARNING** : Avertissement sur une anomalie possible
- **ERROR** : Erreur empêchant l'exécution

```
from dagster import job, op

@op
def my_task(context):
    context.log.info("Op en cours d'exécution...")
    return "Résultat traité"

@job
def my_job():
    my_task()
```

3. Bonnes pratiques

Gestion des Logs dans Dagster (dagster-log)

The screenshot displays the Dagster web interface for a successful run of 'demo_job' (ID: 219c7b51). The top navigation bar includes 'Overview', 'Runs', 'Assets', and 'Deployment'. The run status is 'Success' with a duration of 4.567s. The main timeline shows the execution of the 'hello_logs' step, which is highlighted in green. The right sidebar provides a summary of the run's status, showing 'Preparing (0)', 'Executing (0)', 'Errored (0)', and 'Succeeded (1)' steps. The 'Succeeded (1)' section lists the 'hello_logs' step with a duration of 0.065s. Below the timeline, a search bar is set to 'step:hello_logs', and the 'Hide non-matches' checkbox is checked. The log table below shows the following events:

TIMESTAMP	OP	EVENT TYPE	INFO
12:07:26.643	hello_logs	STEP_WORKER_STARTL...	Launching subprocess for "hello_logs".
12:07:30.364	hello_logs	STEP_WORKER_STARTED	Executing step "hello_logs" in subprocess.
12:07:30.424	hello_logs	RESOURCE_INIT_STAR...	Starting initialization of resources [io_manager].
12:07:30.437	hello_logs	RESOURCE_INIT_SUCC...	Finished initialization of resources [io_manager].
12:07:30.508	hello_logs	STEP_START	Started execution of step "hello_logs".
12:07:30.524	hello_logs	INFO	Hello, world!
12:07:30.535	hello_logs	STEP_OUTPUT	Yielded output "result" of type "Any". (Type check passed).
12:07:30.562	hello_logs	HANDLED_OUTPUT	Handled output "result" using IO manager "io_manager" path: /Users/erincochran/Desktop/dagster-examples/project-dagster-university/tmpzls_rf84/storage/219c7b51-b62f-4e5b-8de8-0e7a616b961c/hello_logs/result
12:07:30.573	hello_logs	STEP_SUCCESS	Finished execution of step "hello_logs" in 49ms.

3. Bonnes pratiques

Enrichissement des Logs avec EventMetadata

Fournir des **informations détaillées** sur chaque événement

Faciliter la **recherche et l'analyse des logs**

Enrichir les logs avec des **graphiques, liens ou métriques**

```
from dagster import AssetMaterialization, MetadataValue

@op
def process_data(context):
    context.log.info(
        "Traitement terminé",
        extra={
            "métrique": MetadataValue.float(99.5),
            "lien_resultat": MetadataValue.url("https://example.com/report"),
        }
    )
```

3. Bonnes pratiques

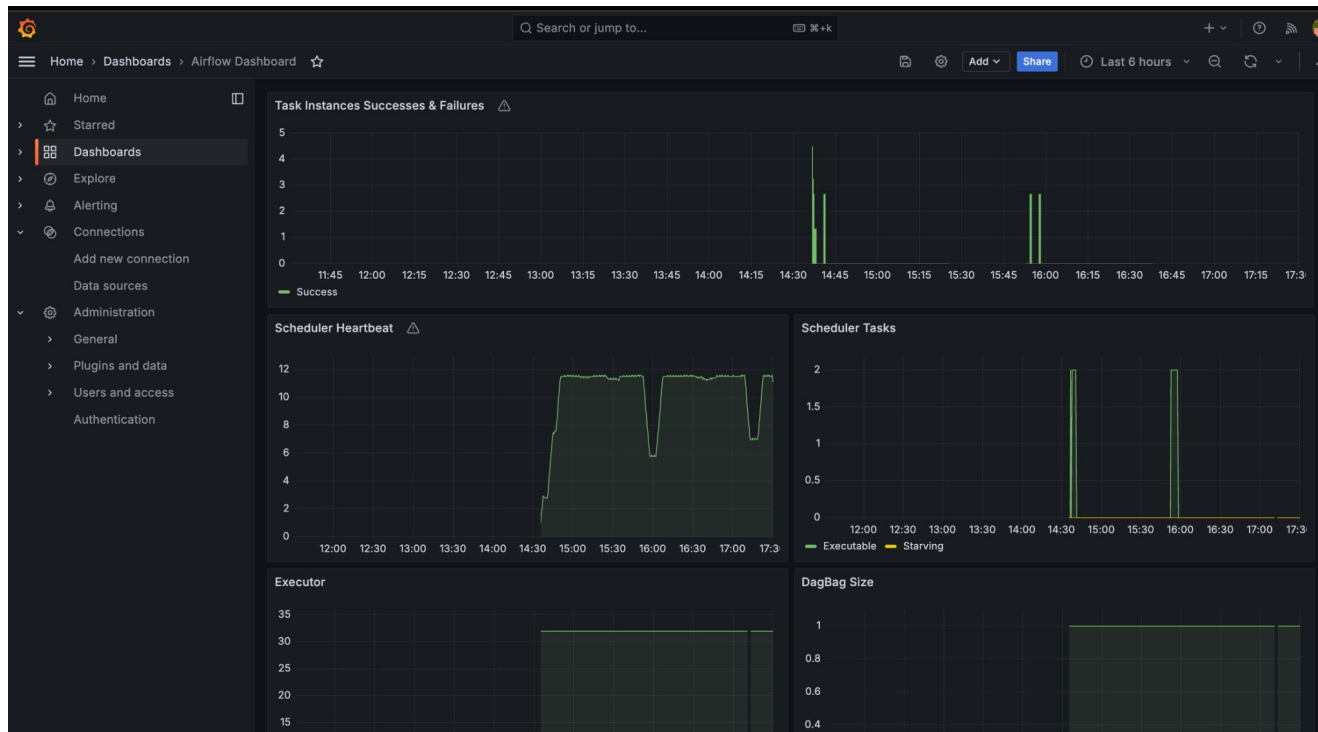
Intégration avec des Outils de Monitoring (Prometheus, Grafana...)

- ◆ Pourquoi connecter Dagster à des outils de monitoring ?
 - Surveiller l'état des jobs en temps réel
 - Détecter les anomalies et latences
 - Avoir des dashboards graphiques pour mieux analyser les performances

Outil	Usage
Prometheus	Collecte de métriques sur les exécutions
Grafana	Visualisation des performances et logs
Elasticsearch + Kibana	Analyse et recherche avancée des logs
AWS CloudWatch / GCP Stackdriver	Monitoring Cloud des jobs Dagster

3. Bonnes pratiques

Intégration avec des Outils de Monitoring (Prometheus, Grafana...)



3. Bonnes pratiques

Gestion des Erreurs et Ré exécutions Automatiques

- ◆ Pourquoi automatiser la gestion des erreurs ?
 - Redémarrer automatiquement les jobs en cas d'échec
 - Notifier les équipes **immédiatement** en cas de problème
 - Éviter que des erreurs **mineures bloquent un pipeline entier**

```
from dagster import RetryPolicy, op

@op(retry_policy=RetryPolicy(max_retries=3, delay=5))
def unstable_task():
    raise Exception("Échec temporaire, à réessayer...")
```

3. Bonnes pratiques

```
from dagster import failure_hook, success_hook

@failure_hook
def notify_failure(context):
    print(f"💣 Op {context.op.name} a échoué !")

@success_hook
def notify_success(context):
    print(f"✅ Op {context.op.name} exécuté avec succès !")
```


3. Bonnes pratiques

```
from dagster import failure_hook, success_hook

@failure_hook
def notify_failure(context):
    print(f"💣 Op {context.op.name} a échoué !")

@success_hook
def notify_success(context):
    print(f"✅ Op {context.op.name} exécuté avec succès !")
```

3. Bonnes pratiques

Optimisation de la Surveillance des Pipelines

- ✓ Les logs (**dagster-log**) facilitent le suivi des exécutions
- ✓ Les **EventMetadata** enrichissent les logs avec des métriques et liens
- ✓ Les outils externes (Prometheus, Grafana) offrent un monitoring avancé
- ✓ Les **RetryPolicy** et **Hooks** assurent une meilleure gestion des erreurs

3. Bonnes pratiques

Comment structurer tout ça ?!

Une Structure Modulable et Scalable

✅ **Modularité** : Chaque composant du projet (assets, jobs, partitions, ressources, capteurs) est isolé dans son propre dossier, facilitant la maintenance et l'évolution du code.

✅ **Réutilisabilité** : Les assets et les ressources sont centralisés, permettant leur réutilisation dans plusieurs jobs sans duplication de code.

✅ **Clarté** : Une séparation nette entre les fichiers de configuration (`pyproject.toml`, `setup.py`), les données (`data/`), et la logique Dagster (`dagster_university/`).

✅ **Scalabilité** : Cette structure permet d'ajouter facilement de nouveaux assets, capteurs ou partitions sans perturber l'architecture existante.

```
.
├── README.md
├── dagster_university/
│   ├── assets/
│   │   ├── __init__.py
│   │   ├── constants.py
│   │   ├── metrics.py
│   │   └── trips.py
│   ├── jobs/
│   ├── partitions/
│   ├── resources/
│   ├── schedules/
│   ├── sensors/
│   └── __init__.py
├── dagster_university_tests
├── data/
│   ├── outputs/
│   ├── raw/
│   ├── requests/
│   │   └── README.md
│   └── staging/
├── .env
├── .env.example
├── pyproject.toml
├── setup.cfg
└── setup.py
```

3. Bonnes pratiques

Comment structurer tout ça ?!

Comment cette structure améliore l'orchestration ?

- **Séparation des responsabilités :**
 - `assets/` : Contient les définitions des actifs de données (ex. `trips.py`, `metrics.py`).
 - `jobs/` : Définit les ensembles d'actifs à exécuter ensemble.
 - `sensors/` : Surveille les événements déclencheurs et exécute des jobs automatiquement.
- **Gestion efficace des dépendances :**
 - `partitions/` permet une gestion optimisée des données par plages temporelles
 - `resources/` regroupe les connexions aux bases de données et services externes évitant la redondance.
- **Traçabilité et monitoring améliorés :**
 - `data/outputs/` stocke les résultats, facilitant l'audit des pipelines.
 - `dagster_university_tests/` prépare un cadre pour tester les assets et pipelines.

```
.
├── README.md
├── dagster_university/
│   ├── assets/
│   │   ├── __init__.py
│   │   ├── constants.py
│   │   ├── metrics.py
│   │   └── trips.py
│   ├── jobs/
│   ├── partitions/
│   ├── resources/
│   ├── schedules/
│   ├── sensors/
│   └── __init__.py
├── dagster_university_tests
├── data/
│   ├── outputs/
│   ├── raw/
│   ├── requests/
│   │   └── README.md
│   └── staging/
├── .env
├── .env.example
├── pyproject.toml
├── setup.cfg
└── setup.py
```

3. Bonnes pratiques



4. Examples

4. Exemples

RDV sur le GitHub du cours

4. Exemples



