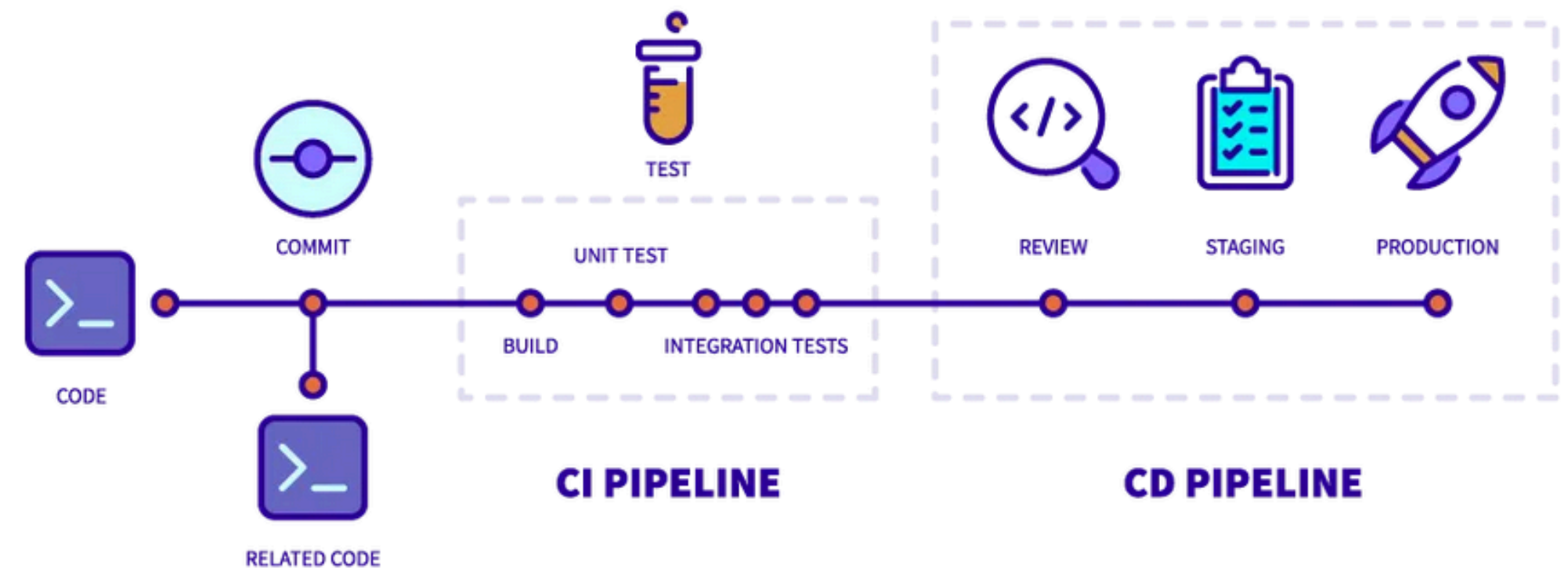


# Mises en place de tests et pipelines

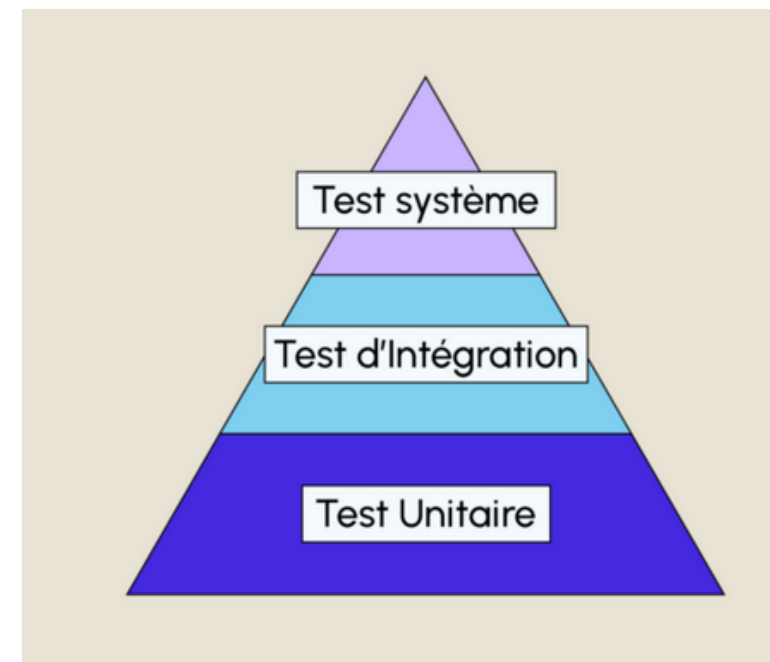
CI/CD, GitHub Actions

FORMATEUR : DIALLO ALIMOU



- **Comprendre l'importance des tests pour les pipelines de données.**
- **Connaître les types de tests spécifiques au domaine data (tests unitaires, de qualité de données, de performance).**
- **Mettre en place un pipeline de traitement de données automatisé et testé.**
- **Utiliser des outils modernes de gestion de pipelines (Airflow, Prefect) et d'intégration continue (GitLab CI/CD, GitHub Actions).**
- **Structurer un projet data pour le rendre testable et maintenable.**

## ← Le Test unitaire en analyse de données



Le test unitaire, en programmation informatique, désigne la procédure qui permet de s'assurer du bon fonctionnement d'un logiciel ou d'un code source (respectivement d'une partie d'un logiciel ou d'une partie d'un code).

- Les tests unitaires qui constituent la base (comme illustré sur le graphique ci-dessous)
- Les test d'intégration pour s'assurer du fonctionnement optimal de différents composants
- Les test systèmes pour s'assurer du fonctionnement global du système d'ensemble

En langage Python les deux principales bibliothèques utilisées pour réaliser des tests unitaires sont **Unittest** et **Pytest**. L'utilisation de la bibliothèque unittest pour la réalisation de tests nécessite une bonne connaissance de la programmation orientée objet de Python ceci à cause du fait que les tests sont réalisés par l'intermédiaire d'une classe.

À contrario, l'utilisation de la bibliothèque Pytest offre un peu plus de flexibilité dans le design et la réalisation de tests. En plus du fait que la réalisation de tests avec la bibliothèque Pytest est beaucoup plus flexible, cette dernière bibliothèque offre davantage de commandes et options de test (assert instruction).

Un exemple : Test d'une fonction calculant le carré d'un entier : Nous testerons que les valeurs données en sortie sont des entiers (test réalisés avec Pytest)

Etape 1 : Définition de la fonction `number_squared` dans le fichier `function.py`

```
docs > Test-pipeline > pytest > function.py > number_squared
1  def number_squared(x):
2      return x**2
```

Etape 2 : Définition de la fonction de test dans le fichier `test_in_int.py`

```
docs > Test-pipeline > pytest > test_in_int.py > test_square_return_value_type_is_int
1  # Import du module pytest
2  import pytest
3
4  # Import de ma fonction à tester
5  from function import number_squared
6
7  # Définition de notre campagne de test
8  # Définition de la liste de valeurs à tester
9  @pytest.mark.parametrize('inputs', [2, 4, 6])
10 def test_square_return_value_type_is_int(inputs):
11     # Quand
12     value = number_squared(inputs)
13     # Alors on vérifie que les sorties sont des entiers
14     assert isinstance(value, int)
```

L'exécution de la commande pytest dans un terminal renvoie le résultat suivant :

```
===== test session starts =====  
platform linux -- Python 3.12.3, pytest-8.4.1, pluggy-1.6.0  
rootdir: /home/santoudllo/Bureau/PROJETS/master2-supdevinci/docs/Test-pipeline/pytest  
plugins: anyio-4.9.0  
collected 3 items  
  
test_in_int.py ...
```



## ← PRÉSENTATION DE CI/CD, DE GIT, ET DES PIPELINES DE DÉPLOIEMENT

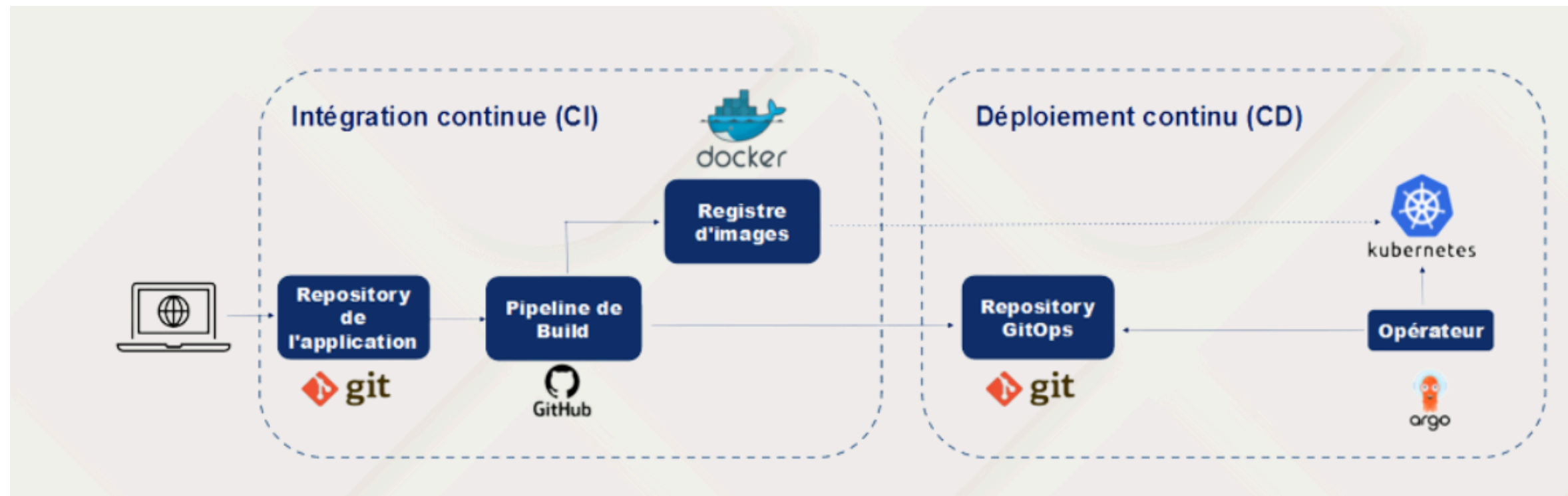
### **Intégration continue (CI)**

Les développeurs commitent fréquemment dans une branche principale managée par Git, ce qui déclenche des tests automatisés et des builds pour l'intégration. Git effectue le suivi des modifications pour activer l'extraction et le test automatiques des nouveaux commits.

### **Déploiement continu (CD)**

Est axée sur le déploiement de modifications vérifiées dans les développements de production via des phases de déploiement structurées dans les pipelines de déploiement.

- Intégration continue (CI) : chaque commit déclenche un processus "test, build and release"
  - GitHub : GitHub Actions
  - GitLab : GitLab CI/CD





Les outils CI/CD jouent un rôle important dans le bon déroulement de chaque étape du pipeline. Voici donc une liste des outils les plus utilisés :

- **Jenkins** : un serveur java open source gratuit qui gère l'intégration continue.
- **Bamboo** : un logiciel d'intégration, de déploiement et de livraison continue qui permet de rassembler et d'automatiser les builds, les tests et les versions dans un seul workflow.
- **Azure Pipeline** : outil récent développé par Microsoft pour les dépôts Azure DevOps. Il possède l'avantage d'être intégré dans l'écosystème Azure.
- **Circle CI** : une solution CI très personnalisable. Son point fort : elle prend en charge des langages de programmation très variés (Java, Python, JS, Haskell, Ruby on Rails et Scala).
- **ArgoCD** : un outil open source de déploiement continu basé sur GitOps, permettant la synchronisation automatique des applications Kubernetes avec les référentiels Git.



## ← INTÉGRATION DE GIT AVEC DES PIPELINES DATA

Git est un système de contrôle de version qui permet aux développeurs de suivre les modifications dans leur codebase (ou définitions de code JSON, dans le cas de pipelines) et de collaborer avec d'autres personnes. Il fournit un dépôt centralisé où les modifications de code sont stockées et gérées. Actuellement, Git est pris en charge dans Fabric via GitHub ou Azure DevOps. Il existe quelques concepts essentiels du workflow à comprendre lors de l'utilisation de Git.

- **Branche primaire** : la branche primaire, parfois appelée branche maîtresse, contient du code prêt pour la production.
- **Branches de fonctionnalités** : ces branches sont distinctes de la branche primaire, et permettent d'effectuer un développement isolé sans modifier la branche primaire.
- **Demandes de tirage (PR)** : les demandes de tirage (pull requests) permettent aux utilisateurs de proposer, réviser et discuter des modifications avant l'intégration.
- **Fusion** : elle se produit lorsque les modifications sont approuvées. Git intègre ces modifications, mettant à jour le projet en continu.

## ← QU'EST-CE QU'UN PIPELINE CI/CD

Le **pipeline** est l'ensemble des étapes qu'on met en place pour développer un logiciel dans une stratégie CI/CD. On veut donc automatiser les processus et les rendre plus rapides et plus fiables.

Si les étapes spécifiques varient d'une entreprise à une autre, la structure d'un pipeline reste souvent la même :

- **Développement** : chaque développeur travaille simultanément sur une fonctionnalité ou une itération différente du code source.
- **Build** : chaque nouveau code validé dans le dépôt (GIT) est rassemblé, validé et compilé.
- **Test** : la compilation passe par une série de tests automatisés pour s'assurer qu'elle fonctionne bien. Le plus souvent, ce sont des **tests unitaires**, où le code est divisé en petites unités testées individuellement.
- **Analyse** : analyse du code en profondeur pour obtenir des métriques sur la qualité de code, le code smelling, les dépendances avec des CVE et le calcul de la dette technique.
- **Livraison** : l'application est distribuée vers le référentiel qui stocke les builds, ou morceaux de codes (dépendances) que vous avez créés lors de la phase de CI (Intégration Continue), des artefacts de build créés par l'intégration continue.
- **Déploiement** : une fois que le code a été vérifié par des tests unitaires et des tests d'acceptation utilisateur (UAT) et que tout est prêt pour la distribution, le projet est envoyé vers un environnement de déploiement. Ce sont généralement des simulations et des vérifications supplémentaires et chaque modification manuellement approuvée est envoyée en production.

- Linters: diagnostic de qualité du code
  - Pylint
- Formatters: application automatique des standards
  - Black

### **Astuce**

- Exemples d'erreurs repérées par un linter:
  - lignes de code trop longues ou mal indentées, parenthèses non équilibrées, noms de fonctions mal construits...
- Exemples d'erreurs non repérées par un linter:
  - fonctions mal utilisées, arguments mal spécifiés, structure du code incohérente, code insuffisamment documenté...

- Linters: diagnostic de qualité du code
  - Pylint
- Formatters: application automatique des standards
  - Black

### **Astuce**

- Exemples d'erreurs repérées par un linter:
  - lignes de code trop longues ou mal indentées, parenthèses non équilibrées, noms de fonctions mal construits...
- Exemples d'erreurs non repérées par un linter:
  - fonctions mal utilisées, arguments mal spécifiés, structure du code incohérente, code insuffisamment documenté...

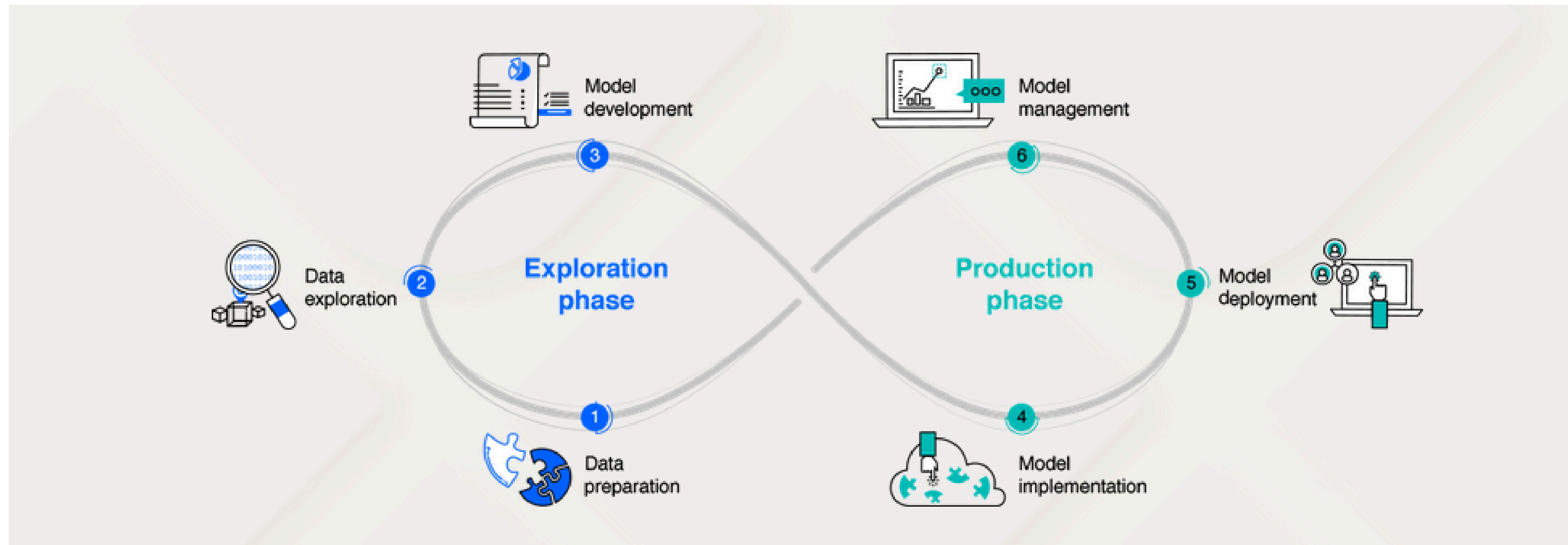
← EXEMPLE DE  
CI.YML

```
1  name: CI
2
3  on: [push, pull_request]
4
5  jobs:
6    build:
7      runs-on: ubuntu-latest
8
9      steps:
10       - name: 📄 Checkout repository
11         uses: actions/checkout@v2
12
13       - name: 🐍 Set up Python
14         uses: actions/setup-python@v2
15         with:
16           python-version: '3.10'
17
18       - name: 📦 Install dependencies
19         run: |
20           python -m pip install --upgrade pip
21           pip install "kedro[all]"
22           pip install -r belibet1/requirements.txt
23
24       - name: 🏗️ Run Kedro pipeline
25         run: |
26           cd belibet1
27           kedro run
```

GitHub exécute le job dans une machine virtuelle Ubuntu

Étape	But pédagogique
checkout	Récupérer le code du repo
setup-python	Avoir un environnement Python prêt
install dependencies	Installer tous les outils nécessaires
kedro run	Vérifier que tout le pipeline s'exécute sans erreur

## Un pipeline reproductible et automatisé



Projet devrait avoir quelque chose comme ça

```
.
├── .github/
│   └── workflows/
│       └── ci.yml    # Ton workflow CI
├── dags/             # Tes DAGs Airflow
├── src/              # Tes scripts Python
├── tests/            # Tes tests Pytest
├── .env.example      # Exemple de ton .env
├── requirements.txt
├── Makefile
└── README.md
```



**Mettre en place GitHub Actions pour exécuter automatiquement les tests unitaires Python à chaque push/pull request.**

Dans leprojet

```
mkdir -p .github/workflows
```

Créer le fichier GitHub Action

```
touch .github/workflows/python-tests.yml
```

Qu'est-ce que Kedro ?

Framework Python open source pour construire des projets data robustes, maintenables, reproductibles et testés.

Permet d'organiser code, données, pipelines ETL/ML avec architecture modulaire.

Utilise les concepts de pipelines, nodes, catalogue de données, configuration.

Favorise la collaboration en équipe et la bonne ingénierie logicielle dans la data.

Prérequis :


Python 3.7+ (de préférence 3.8+)

pip à jour

`pip install kedro`

Ou pour la dernière version :

`pip install --upgrade kedro`

Commande	Objectif principal	Quand l'utiliser	
<code>kedro new</code>	Crée un <b>nouveau projet Kedro</b> dans un <b>nouveau dossier</b>	✓ Si tu pars <b>de zéro</b> , sans dossier existant	
<code>kedro init</code>	Initialise Kedro <b>dans un dossier déjà existant</b>	✓ Si tu as <b>déjà créé un dossier</b> manuellement	

← Kedro + MongoDB  
— Pipeline  
d'ingestion de  
données  
d'exportation de  
Kedro

## Contexte général

Tu utilises Kedro, un framework Python très puissant pour construire des pipelines de données reproductibles, modulaires et bien organisés.

Ici, tu as créé une pipeline pour récupérer des données depuis une API publique, les nettoyer, puis les insérer dans une base de données MongoDB.

```
belibetl/  
└─ src/  
    └─ belibetl/  
        └─ pipelines/  
            └─ ingestion/  
                ├── __init__.py  
                ├── nodes.py  
                ├── pipeline.py  
                └─ catalog.yml (ou dans config/ tu as catalog.yml)
```

### **pipelines/ingestion/ :**

C'est ici que tu définis ta logique métier, les nodes qui correspondent à chaque étape (extraction, nettoyage, insertion).

### **nodes.py :**

Ce fichier contient les fonctions qui vont être exécutées dans ta pipeline. Par exemple `fetch_api_data()`, `clean_data()`, `insert_to_mongo()`.

### **pipeline.py :**

Ici tu crées et relies les nodes en pipeline, c'est la séquence d'exécution.

### **catalog.yml** (dans conf/base ou ailleurs selon Kedro) :

Ce fichier configure les sources et destinations des données. Par exemple, tu définis un `MemoryDataset` pour stocker les données temporairement entre nodes.

kedro new

Réponds aux questions :

Project Name: donner un nom le nom du projet comme (BelibETL)

## Outils conseillés pour un projet ETL

1,2,3,5

Option	Outil	Pourquoi le choisir ?
1	Lint (Ruff)	Pour vérifier la qualité du code automatiquement. Important en CI/CD.
2	Test (pytest)	Pour écrire des tests unitaires et les intégrer dans GitHub Actions.
3	Log	Pour une meilleure gestion des logs selon l'environnement (dev/prod).
5	Data Folder	Structure dédiée pour organiser tes datasets (raw, interim, processed, etc.).

## À éviter pour le moment

Option	Pourquoi sauter ?
4 (Docs/Sphinx)	Tu pourras l'ajouter plus tard, utile pour gros projets ou industrialisation.
6 (PySpark)	Pas utile ici, ton ETL est simple et tourne sans Spark.

## Choix d'ajouter un pipeline il faut choisir non(N)

Pourquoi ne pas inclure l'exemple ?

- Le pipeline spaceflights est juste un exemple fictif (NASA, voyages spatiaux).
- Tu vas construire ton propre pipeline , donc pas besoin de polluer ton projet avec des fichiers inutiles.