Chapitre 2bis:

Outils incontournables et notion de conteneurisation

4DATA

Benjamin Jurczak

SUPINFO

Sommaire

1. Introduction à la conteneurisation
2. Conteneurisation et pipelines de données
3. Industrialisation et sécurisation des pipelines conteneurisés

1. Introduction à la conteneurisation

1. Introduction à la conteneurisation

Conteneurisation ??

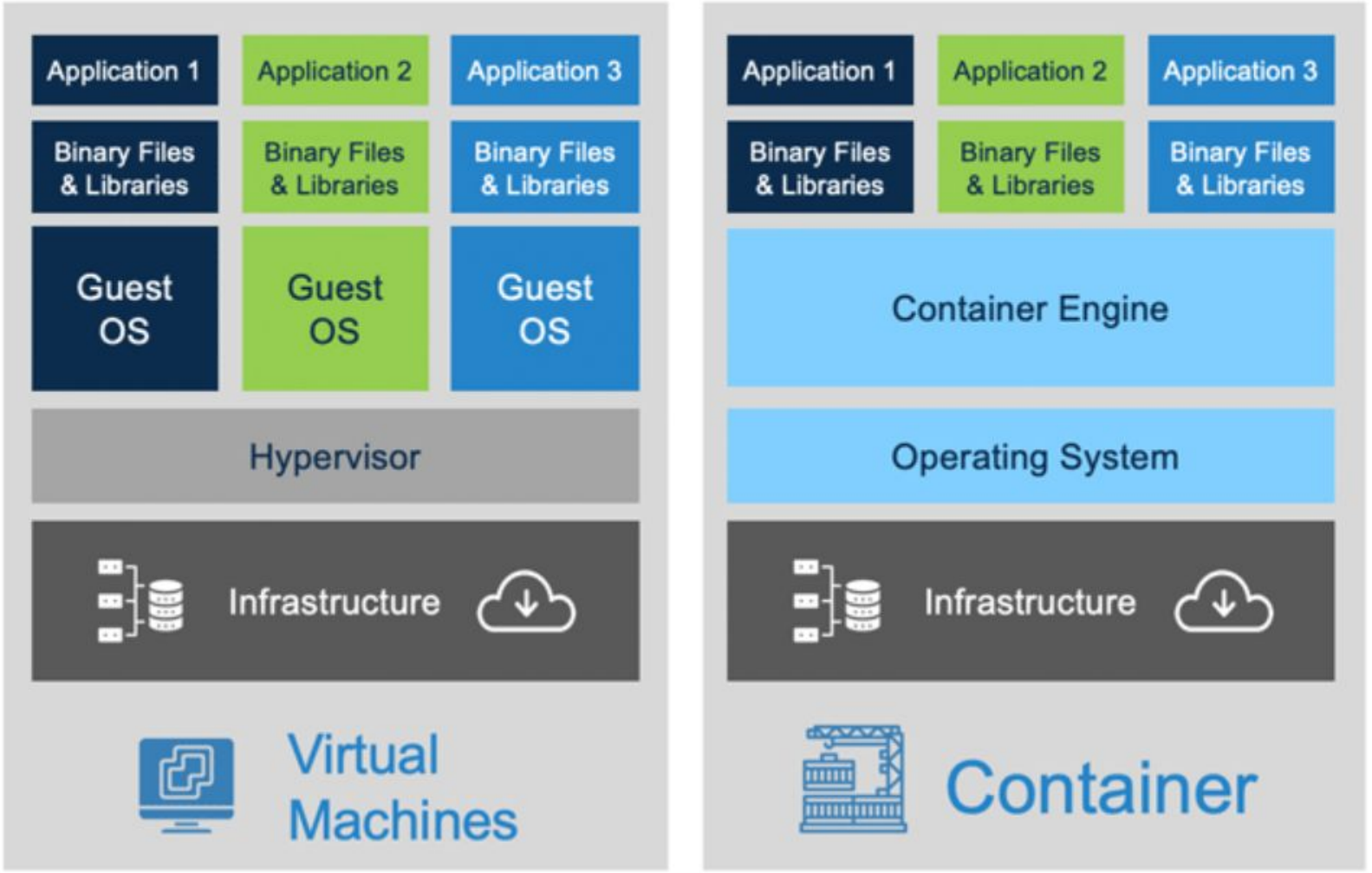
- La conteneurisation est une technologie permettant d'encapsuler une application et ses dépendances dans un environnement isolé et portable.

- Contrairement aux machines virtuelles (VM), les conteneurs partagent le même noyau du système d’exploitation, ce qui les rend plus légers et rapides.

- Très utilisée dans le développement et la gestion des pipelines de données pour garantir la portabilité et la scalabilité.

1. Introduction à la conteneurisation

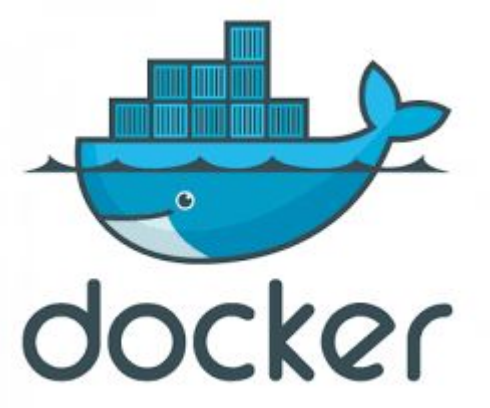
VM vs Containers



1. Introduction à la conteneurisation

Qu’est-ce que la conteneurisation ?

- **Définition** : Processus d'encapsulation d’une application et de ses dépendances dans un conteneur portable et reproductible.
- **Principe** : Basé sur l’isolation, chaque conteneur exécute une application dans un environnement isolé.
- **Exemples d’outils** : Docker, Podman, LXC, Kubernetes (orchestration).



1. Introduction à la conteneurisation

Avantages de la Conteneurisation

- **Léger et rapide** : Moins gourmand en ressources que les VM.
- **Portabilité** : Fonctionne sur n'importe quelle machine avec le même comportement.
- **Scalabilité** : Facilement déployable et orchestrable (ex: Kubernetes).
- **Isolation** : Empêche les conflits de dépendances entre applications.
- **Automatisation** : Compatible avec CI/CD pour l'intégration et le déploiement.

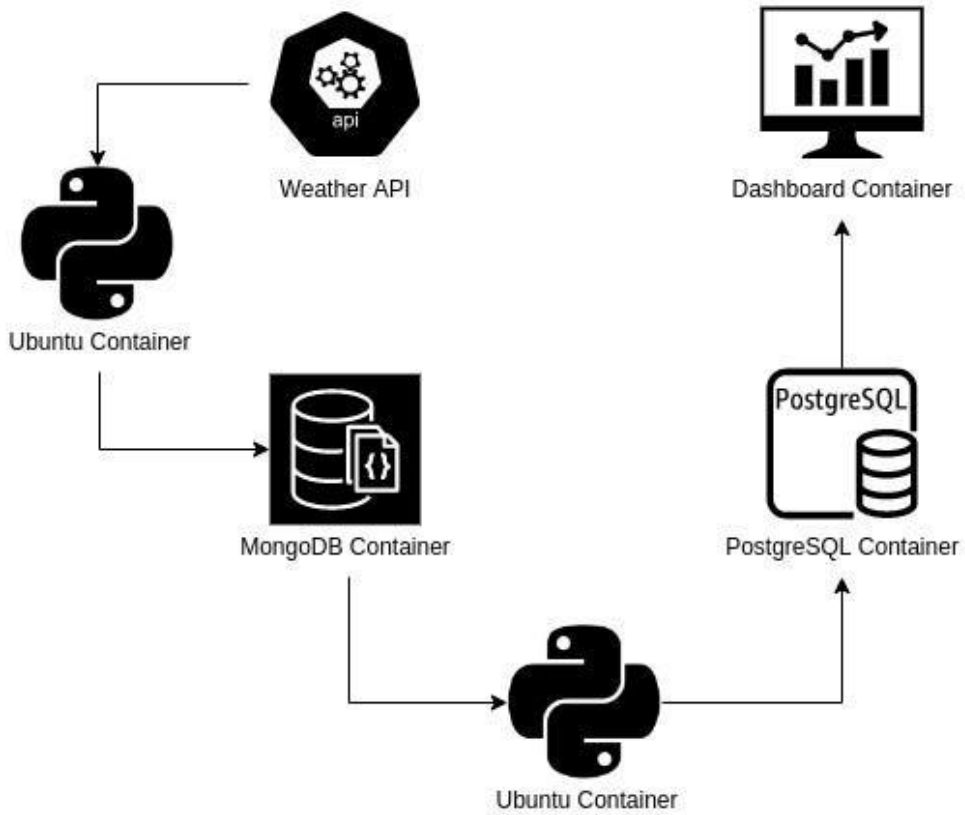
1. Introduction à la conteneurisation

Cas d’usage dans les pipelines de données

- **Déploiement des bases de données** : PostgreSQL, MySQL en conteneurs pour éviter l’installation manuelle.
- **Conteneurisation des ETL** : Exécution d’Airflow, Dagster ou dbt dans des conteneurs.
- **Traitement distribué** : Scalabilité accrue pour les jobs de transformation de données.
- **Facilité d'intégration** : CI/CD pour tester et déployer des pipelines en production.

1. Introduction à la conteneurisation

Cas d’usage dans les pipelines de données



1. Introduction à la conteneurisation

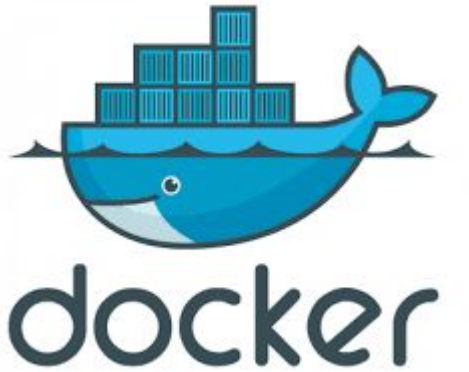
Introduction aux outils de conteneurisation

Les principaux outils de conteneurisation

Plusieurs outils existent pour la création et la gestion des conteneurs.

- **Docker** : Outil le plus répandu, facilitant la gestion des conteneurs.
- **Podman, LXC** : Alternatives à Docker sans daemon central.
- **Kubernetes** : Plateforme d’orchestration de conteneurs à grande échelle.

1. Introduction à la conteneurisation



Introduction aux outils de conteneurisation

Docker - Fonctionnement et Principes de Base

Docker - Le leader du marché

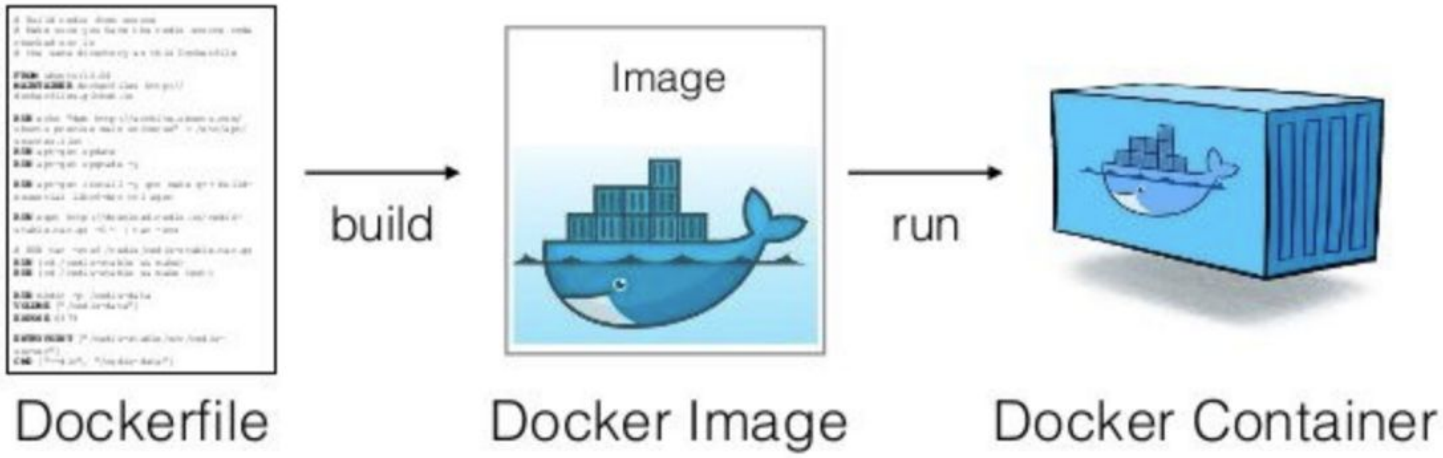
- **Qu'est-ce que Docker ?**
  - Plateforme permettant de créer, exécuter et déployer des conteneurs.
  - Fonctionne sur Windows, Linux et macOS.
- **Concepts Clés :**
  - **Images Docker** : Modèles immuables de conteneurs.
  - **Conteneurs** : Instances exécutables d'une image Docker.
  - **Dockerfile** : Script définissant la construction d'une image.
  - **Docker Hub** : Référentiel public d'images prêtes à l'emploi.

1. Introduction à la conteneurisation

Introduction aux outils de conteneurisation

Docker - Fonctionnement et Principes de Base

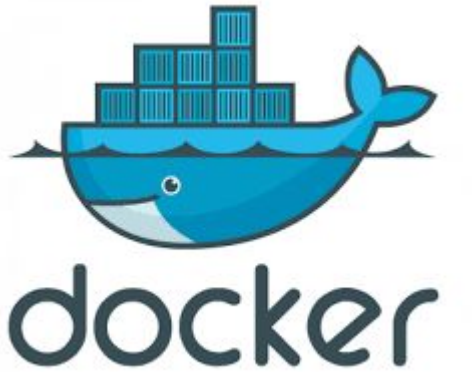
Docker - Le leader du marché



1. Introduction à la conteneurisation

Introduction aux outils de conteneurisation

Docker -Avantages et Inconvénients



- **Avantages :**
  - Portabilité des applications ("Build once, run anywhere").
  - Isolation des applications.
  - Facile à utiliser et bien documenté.
  - Intégration avec CI/CD et Kubernetes.
- **Inconvénients :**
  - Utilisation d'un daemon central (élévation des privilèges).
  - Gestion complexe des volumes et du stockage persistant.
  - Performance moindre que les solutions nativement intégrées à l’OS (LXC).

1. Introduction à la conteneurisation

Introduction aux outils de conteneurisation

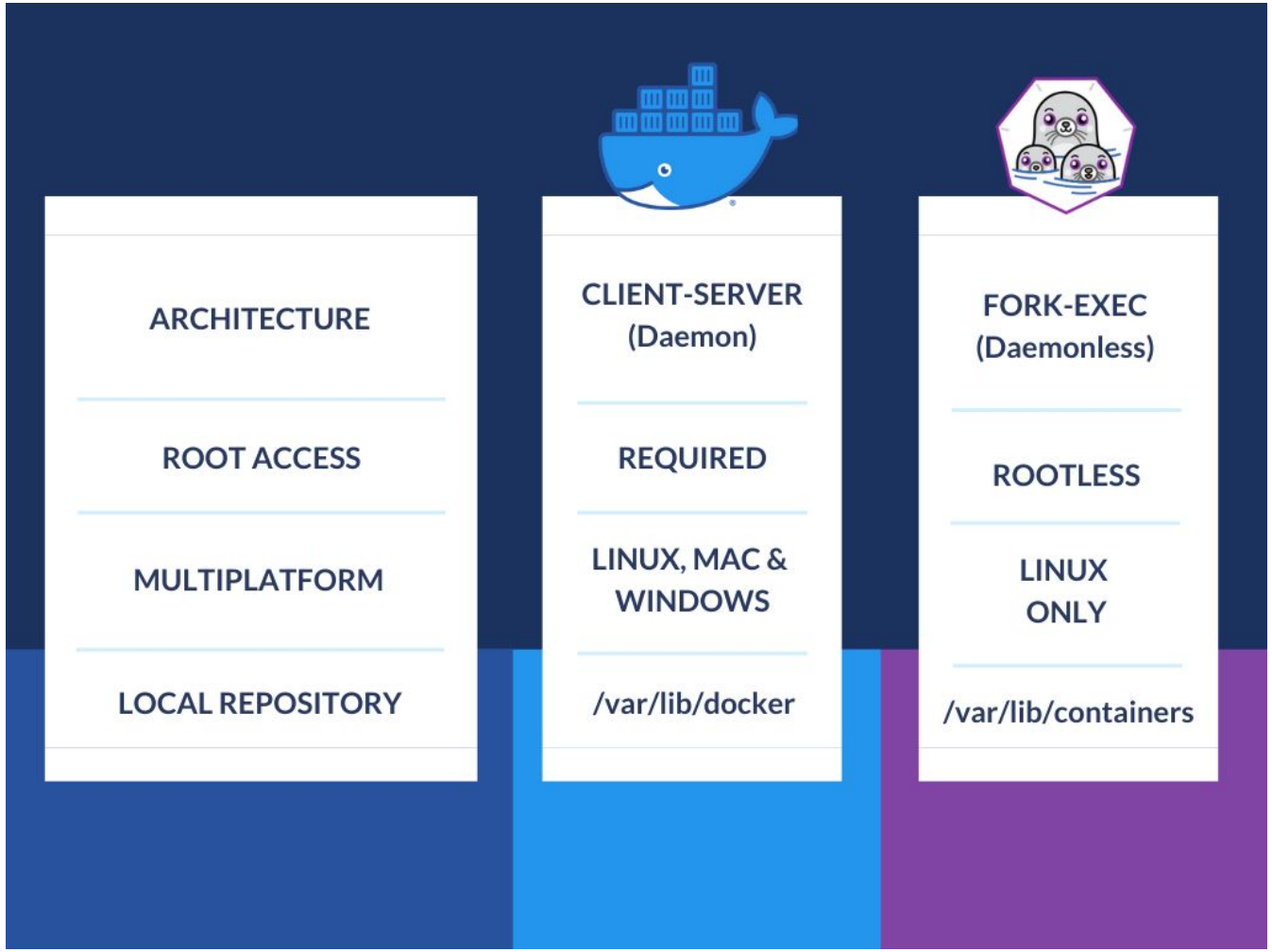
Alternatives à Docker : Podman et LXC

- **Podman** :
  - Fonctionne sans daemon (évite les problèmes de privilèges).
  - Compatible avec Docker (mêmes commandes et images).
  - Plus sûr pour des environnements critiques.
- **LXC (Linux Containers)** :
  - Conteneurisation plus proche de la virtualisation.
  - Utilisation plus avancée pour des systèmes complexes.

1. Introduction à la conteneurisation

Introduction aux outils de conteneurisation

Podman vs Docker



1. Introduction à la conteneurisation

Introduction aux outils de conteneurisation

Kubernetes - Gérer des conteneurs à grande échelle

- **Pourquoi Kubernetes ?**
  - Gère et automatise le déploiement de conteneurs à grande échelle.
  - Répartit la charge et réplique les applications selon les besoins.
- **Principaux composants :**
  - **Pods** : Groupes de conteneurs exécutés ensemble.
  - **Services** : Gèrent l'accès réseau et le load balancing.
  - **Deployments** : Assurent la gestion des mises à jour et du scaling.
- **Exemples d'utilisation :**
  - Gestion de pipelines de données avec des jobs conteneurisés.
  - Exécution de tâches de machine learning distribuées.

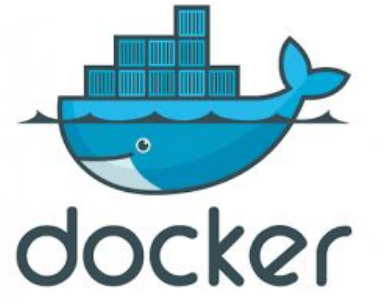
1. Introduction à la conteneurisation

Introduction aux outils de conteneurisation

Quel outil choisir pour quel besoin ?

- **Docker** : Idéal pour le développement et les tests locaux.
- **Podman** : Adapté aux environnements exigeant une sécurité renforcée.
- **LXC** : Pour une approche plus proche de la virtualisation.
- **Kubernetes** : Pour la gestion à grande échelle des conteneurs en production

Si un à retenir ? Docker !



1. Introduction à la conteneurisation

Introduction aux concepts fondamentaux

Comprendre les bases de la conteneurisation

Contenu de la section :

- Définition des concepts fondamentaux de la conteneurisation.
- Différence entre **images et conteneurs**.
- Gestion du **stockage** avec les volumes.
- Communication entre conteneurs via le **réseau**.
- **Docker Registry** et la gestion des images.

1. Introduction à la conteneurisation

Introduction aux concepts fondamentaux

Images et Conteneurs - Différences et fonctionnement

- **Images Docker** : Modèles immuables pour créer des conteneurs.
- **Conteneurs** : Instances exécutables basées sur des images.
- **Cycle de vie d'un conteneur** : build → run → stop → remove.
- Commandes principales :

```
docker build -t mon_image .
docker run -d --name mon_conteneur mon_image
docker ps
docker stop mon_conteneur
docker rm mon_conteneur
```

1. Introduction à la conteneurisation

Introduction aux concepts fondamentaux

Volumes et Stockage Persistant

Gérer les données dans les conteneurs

- Problème : Les conteneurs sont éphémères → besoin de persistance.
- **Volumes Docker** :
  - Gestion externe des fichiers et bases de données.
  - Partage des données entre plusieurs conteneurs.

```
docker volume create mon_volume
docker run -d -v mon_volume:/data mon_image
```

1. Introduction à la conteneurisation

Introduction aux concepts fondamentaux

Réseaux et Communication entre Conteneurs

- Docker utilise un réseau par défaut, mais permet la configuration avancée.
- **Types de réseaux Docker :**
  - **Bridge** : Communication entre conteneurs sur la même machine.
  - **Host** : Utilisation directe de l'interface réseau de l'hôte.
  - **Overlay** : Communication entre plusieurs machines (Swarm/K8s).

```
docker network create mon_reseau
docker run -d --net=mon_reseau --name conteneur1 mon_image
docker network ls
```

1. Introduction à la conteneurisation

Introduction aux concepts fondamentaux

Docker Registry et Gestion des Images

- **Docker Hub** : Référentiel public d'images.
- **Docker Registry privé** : Hébergement local ou cloud.
- **Principales commandes :**
  - **Pousser une image vers Docker Hub :**

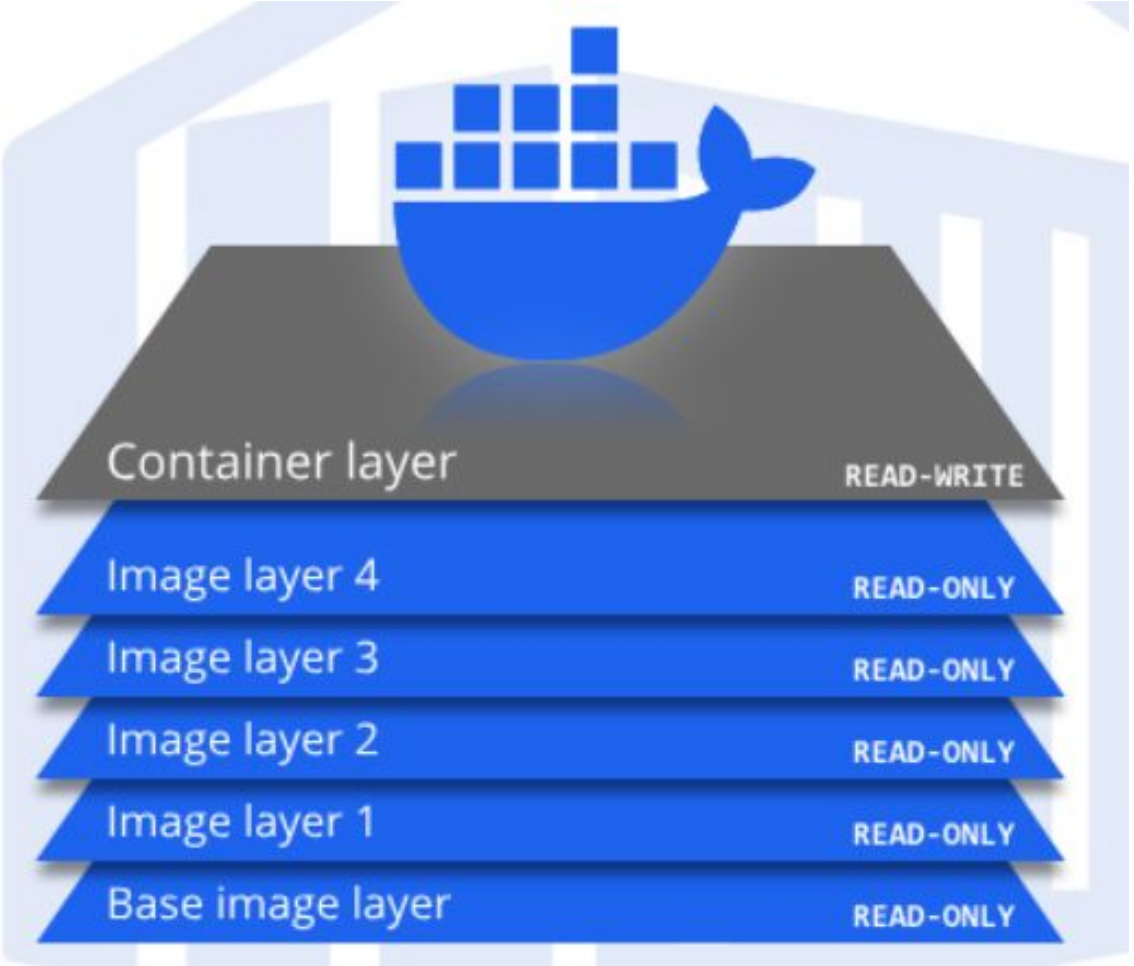
```
docker login
docker tag mon_image mon_utilisateur/mon_image:v1
docker push mon_utilisateur/mon_image:v1
```

  - **Créer un Docker Registry local :**

```
docker run -d -p 5000:5000 --name registry registry:2
docker tag mon_image localhost:5000/mon_image
docker push localhost:5000/mon_image
```

1. Introduction à la conteneurisation

Introduction aux concepts fondamentaux



2. Introduction à la conteneurisation

2. Conteneurisation et pipelines de données

2. Conteneurisation et pipelines de données

Introduction à la Création et Gestion des Images Docker

Pourquoi utiliser des images Docker ?

- Définition d’une image Docker : un modèle immuable contenant une application et ses dépendances.
- Importance des images pour la portabilité et la reproductibilité.
- Différence entre une image et un conteneur (une image est un modèle, un conteneur est une instance en cours d’exécution).
- Principales commandes :

```
docker images  # Lister les images disponibles
docker pull <image>  # Télécharger une image du Docker Hub
docker run -it <image>  # Exécuter un conteneur
```

2. Conteneurisation et pipelines de données

Structure d'un Dockerfile et Bonnes Pratiques

Comment écrire un Dockerfile efficace ?

- Définition : un Dockerfile est un script permettant de construire une image Docker.
- **Structure typique d'un Dockerfile :**

```
FROM python:3.9-slim  # Image de base légère
WORKDIR /app  # Définition du répertoire de travail
COPY requirements.txt .  # Copier les dépendances
RUN pip install -r requirements.txt  # dépendances
COPY . .  # Copier le reste du projet
CMD ["python", "app.py"]  # Commande d'exécution par défaut
```

Bonnes pratiques :

- Utiliser des images minimales (alpine, slim).
- Éviter les fichiers inutiles en utilisant .dockerignore.
- Grouper les instructions RUN pour minimiser le nombre de couches.

2. Conteneurisation et pipelines de données

Construction et Optimisation des Images Docker

Comment améliorer les performances des images ?

- **Utilisation du cache Docker :**
  - Ordre des instructions : copier et installer les dépendances avant d'ajouter le code.
  - Exemple optimisé :

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

2. Conteneurisation et pipelines de données

Construction et Optimisation des Images Docker

Comment améliorer les performances des images ?

Multi-stage builds :

- Réduction de la taille des images en séparant la construction et l'exécution.

```
FROM python:3.9 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

FROM python:3.9-slim
WORKDIR /app
COPY --from=builder /app .
CMD ["python", "app.py"]
```

Utiliser des utils d'analyse d'image (docker-slim) pour vérifier la taille et les couches.

2. Conteneurisation et pipelines de données

Sécurisation des Images Docker

Comment rendre une image Docker plus sécurisée ?

Vérification et scan des vulnérabilités :

```
trivy image mon_image
```

Utilisation de trivy, clair ou snyk pour analyser les vulnérabilités.

Utilisation de signatures numériques :

Signer et vérifier une image avec Docker Content Trust (DCT).

```
docker trust sign mon_image
docker trust inspect --pretty mon_image
```

Privilégier un utilisateur non root :

Limiter les permissions et éviter les accès non nécessaires.

```
FROM python:3.9-slim
RUN addgroup --system app && adduser --system --group app
USER app
```

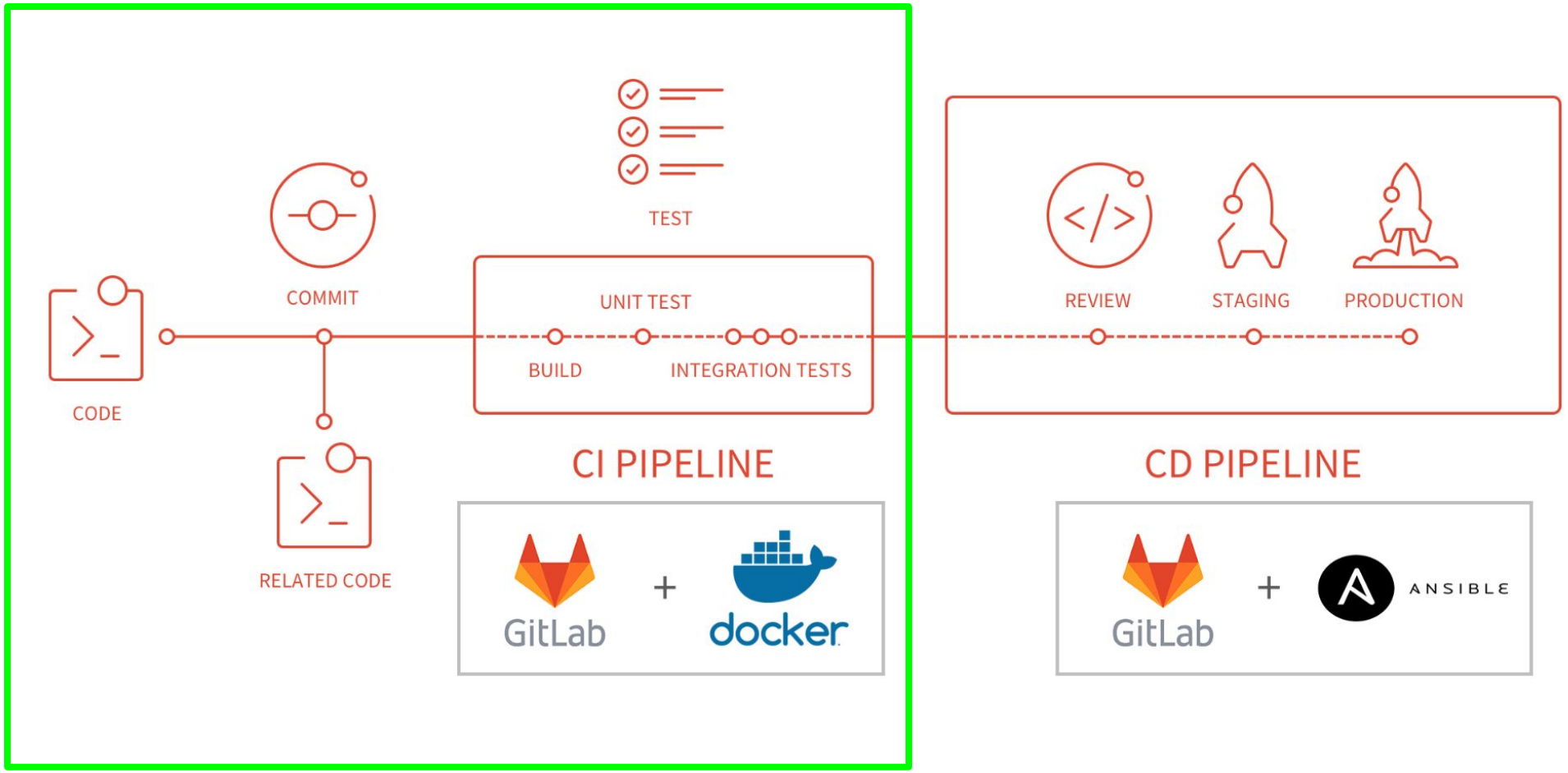
2. Conteneurisation et pipelines de données

Gestion des Mises à Jour et Distribution des Images

- **Stockage des images sur Docker Hub ou registre privé (Harbor, Nexus, AWS ECR) :**
- **Gestion des versions et bonnes pratiques :**
  - Versionner les images (:v1, :latest, :stable).
  - Nettoyer les images obsolètes (docker rmi).
  - Éviter d’utiliser latest en production.
- **Déploiement automatisé via CI/CD avec des pipelines Docker.**

2. Conteneurisation et pipelines de données

Gestion des Mises à Jour et Distribution des Images



2. Conteneurisation et pipelines de données

Gestion des Dépendances et Environnement d’Exécution

Pourquoi gérer les dépendances dans un conteneur ?

- Un conteneur ne doit contenir que les dépendances strictement nécessaires.
- Importance de l’isolation des environnements pour éviter les conflits de versions.
- Comparaison entre différentes approches : pip, Poetry, conda.
- Exécution de scripts SQL et interaction avec les bases de données depuis un conteneur.

2. Conteneurisation et pipelines de données

Isolation des Dépendances pour Python

Différentes Approches pour Gérer les Dépendances

Utilisation de pip (approche classique) :

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

Problèmes : Pas de gestion avancée des dépendances et des versions.

2. Conteneurisation et pipelines de données

Isolation des Dépendances pour Python

Différentes Approches pour Gérer les Dépendances

Utilisation de Poetry (gestion moderne des dépendances) :

```
FROM python:3.9
WORKDIR /app
COPY pyproject.toml poetry.lock ./
RUN pip install poetry && poetry install --no-dev
COPY . .
CMD ["poetry", "run", "python", "app.py"]
```

Avantages : Gestion des versions et des environnements virtualisés.

2. Conteneurisation et pipelines de données

Isolation des Dépendances pour Python

Différentes Approches pour Gérer les Dépendances

Utilisation de Conda (idéal pour la data science) :

```
FROM continuumio/miniconda3
WORKDIR /app
COPY environment.yml .
RUN conda env create -f environment.yml
SHELL ["/bin/bash", "-c"]
CMD ["conda", "run", "-n", "mon_env", "python", "app.py"]
```

Avantages : Gestion avancée des dépendances avec optimisation des bibliothèques C.

2. Conteneurisation et pipelines de données

Utilisation de Conteneurs pour Exécuter des Scripts SQL

- **Pourquoi exécuter SQL depuis un conteneur ?**
  - Uniformisation des environnements de développement et production.
  - Éviter les différences de configuration entre machines.

Exécution d'un script SQL dans un conteneur PostgreSQL :

```
docker run --rm -v $(pwd)/scripts:/scripts \
           --network some_network \
           postgres:latest psql -h database \
           -U user -d mydb -f /scripts/init.sql
```

- **Avantages :**
  - Assure que les scripts s'exécutent dans un environnement contrôlé.
  - Facilité de gestion des versions des bases de données via les conteneurs.

2. Conteneurisation et pipelines de données

Connexion aux Bases de Données depuis un Conteneur

Utilisation de variables d'environnement pour stocker les informations de connexion :

```
ENV POSTGRES_USER=admin
ENV POSTGRES_PASSWORD=secret
ENV POSTGRES_DB=mydb

docker run -e POSTGRES_USER=admin
           -e POSTGRES_PASSWORD=secret
           -e POSTGRES_DB=mydb postgres:latest
```

2. Conteneurisation et pipelines de données

Connexion aux Bases de Données depuis un Conteneur

Connexion depuis une application Python dans un conteneur :

```python
import psycopg2
import os

conn = psycopg2.connect(
    dbname=os.getenv("POSTGRES_DB"),
    user=os.getenv("POSTGRES_USER"),
    password=os.getenv("POSTGRES_PASSWORD"),
    host="database"
)
```

2. Conteneurisation et pipelines de données

Connexion aux Bases de Données depuis un Conteneur

Utilisation de Docker Compose pour simplifier la gestion des connexions :

```yaml
services:
  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: mydb
    ports:
      - "5432:5432"
```

2. Conteneurisation et pipelines de données

Introduction à Docker Compose

- **Définition :**
  - Docker Compose est un outil permettant de définir et gérer des applications multi-conteneurs.
  - Utilisation d'un fichier YAML (docker-compose.yml) pour spécifier les services.
- **Pourquoi utiliser Docker Compose ?**
  - Facilite le déploiement d'environnements complexes.
  - Simplifie la gestion des connexions entre conteneurs.
  - Permet de lancer l'ensemble des services avec une seule commande.

```yaml
version: '3'
services:
  app:
    image: my_app:latest
    depends_on:
      - db
  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: mydb
```

```
docker-compose up -d  # Démarrer en arrière-plan
docker-compose down  # Arrêter et supprimer les conteneurs
```

2. Conteneurisation et pipelines de données

Pourquoi conteneuriser les composants des pipelines de données

- Assurer la portabilité et la reproductibilité des traitements de données.
- Standardiser les environnements entre le développement, les tests et la production.
- Optimiser la gestion des dépendances et simplifier les mises à jour.
- Sécuriser et isoler les différents composants du pipeline.

2. Conteneurisation et pipelines de données

Préparer l’environnement de conteneurisation

Outils et prérequis

- Installation de **Docker** et **Docker Compose**.
- Vérification de l’installation avec :

```
docker --version
docker-compose --version
```

- Création d’une structure de projet pour gérer les fichiers Docker :

```
my_pipeline/
├── dagster_project/
├── dbt_project/
├── docker-compose.yml
```

2. Conteneurisation et pipelines de données

Préparer l'environnement de conteneurisation

Conteneurisation de Dagster

- Pourquoi utiliser **Dagster** ?
  - Gestion centralisée des pipelines de données.
  - Déclenchement et planification des tâches.
- **Dockerfile** pour conteneuriser Dagster :

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["dagster", "api", "grpc", "--module-name", "my_dagster_project"]
```

- Intégration avec Docker Compose :

```
services:
  dagster:
    image: my_dagster_image:latest
    ports:
      - "3000:3000"
    depends_on:
      - postgres
```

2. Conteneurisation et pipelines de données

Préparer l’environnement de conteneurisation

Conteneurisation des Bases de Données

- Problèmes des bases locales : **incohérences entre environnements, difficultés de mise à jour**.
- **Avantages de la conteneurisation :**
  - Facilité de déploiement et gestion simplifiée des versions.
  - Éviter les conflits de configuration.
  - Portabilité et reproductibilité des environnements.

- **Exemple avec PostgreSQL** :

```
services:
  postgres:
    image: postgres:13
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: mydb
    ports:
      - "5432:5432"
    volumes:
      - pg_data:/var/lib/postgresql/data
volumes:
  pg_data:
```

2. Conteneurisation et pipelines de données

Préparer l’environnement de conteneurisation

Persistance des Données et Bonnes Pratiques

Titre : Gérer la persistance des bases de données conteneurisées

Contenu :

- **Problème :** Un conteneur est éphémère, comment conserver les données ?
- **Solution : Utilisation de volumes Docker**

```
volumes:
  pg_data:
```

- **Sauvegarde et restauration des bases conteneurisées** :

```
docker exec -t postgres pg_dump -U admin mydb > backup.sql
docker exec -i postgres psql -U admin mydb < backup.sql
```

2. Conteneurisation et pipelines de données

Préparer l’environnement de conteneurisation

Conteneurisation des Transformations avec dbt

- **Pourquoi conteneuriser dbt ?**
  - Assurer une cohérence des dépendances entre environnements.
  - Automatiser et simplifier les exécutions via CI/CD.
- **Dockerfile dbt :**

```
FROM python:3.9
RUN pip install dbt-core dbt-postgres
COPY dbt_project /dbt_project
WORKDIR /dbt_project
CMD ["dbt", "run"]
```

- **Intégration avec Docker Compose :**

```
services:
  dbt:
    image: my_dbt_image:latest
    volumes:
      - ./dbt_project:/dbt_project
    depends_on:
      - postgres
    command: ["dbt", "run"]
```

2. Conteneurisation et pipelines de données

Préparer l’environnement de conteneurisation

Exécution du Pipeline Conteneurisé

- **Démarrage du pipeline complet :**
  docker-compose up -d
- **Vérification des logs :**
  docker logs -f <nom_conteneur>
- **Accès aux interfaces :**
  - Dagster UI : http://localhost:3000
  - PostgreSQL via psql ou un client GUI.

2. Conteneurisation et pipelines de données

3. Industrialisation et sécurisation des pipelines de données

3.3 Bases de données: comparatif

Introduction à l’Industrialisation et Sécurisation des Pipelines Conteneurisés

Pourquoi industrialiser un pipeline conteneurisé ?

- Objectifs de l’industrialisation : **fiabilité, scalabilité, automatisation**.
- Automatiser les tests, le déploiement et la gestion des mises à jour.
- Améliorer la **sécurité** et assurer un contrôle qualité rigoureux.
- Utilisation des outils de CI/CD pour déployer et gérer efficacement les pipelines.

3.3 Bases de données: comparatif

Introduction à l’Industrialisation et Sécurisation des Pipelines Conteneurisés

Introduction à la CI/CD pour les Pipelines Conteneurisés

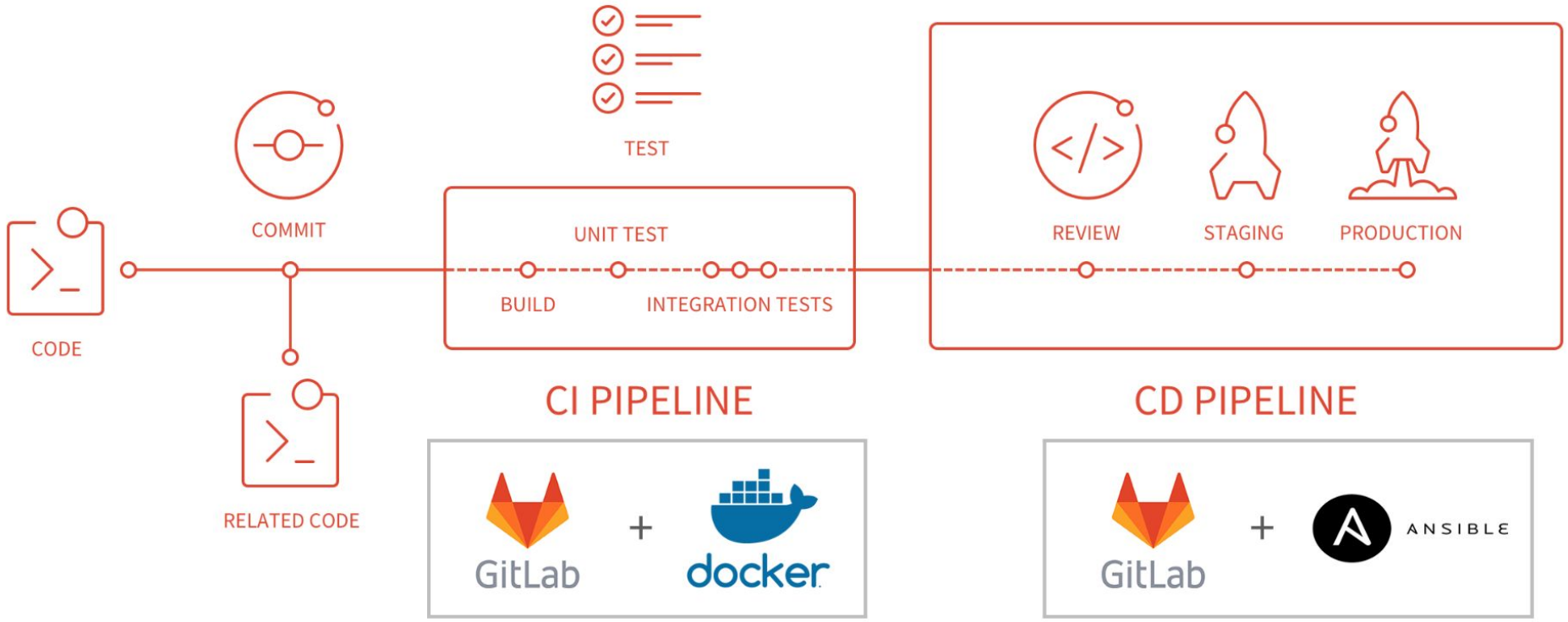
Comprendre l’intégration et le déploiement continus

- **Définition CI/CD :**
  - **CI (Continuous Integration)** : Tests automatisés et validation du code.
  - **CD (Continuous Deployment/Delivery)** : Déploiement automatique en production.
- **Pourquoi utiliser CI/CD pour les pipelines de données ?**
  - Détection rapide des erreurs.
  - Maintien d’un environnement stable et reproductible.
  - Optimisation du temps de déploiement.

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Introduction à la CI/CD pour les Pipelines Conteneurisés



3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Utilisation de GitLab CI/CD pour Automatiser un Pipeline

- **Définition des jobs et runners dans GitLab CI/CD**

- **Exemple de .gitlab-ci.yml pour builder une image Docker et exécuter un pipeline Dagster :**
  a. **Build et test de l'image automatiquement à chaque push.**
  b. **Assurance de la stabilité du pipeline avant déploiement.**

```yaml
image: docker:latest

services:
  - docker:dind

stages:
  - build
  - test
  - deploy

variables:
  DOCKER_DRIVER: overlay2

build:
  stage: build
  script:
    - docker build -t my_pipeline_image .

test:
  stage: test
  script:
    - docker run --rm my_pipeline_image pytest

deploy:
  stage: deploy
  script:
    - echo "Déploiement en production..."
```

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Introduction à la CI/CD pour les Pipelines Conteneurisés

Automatisation des Tests et Déploiement des Pipelines

- **Tests automatiques sur un pipeline Dagster avec GitLab CI/CD :**

```
test:
  stage: test
  script:
    - docker-compose up -d
    - docker-compose exec dagster dagster pipeline execute -f my_pipeline.py
```

Bonnes pratiques :

- Tests unitaires sur les scripts de transformation de données.
- Tests d'intégration pour valider l'ensemble du pipeline.
- Automatisation des tests avant tout déploiement.

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Introduction à la CI/CD pour les Pipelines Conteneurisés

Exécution des Pipelines sur des Runners Docker

Optimiser l'exécution sur des runners GitLab CI/CD

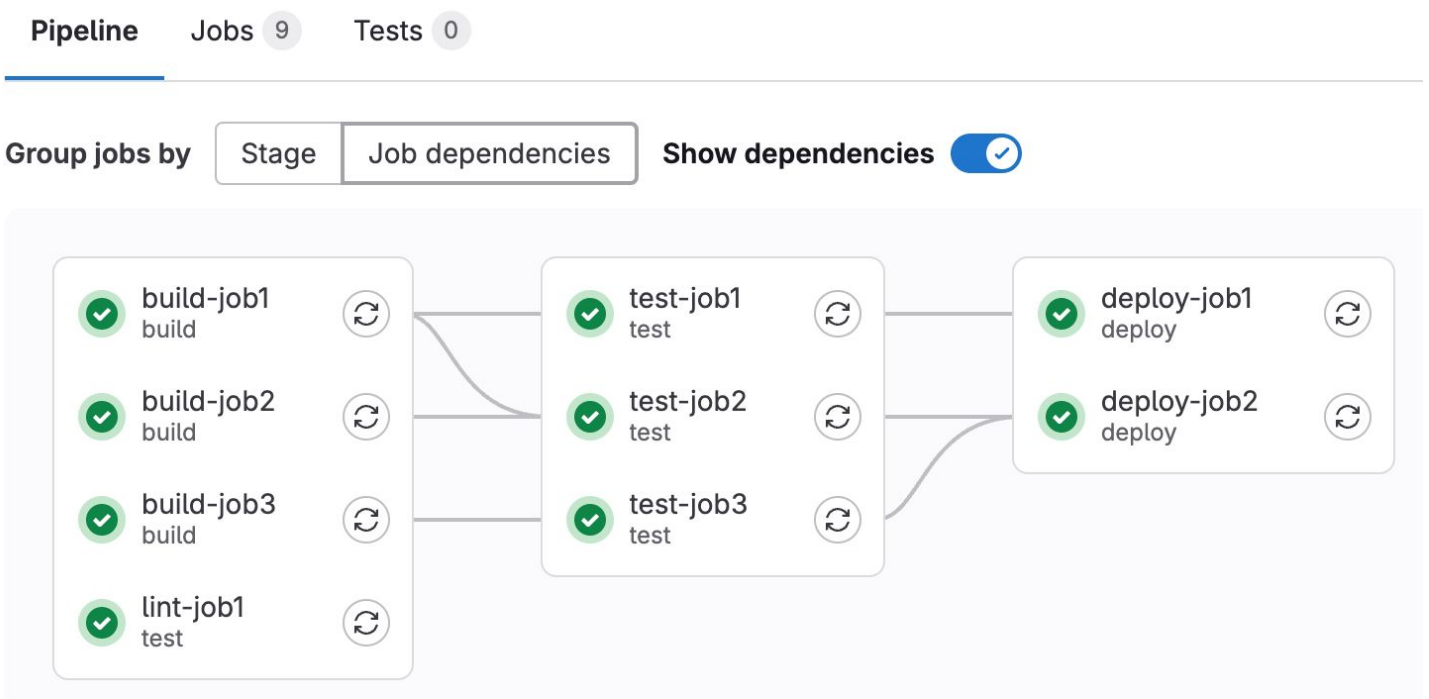
- **Pourquoi utiliser des runners Docker pour exécuter les pipelines ?**
  - Éviter les problèmes d'environnement local.
  - Assurer la reproductibilité des traitements.
  - Optimiser la gestion des ressources.
- **Configuration d'un runner Docker GitLab CI/CD :**

```
sudo gitlab-runner register \
  --non-interactive \
  --url https://gitlab.com/ \
  --registration-token <TOKEN> \
  --executor docker \
  --docker-image "docker:latest"
```

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

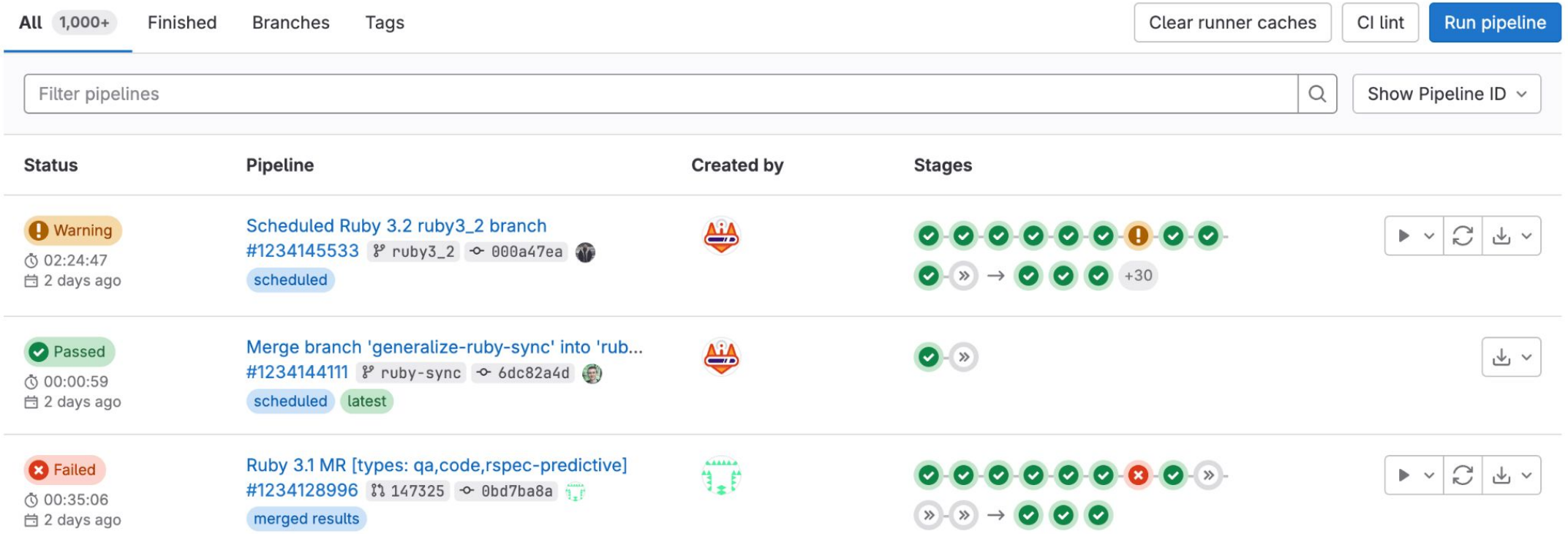
Utilisation de GitLab CI/CD pour Automatiser un Pipeline



3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Utilisation de GitLab CI/CD pour Automatiser un Pipeline



3.3 Bases de données: comparatif

Introduction à l’Industrialisation et Sécurisation des Pipelines Conteneurisés

Sécurisation des Conteneurs dans un Pipeline de Données

- **Importance de la sécurité dans les pipelines de données** : Protection des données sensibles, prévention des attaques, conformité aux normes.
- **Menaces courantes** :
  - Images compromises ou vulnérables.
  - Fuite de secrets et mauvaises gestions des accès.
  - Conteneurs non isolés pouvant compromettre l’ensemble du système.
- **Approches de sécurisation :**
  - Sécurisation des images Docker.
  - Gestion des secrets et accès.
  - Scanning des vulnérabilités et mise en place de politiques de sécurité.

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Meilleures Pratiques pour Sécuriser les Images Docker

- **Utiliser des images de base fiables et minimales** :
  - Privilégier des images officielles et maintenues (python:3.9-slim, alpine...)
  - Éviter les images inconnues ou non vérifiées.
- **Réduction de la surface d'attaque :**
  - Utiliser des **multi-stage builds** pour ne garder que les fichiers nécessaires.
  - Supprimer les outils inutiles dans le conteneur final.

```
FROM python:3.9 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

FROM python:3.9-slim
WORKDIR /app
COPY --from=builder /app .
CMD ["python", "app.py"]
```

- **Exécution avec un utilisateur non root :**

```
RUN addgroup --system app && adduser --system --group app
USER app
```

3.3 Bases de données: comparatif

Introduction à l’Industrialisation et Sécurisation des Pipelines Conteneurisés

Meilleures Pratiques pour Sécuriser les Images Docker

Gestion des Secrets et des Accès

Protéger les informations sensibles dans un pipeline conteneurisé

- **Pourquoi sécuriser les secrets ?**
  - Éviter l’exposition des mots de passe et clés API dans le code source.
  - Centraliser la gestion des accès pour garantir une sécurité renforcée.
- **Mauvaises pratiques à éviter :**
  - Stocker des secrets dans des fichiers .env ou des variables d’environnement non sécurisées.
  - Commits accidentels de secrets dans un dépôt Git.

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Optimisation des Performances des Conteneurs pour le Traitement de Données

Pourquoi optimiser les performances des conteneurs ?

- **Optimiser l'utilisation des ressources** : éviter le gaspillage de CPU et RAM.
- **Assurer l'efficacité des traitements de données lourds** : exécutions plus rapides et meilleures performances.
- **Faciliter l'intégration avec des infrastructures scalables** comme **Kubernetes**.

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Optimisation des Performances des Conteneurs pour le Traitement de Données

Optimisation des Ressources (CPU, RAM) dans les Conteneurs

- **Limiter la consommation de ressources dans Docker** :
  - Contraindre l'usage de la RAM et CPU :

```
docker run --memory=2g --cpus=2 my_pipeline_image
```

- **Ajouter des limites dans un fichier Docker Compose :**

```
services:
  pipeline:
    image: my_pipeline_image
    deploy:
      resources:
        limits:
          memory: 2g
          cpus: "2.0"
```

Bonnes pratiques :

- Utiliser des conteneurs spécialisés par tâche (prétraitement, transformation, stockage).
- Profiler et ajuster les ressources dynamiquement.

3.3 Bases de données: comparatif

Introduction à l'Industrialisation et Sécurisation des Pipelines Conteneurisés

Optimisation des Performances des Conteneurs pour le Traitement de Données

Utilisation de GPU dans des Conteneurs pour des Traitements Lourds

- **Pourquoi utiliser des GPU dans un pipeline de données ?**
  - Accélération des tâches gourmandes en calcul (ex : Machine Learning, Deep Learning, traitement d'images).
  - Réduction du temps de traitement par rapport aux CPU.
- **Exécuter un conteneur avec support GPU (NVIDIA):**

```
docker run --gpus all my_pipeline_image
```

- **Exemple de configuration dans Docker Compose :**

```
services:
  ai_model:
    image: my_pipeline_image
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
```

Utilisation de frameworks compatibles GPU :

- TensorFlow, PyTorch, RAPIDS.

3.3 Bases de données: comparatif

Introduction à l’Industrialisation et Sécurisation des Pipelines Conteneurisés

Conclusion

- **Mise en place d’une CI/CD robuste** : Automatisation du build, test et déploiement avec GitLab CI/CD.
- **Sécurisation des conteneurs** :
  - Utilisation d’images minimales et audit de sécurité régulier.
  - Gestion sécurisée des secrets avec Vault et Kubernetes Secrets.
- **Optimisation des performances** :
  - Contrôle des ressources CPU et mémoire pour chaque conteneur.
  - Exploitation des GPU pour les traitements lourds.
- **Déploiement et intégration multi-cloud** :
  - Comparaison des performances entre pipeline classique et conteneurisé.
  - Bonnes pratiques pour la gestion et l'orchestration des conteneurs sur plusieurs infrastructures cloud.

3. Industrialisation et sécurisation des pipelines de données