M2 SIAD Data Science
2021 - 2022

La Data Science Orientée Production

3. Le DevOps pour la Data Science

Corentin Vasseur

1

Introduction

Définition du DevOps
Tool chain
MLOps

2

Git

Introduction
Services et fonctionnalités
Les 10 commandements

3

CI/CD

L'intégration continue
Les tests unitaires
La documentation
Déploiement continue

1

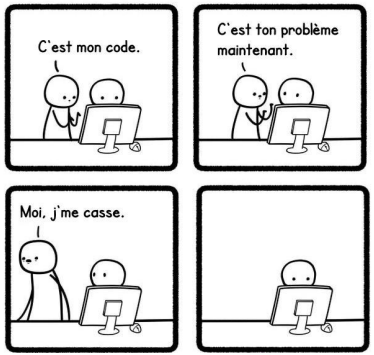
Introduction

RETOUR SOMMAIRE

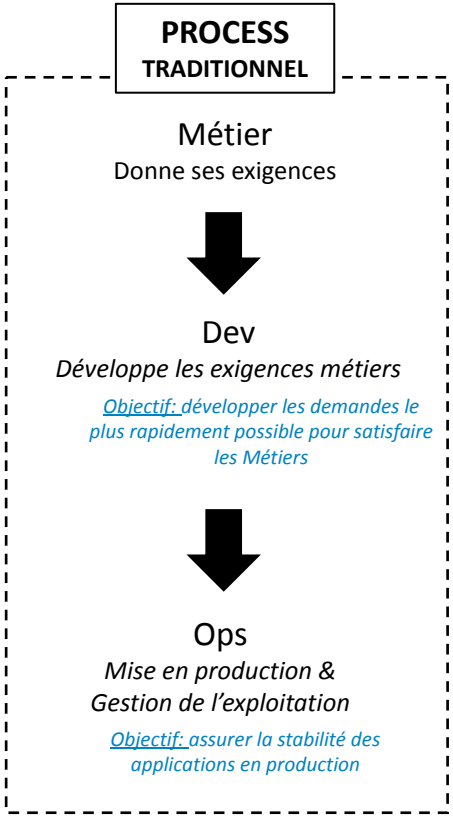
1. Introduction

- L'idée est de vous faire **découvrir** quelques **bonnes pratiques** de développement pour un projet de data science et d'y comprendre les **enjeux**.

- **Objectifs :**
  - ❑ Développer et/ou renforcer les bonnes pratiques de code dans le cadre d'un processus Quality Insurrance.
  - ❑ Définir des bases communes en terme de bonnes pratiques.
  - ❑ Découvrir et comprendre les méthodes d'industrialisation des modèles à travers le concept DevOps.



1. Introduction



Limites :

- Time to Market long : les dev se plaignent d'une mise en prod trop longue
- Les Ops se plaignent d'avoir trop d'incidents causés par un code de mauvaise qualité

Solution : DevOps

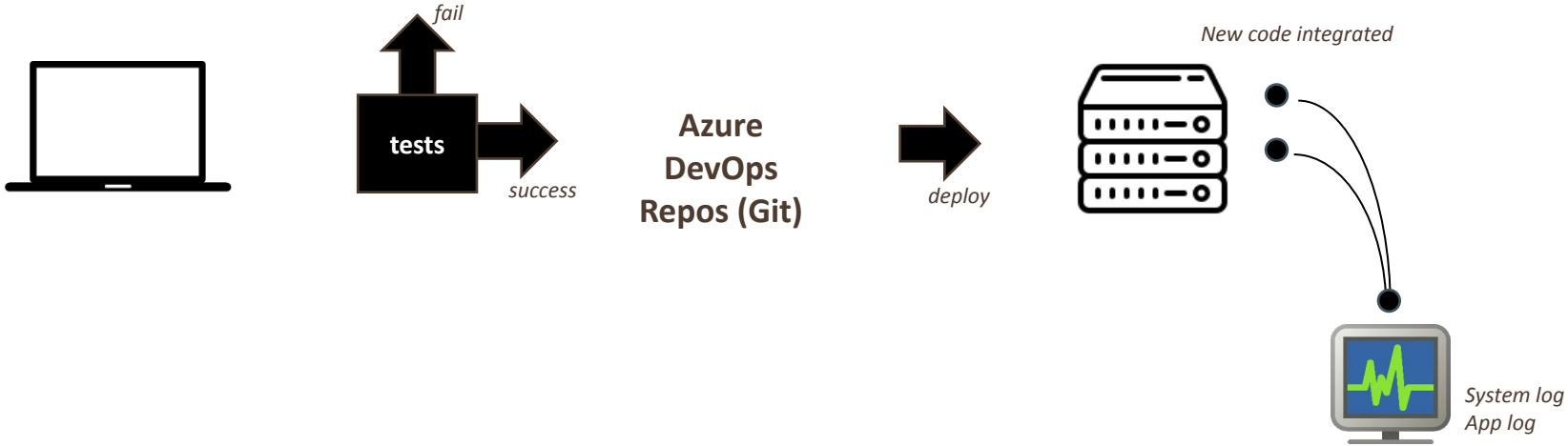
L'idée est de fluidifier la communication entre Dev et Ops en cassant les silos

Comment ?

- Intégration continue du logiciel par l'équipe de développement initiale
- Lancement automatisé de tests de qualité
- Pousser du code dans un environnement de production
- Déploiement d'une version d'un logiciel en un click

Le DevOps, c'est intégrer les développeurs & opérateurs dans le but d'améliorer la productivité en automatisant l'infra, les workflow et en mesurant les performances des applications.

1. Introduction



1. Introduction

Dev **Ops**

Development **Operations**

responsable de la création de nouvelles fonctionnalités

responsable de rendre accessible le logiciel aux utilisateurs

Nous **Digital**

1. Introduction

DevOps : le pourquoi, le comment

Pourquoi :

- Réduction du risque associé au déploiement
- Changer la version d'un produit plus rapidement
- Mesurer l'effet d'une nouvelle feature plus rapidement (test and learn)

Comment :

- Intégration continue du logiciel par l'équipe de développement initiale
- Lancement automatisé de tests de qualité
- Pousser du code dans un environnement de production
- Déploiement d'une version d'un logiciel en un click

1. Introduction

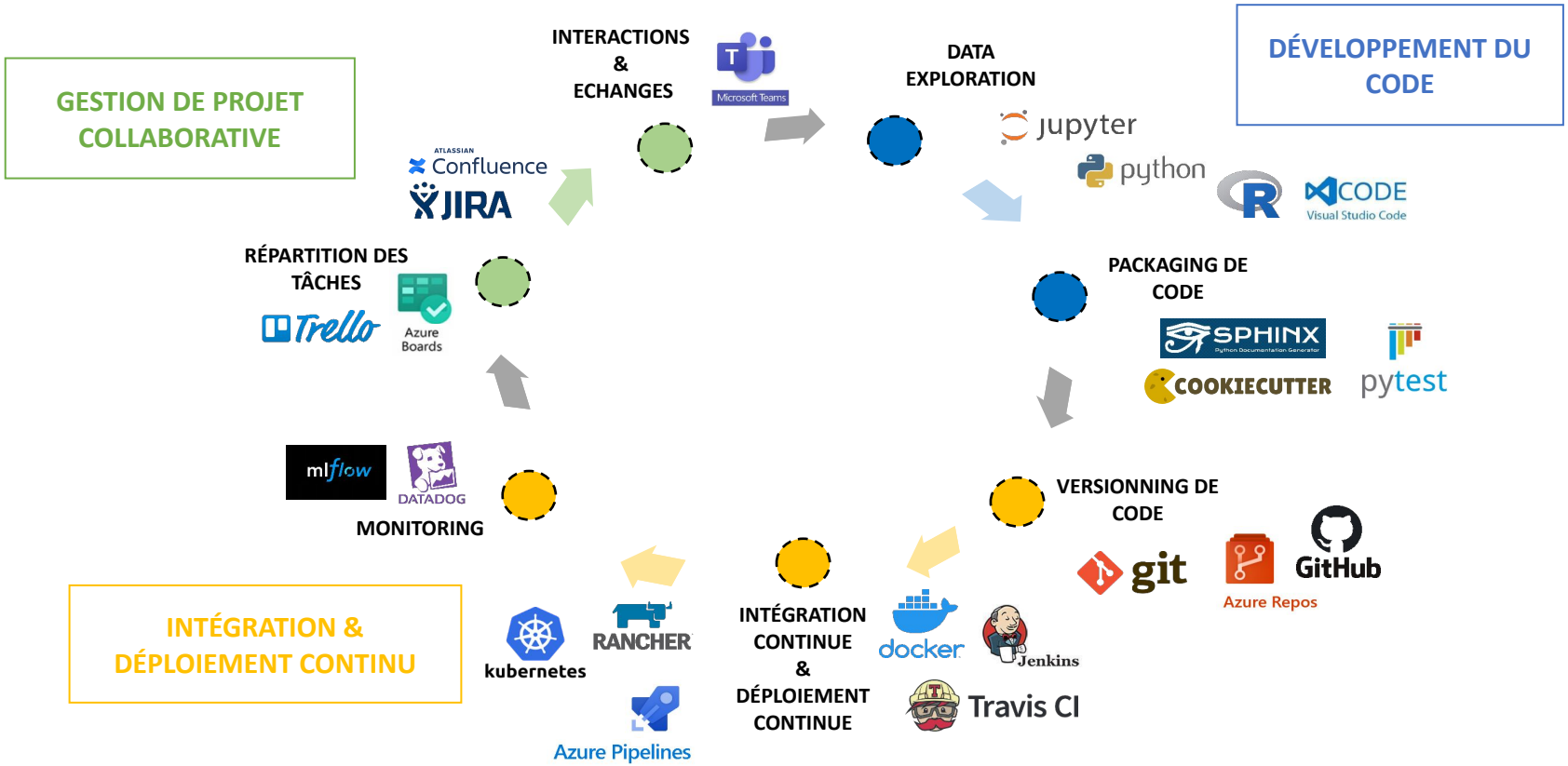
DevOps : en quoi est-ce utile ?

Sans DevOps

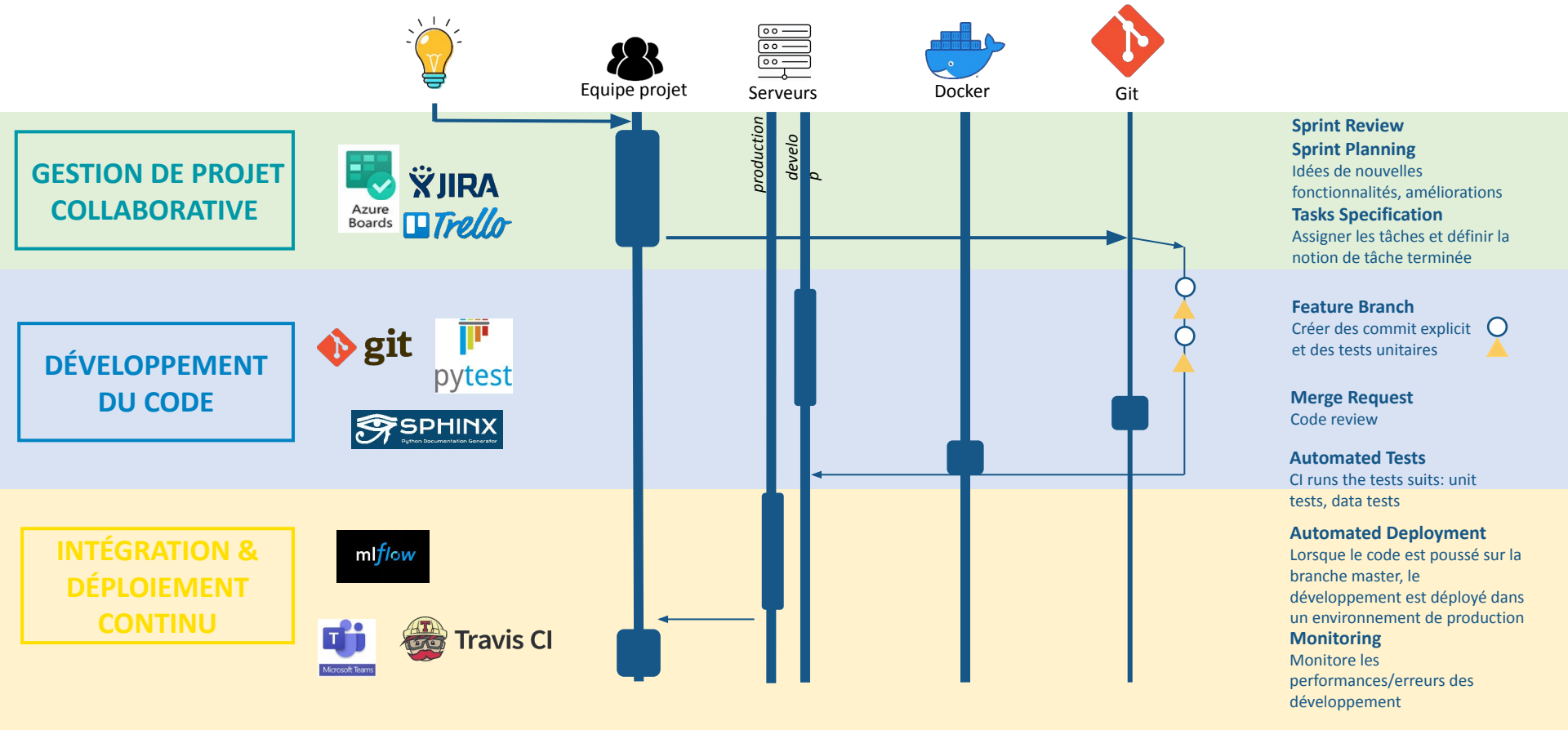
Avec DevOps

C’est se forcer à réfléchir au processus d’industrialisation le plus tôt possible !

1. Introduction



1. Introduction



1. Introduction

Dev

- Maîtrise de git
- Maîtrise des environnements virtuels
- Maîtrise du packaging d'une application (environnements virtuels, dépendances, scripts de build)
- Bonne conception du code pour permettre à celui-ci d'être testé
- Maîtrise de la conception des tests unitaires et d'intégration
- Maîtrise des quality gate sur le code (pep8, couverture de test, tests de sécurité, tests de performances) et de leur automatisation

Ops

- Savoir interagir avec un serveur
- Installer des projets et des dépendances projets sur un serveur
- Gérer les logs et les alertes provenant d'un programme
- Comprendre la notion de service (au sens informatique service unix)

1. Introduction

MLOps = adaptation du DevOps pour le ML

C ’est plus complexe car il faut être en mesure de gérer les changements sur 3 niveaux :

DATA
Data Engineering Pipelines

Data ingestion
Exploration et validation
Data wrangling
Data splitting

MODEL
Model Pipelines

ML training
Model evaluation
Model testing
Model packaging

CODE
Code Deploy Pipelines

Intégration ML dans le produit final (ex. configuration, bug fixes)
Model serving
Model Perf monitoring
Model Perf logging

2

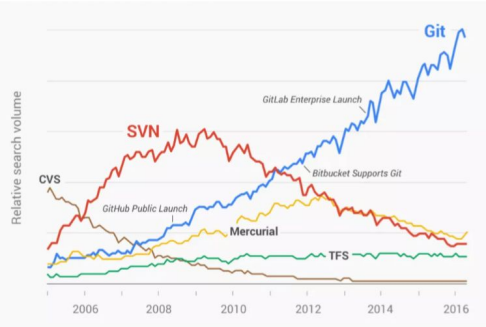
Git

RETOUR SOMMAIRE

Introduction

C’est quoi ?

- Système de contrôle de version créé en 2005
- Décentralisé et Distribué (= chaque développeur possède sur son poste sa ou ses propres versions du projet)
- Basé sur un modèle multi-branches
- Chaque branche peut être totalement indépendante des autres



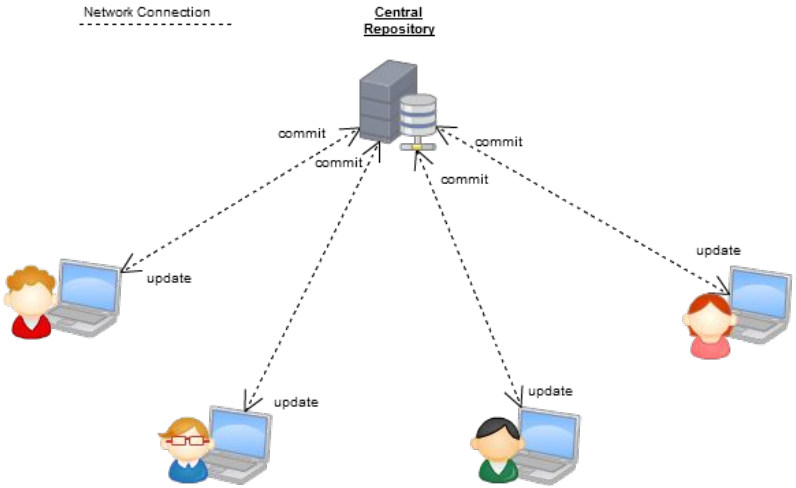
A quoi ça sert ?

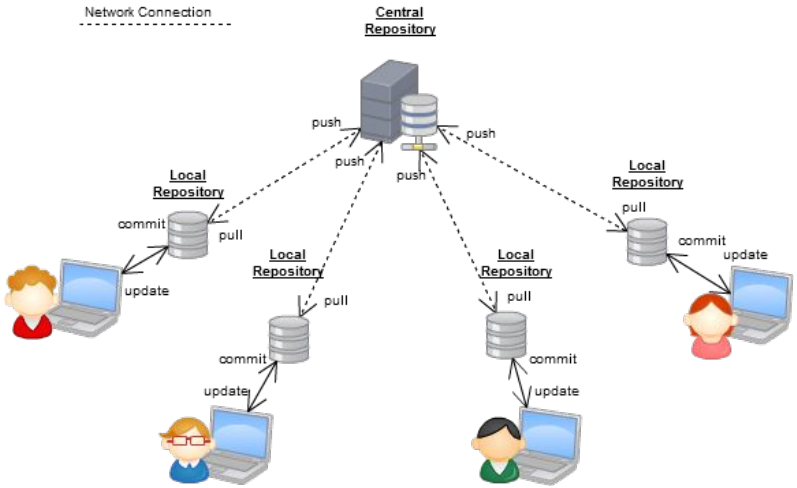
- Sauvegarde votre travail
- Permet de faire cohabiter différentes versions d’un projet
- Fusionner différentes versions pouvant provenir de plusieurs collaborateurs
- Conserve l’historique des modifications
- Permet la review de code (Merge/Pull Request)

Idée Générale : Git s’articule autour de plusieurs éléments :

- Le dépôt est le dossier (nommé .git) qui va contenir toutes les versions du projet
- Les commit représentent chaque incrément du projet. Ils sont identifiés par Git par une empreinte (hash).

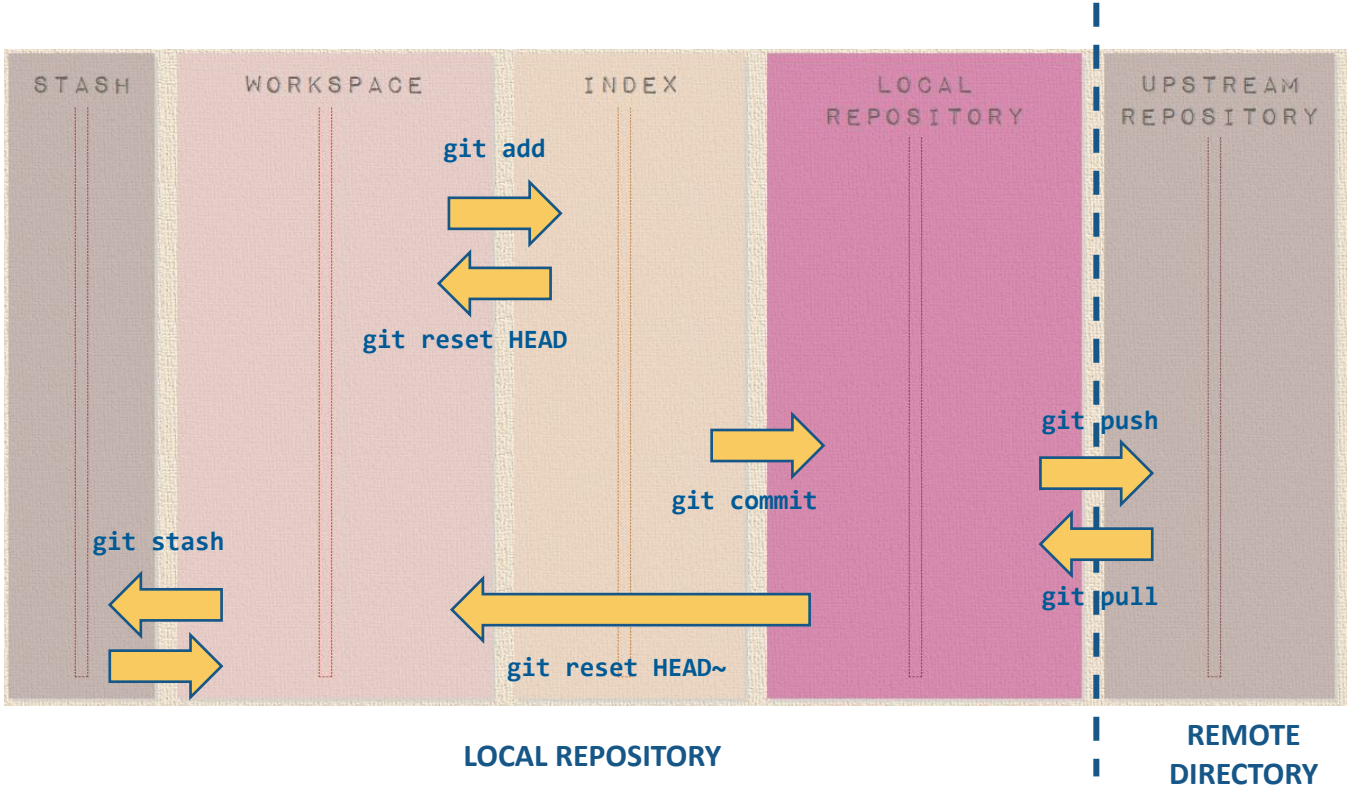
Centralisé vs décentralisé



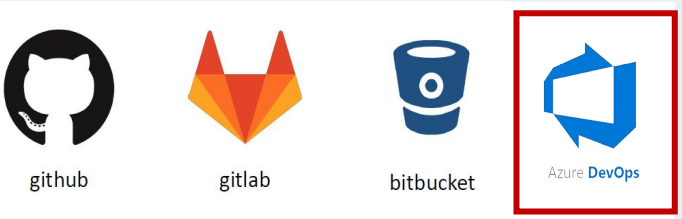


Principe de base

Comment ça fonctionne ?



Services et fonctionnalités



Critères de sélection

- Communauté
- Fonctionnalités
- Prix
- Support

Les commandes les plus utiles pour débuter

Débutant

- **Git status**
- **Git add**
- **Git commit**
- **Git push**
- **Git pull**

Intermédiaire

- Git diff
- Git branch
- Git checkout
- Git reset
- Git merge
- Git clone
- Git log

Avancé

- Git fetch
- Git rebase
- Git tag
- Git show
- Git cherry-pick
- Git describe
- Git stash
- Git config

Services et fonctionnalités

- *git config*
- *git status*
- *git add*
- *git commit*
- *git branch*
- *git checkout*
- *git clone*
- *git push*
- *git pull*

Services et fonctionnalités

- **git config**
- *git status*
- *git add*
- *git commit*
- *git branch*
- *git checkout*
- *git clone*
- *git push*
- *git pull*

- ***QUAND ?***

  Après installation de git (première utilisation) ou au cours de la vie du projet selon les besoins

- ***POURQUOI ?***

  Il est nécessaire après installation de définir les prénom, nom et adresse mail de l'utilisateur (ex. permet en partie de reconnaitre qui a poussé le commit)

- ***COMMENT ?***

  $ git config –global user.name "Corentin Vasseur"

  $ git config –global user.email corentin.vasseur@roquette.com

Services et fonctionnalités

- *git config*
- **git status**
- *git add*
- *git commit*
- *git branch*
- *git checkout*
- *git clone*
- *git push*
- *git pull*

- ***QUAND ?***

  Avant de préparer un commit

- ***POURQUOI ?***

  Cela permet de lister les fichiers modifiés. Cela peut être pratique si certains fichiers non désirés se sont glissés dans le dépôt pendant le développement (ex. credentials, data)

- ***COMMENT ?***

  $ git status

Services et fonctionnalités

- git config
- *git status*
- **git add**
- *git commit*
- *git branch*
- *git checkout*
- *git clone*
- *git push*
- *git pull*

- ***QUAND ?***

  Juste avant de commiter

- ***POURQUOI ?***

  Parce que Git possède la notion d'index. L'index est une zone d'attente de modifications avant un commit. C'est-à-dire que les modifications de fichiers doivent être placées dans l'index pour pouvoir commiter.

- ***COMMENT ?***

  $ git add <chemin>/<fichier.ext>

Services et fonctionnalités

- git config
- *git status*
- *git add*
- **git commit**
- *git branch*
- *git checkout*
- *git clone*
- *git push*
- *git pull*

- ***QUAND ?***

  Un commit doit représenter un semble cohérent de modifications. Cela peut être le correctif d'un bug, l'ajout d'une petite fonctionnalité, etc. Il est déconseillé d'effectuer des commits de plus de 200 lignes de modifications

- ***POURQUOI ?***

  Le commit sert à enregistrer des modifications au projet. Le commit permet d'intégrer le travail d'un développeur dans une branche du projet.

- ***COMMENT ?***

  $ git commit –m " explication en un commit le plus clair possible "

Services et fonctionnalités

- git config
- *git status*
- git add
- *git commit*
- **git branch**
- *git checkout*
- *git clone*
- *git push*
- *git pull*

- ***QUAND ?***

  Au moment de commencer un nouveau développement pour un correctif ou une nouvelle fonctionnalité.

- ***POURQUOI ?***

  Pour séparer le nouveau développement des branches communes à l'équipe de développement. Les branches sont un moyen efficace de permettre à plusieurs version du projet de cohabiter et d'évoluer séparément. Les branches de correctif ou fonctionnalité peuvent ensuite être réintroduites dans les branches communes à l'aide d'une fusion ou d'un rebase.

- ***COMMENT ?***

  $ git branch <nom-de-la-branche>

Services et fonctionnalités

- git config
- *git status*
- git add
- *git commit*
- git branch
- **git checkout**
- *git clone*
- *git push*
- *git pull*

- ***QUAND ?***
  Au moment de change de branche.

- ***POURQUOI ?***
  Pour continuer le développement de cette branche

- ***COMMENT ?***
  $ git checkout nom-de-la-branche

Services et fonctionnalités

- git config
- *git status*
- git add
- *git commit*
- *git branch*
- *git checkout*
- ***git clone***
- *git push*
- *git pull*

- ***QUAND ?***

  Juste avant de commiter

- ***POURQUOI ?***

  Parce que Git possède la notion d'index. L'index est une zone d'attente de modifications avant un commit. C'est-à-dire que les modifications de fichiers doivent être placées dans l'index pour pouvoir commiter.

- ***COMMENT ?***

  $ git add <chemin>/<fichier.ext>

Services et fonctionnalités

- git config
- *git status*
- git add
- *git commit*
- *git branch*
- *git checkout*
- *git clone*
- **git push**
- *git pull*

- ***QUAND ?***

  Après un commit dans une branche partagée entre les développeurs du même dépôt distant

- ***POURQUOI ?***

  Pour que le développeur puisse partager son travail.

- ***COMMENT ?***

  $ git push nom-du-remote nom-de-la-branche

Services et fonctionnalités

- git config
- *git status*
- *git add*
- *git commit*
- *git branch*
- *git checkout*
- *git clone*
- *git push*
- **git pull**

- ***QUAND ?***

  Avant de créer une branche pour un nouveau développement ou au moment de se mettre à jour (pour une revue de code ou un déploiement notamment)

- ***POURQUOI ?***

  Pour recevoir les nouveaux commits du dépôts central

- ***COMMENT ?***

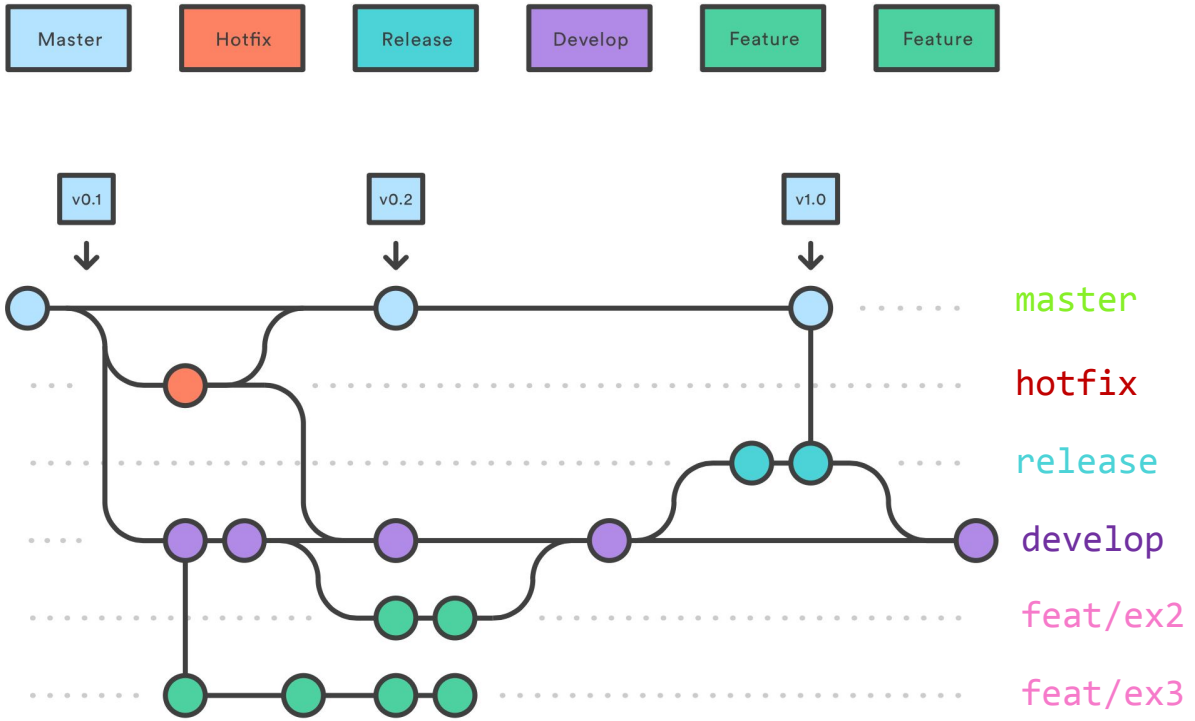
  $ git pull

#Ex1 A vous de jouer

1. **Créer et cloner** le repo distant git-onboarding (ex. sur GitHub)
2. Créez une **nouvelle branche** pour **ajouter une nouvelle fonctionnalité** (ex. feat/demo)
3. **Créez** un **fichier** (feeling.txt).
4. **Ajoutez** vos impressions concernant la formation.
5. **Indexez** ce fichier.
6. **Vérifiez** que ce fichier est bien dans l'index.
7. **Annulez** vos modifications et **supprimez** le fichier feeling.txt
8. **Modifiez** le fichier **README.md** en ajoutant vos impressions sur git
9. **Indexez/commitez** et **poussez** votre branche sur l'origin.
10. **Regardez** votre **historique** et constatez le résultat.
11. **Ouvrez** une **Pull Request** pour **fusionner** cette nouvelle branche sur **develop** et la publier

Workflow des branches

Les bonnes pratiques : fonctionnement des branches



Conventions branches et commit

Les bonnes pratiques : conventions

Branches

Plusieurs conventions existent :

- **Git Flow**
- **Github Flow**
- **Gitlab Flow**

Permet de créer le cycle de vie des sources du produit et conserver un état stable et détermine la production

Les principaux axes (branches) :

- **`master`** : code de production
- **`develop`** : code de pré-production
- Autres:
  - **`feat/*`**
  - **`fix/*`**
  - **`hotfix/*`**
  - **`release/*`**

nouveaux développements qui seront intégrés tôt ou tard à la branche develop

Commit

Plusieurs conventions existent :

- **Git Conventional Commits**
- **Angular**
- **VueJs, etc.**

Permet de créer des règles pour un historique de commit explicite qui :

- **Facilite l'écriture d'outils automatisés**
- **Permet de communiquer la nature des changements aux coéquipiers, au public et aux autres parties prenantes.**

```
<type>(<optional scope>): <subject>
```

- **`type`** :
  - fix : corrige un bogue dans le code
  - feat : introduit une nouvelle fonctionnalité dans le code
  - chore / docs / style / perf / test / …: à la discrétion des équipes
- **`scope`** : section du code impactée par la modification
- **`subject`** : description courte de moins de 90 caractères sur les modifications opérées sur le code

Release

Comment nommer sa release ?

Les 10 commandements

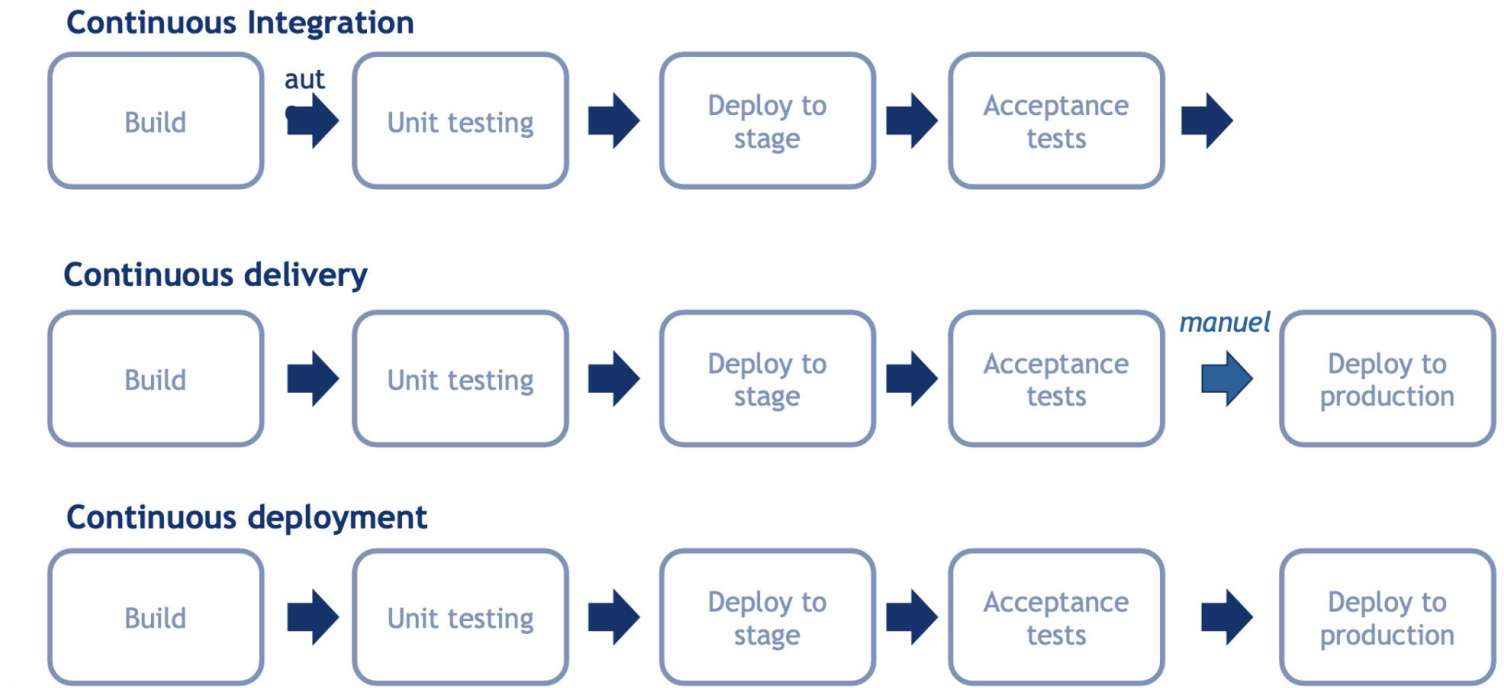
1. **Tu ne pousseras jamais tes métadonnées d'environnement (utiliser le .gitignore)**
2. **Tu utiliseras des conventions de nommages pour tes branches et tes commits**
3. **Tu tireras une branche avant tout nouveau développement**
4. **Tu feras un pull quotidien avant de commencer à travailler**
5. Tu ne pousseras (push) que des travaux qui «compilent »
6. Tu n'utiliseras jamais le push –-force mais le push --force-with-lease
7. Tu ne pousseras jamais sur master
8. Tu éviteras les commits de merge (merge –ff-only)
9. Tu ré-écrieras (rebase / merge --squash) ton historique de commits avant publication sur un dépôt distant
10. Tu ne ré-écrieras jamais l'historique d'une branche déjà sur un dépôt distant

3

Intégration continue et déploiement continu

RETOUR SOMMAIRE

Les tests


Continuous Integration
Build
aut
Unit testing
Deploy to stage
Acceptance tests
Continuous delivery
Build
Unit testing
Deploy to stage
Acceptance tests
manuel
Deploy to production
Continuous deployment
Build
Unit testing
Deploy to stage
Acceptance tests
Deploy to production

Les tests

Pourquoi faire des tests ?

- S'assurer que notre code produit bien le comportement attendu
- S'assurer que notre code ne casse pas une fonctionnalité du code déjà existante
- Spécifier en amont le cahier des charges sur une fonctionnalité

Éviter toute régression du produit

Remarque : Tout l'intérêt de ces tests est qu'ils soient **automatisés** afin de faciliter **l'intégration continue. En effet lorsque l'on veut automatiser le déploiement, les tests sont le seul moyen de vérifier la qualité automatiquement.**

Les tests

Exemple:

- On utilise le keyword « **assert** » en python
- Les noms de fonctions de test doivent commencer par test_ pour être considérés comme des tests par Pytest

```python
# content of test_sample.py
def inc(x):
    return x + 1


def test_answer():
    assert inc(3) == 5
```

Lancer les tests:

- La commande pytest permet d'exécuter les tests
- Pytest découvre automatiquement les tests qui ont été écrit dans un
- dossier test et dont le nom de fonction commence par « test_ »

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F                                                      [100%]

================================= FAILURES =================================
_____________________________ test_answer _____________________________

    def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

test_sample.py:6: AssertionError
========================== 1 failed in 0.12 seconds ===========================
```

Les tests


```python
# content of test_expectation.py
import pytest


@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

- La « paramétrisation » permet de lancer plusieurs tests avec différents paramètres d'entrées
- Très utile pour tester plusieurs comportements de manière moins verbeuse

Les tests

1. **Il est également possible de mesurer la couverture de test d'un code :**
   - Pourcentage de lignes de code exécutées
   - Pourcentage de branches (if/else) empruntées
   - Utilisation du package **coverage**

2. **Test d'intégration**
   - Test qui vérifie le comportement de plusieurs fonctions produites par **PLUSIEURS** développeurs.
   - Il vérifie aussi l'intégration avec **"tous les tuyaux"**. Donc cela teste la communication avec la base de données, avec le front, etc…
   - Ce test peut s'effectuer AVANT ou APRÈS le déploiement sur staging.

3. **Test de performance**
   - Il s'agit de regarder si nos routes exposées répondent en un temps acceptable.
   - Ce test s'effectue **APRÈS le déploiement** sur staging et AVANT le déploiement sur production. Alors que les tests unitaires s'effectuent **AVANT le déploiement** sur staging.

La documentation

Génération automatique de documentation: les docstring

- Lister tous les fichiers du code source
- Extraire les docstring des modules et des fonctions
- Créer un ensemble de pages HTML

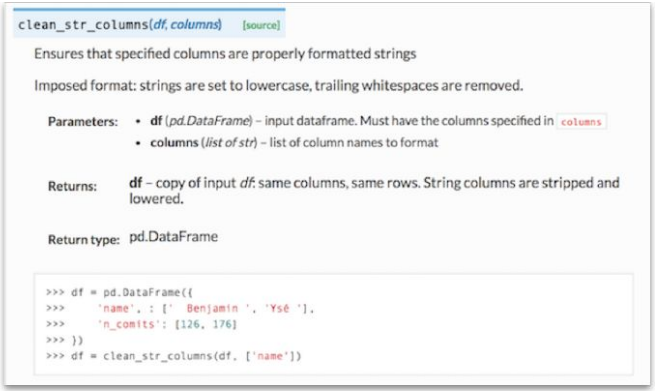
```python
def clean_str_columns(df, columns):
    """Ensures that specified columns are properly formatted strings

    Imposed format: strings are set to lowercase, trailing whitespaces
    are removed.

    Parameters
    ----------
    df: pd.DataFrame
        input dataframe. Must have the columns specified in ``columns``
    columns: list of str
        list of column names to format

    Returns
    -------
    df: pd.DataFrame
        copy of input `df`: same columns, same rows. String columns
        are stripped and lowered.


    >>> df = pd.DataFrame({
    >>>     'name', : ['  Benjamin ', 'Ysé '],
    >>>     'n_comits': [126, 176]
    >>> })
    >>> df = clean_str_columns(df, ['name'])
    """
    df = df.copy()
    for col in columns:
        df[col] = df[col].str.lower().str.strip()
    return df
```



clean_str_columns(*df*, *columns*) [source]

Ensures that specified columns are properly formatted strings

Imposed format: strings are set to lowercase, trailing whitespaces are removed.

Parameters:
- **df** (*pd.DataFrame*) – input dataframe. Must have the columns specified in `columns`
- **columns** (*list of str*) – list of column names to format

Returns: **df** – copy of input *df*: same columns, same rows. String columns are stripped and lowered.

Return type: pd.DataFrame

```
>>> df = pd.DataFrame({
>>>     'name', : ['  Benjamin ', 'Ysé '],
>>>     'n_comits': [126, 176]
>>> })
>>> df = clean_str_columns(df, ['name'])
```

Convention : docstring Numpy
(package numpydoc)

Commenter en priorité les fonctions principales qui sont utilisées ailleurs / qui servent d'API pour le projet

Le déploiement continu

Quand on parle de DevOps, on pense d'abord aux outils mais c'est pas que…

- Le principe de DevOps est de **réduire le temps** de publication de chaque version du produit en **automatisant** les tâches de déploiement.
  - Règle n°1 : Il faut TOUT **scripter** (Infrastructure as Code)
  - Règle n°2 : Il faut TOUT **automatiser**.

1. Makefile

ex : Lancer rapidement des tâches

```
install: ## install python libraries required for using the project
    pip install -r requirements.txt
```

2. Pipeline de Données

Airflow, Luigi

ex: Lancer l'insertion dans une base de données

Gère les dépendances entre tâches

3. Outil CI/CD

Gitlab CI, Jenkins

Lancer des commandes sur des serveurs

4. Containers

Docker, Kubernetes

créer des systèmes rapidement et légers

Le déploiement continu

Définitions :

- Continuous delivery

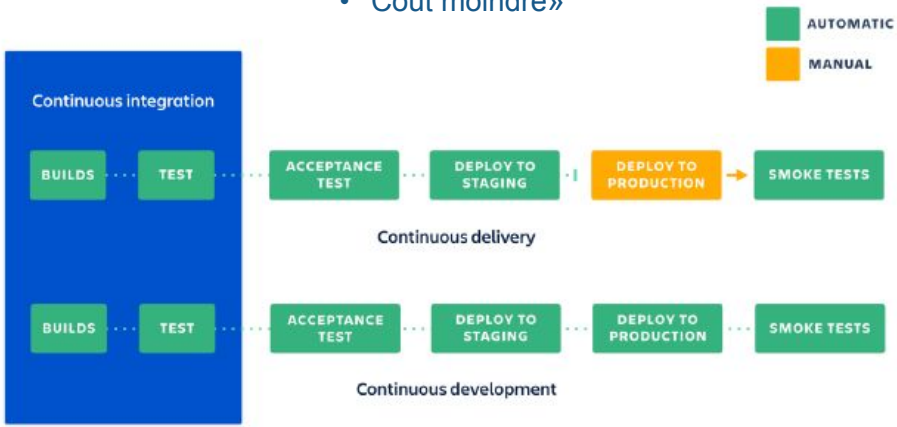
« Ensemble de pratiques de développement visant à s'assurer que les nouvelles versions du code sont **déployables à tout moment** »

- Continuous deployement :

« Ensemble de pratiques de développement visant à s'assurer que les nouvelles versions du code sont **déployées automatiquement** »

Pourquoi ?

- Innovation
- Fast feedback
- Garantie/qualité
- Moins d'effort
- Cout moindre»



Le déploiement continu

Bonne gestion des loggings

configuration (settings)

```
# Logging
LOGGING_FORMAT = '[%(asctime)s][%(levelname)s][%(module)s] %(message)s'
LOGGING_DATE_FORMAT = '%Y-%m-%d %H:%M:%S'
LOGGING_LEVEL = logging.DEBUG
logging.basicConfig(
    format=LOGGING_FORMAT,
    datefmt=LOGGING_DATE_FORMAT,
    level=LOGGING_LEVEL
)
```

usage

```
import logging
logging.debug('')
logging.info('')
logging.warning('')
logging.error('')
logging.critical('')
```

résultat

```
[2018-04-09 14:42:07][INFO][main] Running main application script
[2018-04-09 14:42:07][DEBUG][main] Doing cleaning
[2018-04-09 14:42:07][DEBUG][files] Saving object
[2018-04-09 14:42:07][DEBUG][main] Finished cleaning
[2018-04-09 14:42:07][DEBUG][files] Loading
[2018-04-09 14:42:07][INFO][main] Finished prediction.
[2018-04-09 14:42:07][DEBUG][main] Running prediction
[2018-04-09 14:42:07][DEBUG][main] Date: 2018-04-09
[2018-04-09 14:42:07][DEBUG][main] Target: SEPAL_WIDTH
[2018-04-09 14:42:07][DEBUG][main] Features: ['SEPAL_LENGTH', 'PETAL_LENGTH', 'PETAL_WIDTH']
[2018-04-09 14:42:07][INFO][main] Main application script completed.
```

*>> Pour aller plus loin : **Sentry** (un outil de reporting des log)*

UNE QUESTION ?

corentin.vasseur@decathlon.com

Machine Learning Engineer - MLOps