Programmation
TP : Introduction aux pipelines Git

Augustin Laouar

1 Introduction au CI/CD

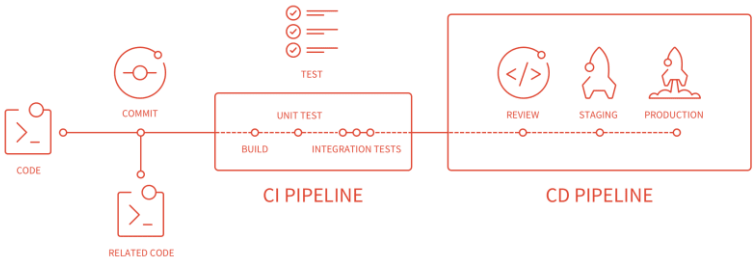

FIGURE 1 – Chaîne d'étape CI/CD.

L'intégration continue (CI) et le déploiement continu (CD) permet d'automatiser le contrôle de qualité du code. L'objectif est de vérifier automatiquement que chaque modification envoyée dans un dépôt Git reste fonctionnelle, cohérente et conforme aux règles du projet. L'idée centrale est simple : à chaque fois qu'un développeur modifie le code et l'envoie sur un dépôt Git, une série de contrôles s'exécute automatiquement afin de s'assurer que ces changements ne dégradent pas le fonctionnement du projet. Le CI/CD devient ainsi une extension naturelle du travail collaboratif, en garantissant que la base de code reste stable et propre, même lorsque plusieurs personnes travaillent en parallèle.
Dans un pipeline CI/CD typique, chaque modification déclenche automatiquement une séquence d'étapes. Ces étapes peuvent analyser la syntaxe du code, vérifier qu'il respecte les règles du projet, exécuter une compilation, lancer des tests automatisés ou encore produire des fichiers utiles au projet, appelés *artefacts*. L'ensemble de ces actions s'effectue dans un environnement isolé, configuré à partir d'images ou de machines standardisées, ce qui garantit que les résultats seront identiques pour tous les développeurs, indépendamment de leurs propres machines.

1.1 Fichier de configuration .gitlab-ci.yml

Le fichier .gitlab-ci.yml est le point central de toute automatisation dans GitLab. Il réunit au même endroit la configuration de l'intégration continue (CI) et le déploiement continu (CD). C'est donc dans ce même document que l'on définit l'ordre des phases, les environnements utilisés pour exécuter les commandes et les règles qui déterminent quand un déploiement doit avoir lieu. Le `.gitlab-ci.yml` est ainsi la seule source de vérité de la pipeline.
Sur GitHub, l'équivalent du fichier `.gitlab-ci.yml` sont les `Github Actions`, configuré dans le dossier `.github/workflows/`.

1.2 Runners

Dans GitLab, les runners sont les machines chargées d'exécuter concrètement les jobs définis dans le fichier `.gitlab-ci.yml`. Il en existe différents types, certains installés et gérés par les utilisateurs, d'autres fournis directement par GitLab. Comme l'objectif de ce TP est d'apprendre la structure et le fonctionnement d'une pipeline plutôt que son infrastructure d'exécution, nous n'entrerons pas dans le détail de la configuration des runners. Nous utiliserons exclusivement les runners partagés mis à disposition par le GitLab de l'ENS.

2 Prise en main

Pour cette première manipulation, créez un nouveau projet Git vide et ajoutez un fichier `.gitlab-ci.yml` contenant une unique étape `test` (et un unique job `job_hello`). Copiez la configuration suivante puis poussez sur GitLab :

```
stages:
  - test

job_hello:
  stage: test
  script:
    - echo "It␣works!"
```

Suivez l'exécution de la pipeline sur l'interface web et vérifiez que tout s'est bien passé.

3 Validation du code avec un Linter

Nous allons enrichir la pipeline en ajoutant une phase de validation du code à l'aide d'un *linter*. Un linter est un outil d'analyse statique qui examine le code source sans l'exécuter, afin de détecter automatiquement des erreurs de style, des problèmes de syntaxe ou des constructions considérées comme dangereuses ou incohérentes.

Exercice 1 : Flake8

Nous utiliserons `flake8`, un linter simple pour Python, capable d'identifier immédiatement des problèmes de style ou des erreurs de syntaxe. Nous aurions pu utiliser `Checkstyle` pour Java, `Clang-Tidy` pour C/C++ ou encore `Clippy` pour Rust. Par souci de simplicité, nous nous concentrerons sur Python dans ce TP.
Remplacez votre fichier `.gitlab-ci.yml` par le contenu suivant :

```
    stages:
        - lint

    lint_code:
        stage: lint
        image: python:3.10
        script:
            - pip install flake8
            - flake8 .
```

Ajoutez ensuite le fichier `main.py` suivant :

```
    def add(a,b):
        return a+b

    def multiply(x, y)
        return x * y


    print("Result:", add(3,4))
```

1. Poussez votre projet sur le dépôt distant et observez le résultat du pipeline.
2. Corrigez les erreurs signalées, puis poussez à nouveau vos modifications.

Nous allons maintenant configurer `Flake8` selon nos besoins. Pour cela, créez un fichier `.flake8` à la racine du projet. Les règles demandées sont les suivantes :

3. En utilisant la documentation de Flake8 (https://flake8.pycqa.org/en/latest/user/configuration.html), écrivez un fichier .flake8 qui assure les règles suivantes :

- La longueur maximale d'une ligne doit être de 80 caractères.
- Les erreurs liées aux sauts de lignes entre les définitions de fonctions (E301, E302, E303, E305) doivent être ignorées.

4 Tests automatisés

Après avoir vérifié que le code respecte un certain style à l'aide d'un linter, l'étape suivante consiste à s'assurer que ce code se comporte correctement. Pour cela, on met en place des tests automatisés, exécutés à chaque modification du dépôt. L'idée est de détecter le plus tôt possible les erreurs de logique ou de comportement, en complément des erreurs de style ou de syntaxe déjà gérées par `flake8`. Dans GitLab CI, les tests ne sont rien d'autre qu'une étape supplémentaire dans la pipeline, qui lance automatiquement la commande habituelle de tests utilisée en local.

Exercice 2 : Tests

Nous allons maintenant ajouter une étape de tests à la pipeline. Nous utiliserons `pytest`, un framework de tests pour Python. Comme précédemment, toute la configuration sera décrite dans le fichier `.gitlab-ci.yml` et exécutée automatiquement par GitLab.
Modifiez votre fichier `.gitlab-ci.yml` afin d'obtenir la configuration suivante, qui ajoute un stage de tests après le linter :

```
stages:
    - lint
    - test

lint_code:
    stage: lint
    image: python:3.10
    script:
        - pip install flake8
        - flake8 .

run_tests:
    stage: test
    image: python:3.10
    script:
        - pip install -r requirements.txt
        - pytest
```

Nous réutiliserons le `main.py` précédent.
Ajoutez ensuite un fichier `requirements.txt` à la racine du projet, contenant au minimum :

```
pytest
```

Enfin, créez un fichier `test_main.py` qui contiendra les tests automatisés. Voici un premier test d'exemple, que vous pouvez utiliser comme point de départ :

```
import main


def test_add_simple():
    assert main.add(2, 3) == 5
```

1. Complétez le fichier `test_main.py` en ajoutant d'autres tests pour les fonctions `add` et `multiply`, en couvrant plusieurs cas (valeurs négatives, zéros, grands nombres, etc.).
2. Ajoutez une fonction `divide` et les tests associés.
3. Poussez vos modifications sur le dépôt distant et observez le résultat du pipeline. Tous les tests doivent passer pour que le job `run_tests` soit marquée comme réussie.

5 Build et artefacts en C

Jusqu'à présent, nous avons travaillé avec du code Python. Dans cette partie, nous reprenons le même exemple fonctionnel (les fonctions `add` et `multiply`), mais cette fois en langage C. L'objectif est de montrer comment utiliser GitLab CI pour construire un binaire à partir d'un fichier source C, le conserver comme artefact, puis le réutiliser dans une étape suivante pour exécuter des tests. Cette approche est typique des langages compilés, où la phase de *build* produit un exécutable que l'on peut tester ou déployer.
Dans GitLab, les *artefacts* ne sont pas réservés aux seuls langages compilés ni uniquement aux exécutables. Un artefact est simplement un fichier produit par un job et conservé par GitLab afin d'être téléchargé ou transmis aux étapes suivantes de la pipeline.

Exercice 3 : Build et test d'un binaire C

Modifiez votre fichier `.gitlab-ci.yml` afin de remplacer les étapes précédentes par la configuration suivante, qui construit un binaire C puis le teste dans un second job :

```
stages:
    - build
    - test

build_c:
    stage: build
    image: gcc:latest
    script:
        - gcc -Wall -Wextra -o calc main.c
    artifacts:
        paths:
            - calc

test_c:
    stage: test
    image: gcc:latest
    dependencies:
        - build_c
    script:
        - ./calc > output.txt
        - 'grep␣"Addition:␣7"␣output.txt'
```

Le job `build_c` utilise l'image `gcc` pour compiler le fichier `main.c` et produire un binaire nommé `calc`. Ce binaire est ensuite enregistré comme artefact afin d'être récupéré par le job suivant `test_c`. Le job `test_c` exécute ce binaire, redirige sa sortie standard vers un fichier `output.txt`, puis vérifie, à l'aide de `grep`, que le texte attendu apparaît bien dans la sortie du programme.
Créez maintenant un fichier `main.c` contenant le code suivant :

```
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

int multiply(int x, int y) {
    return x * y;
}

int main(void) {
    int a = 3;
    int b = 4;

    printf("Addition:␣%d\n", add(a, b));
```

```c
        printf("Multiplication: %d\n", multiply(a, b));

        return 0;
    }
```

Ce programme reprend exactement la même logique que la version Python vue précédemment : il calcule une addition et une multiplication, puis affiche les résultats. Le test fourni dans le job `test_c` vérifie uniquement la présence de la ligne `"Addition: 7"` dans la sortie du programme. Il s'agit d'un premier exemple de test automatisé sur le binaire produit.

1. Poussez votre projet sur le dépôt distant et observez les résultats des jobs `build_c` et `test_c` dans la pipeline.
2. Complétez le script de test (section `script` du job `test_c`) en ajoutant une vérification similaire pour la ligne `"Multiplication: 12"`.
3. Modifiez temporairement l'implémentation de la fonction `multiply` dans `main.c` pour introduire une erreur de calcul, puis poussez à nouveau vos modifications. Observez la manière dont GitLab signale l'échec de du job `test_c`.

6 Déploiement

Dans une chaîne CI/CD complète, la dernière étape après le lint, les tests et la construction des *artefacts* consiste à déployer automatiquement l'application sur un environnement d'exécution. Le *déploiement* désigne l'ensemble des opérations permettant de rendre un logiciel disponible sur une machine cible, qu'il s'agisse d'un serveur, d'une machine virtuelle, d'un conteneur, d'un cluster, ou même d'une plateforme cloud. Dans une pipeline GitLab, cela prend généralement la forme d'un job supplémentaire qui récupère les *artefacts* produits au cours du build, puis les transfère, les installe ou les met en service sur l'environnement prévu.
Le déploiement sert à automatiser la mise en production ou la mise à disposition d'une nouvelle version du logiciel. Il garantit que chaque modification validée par la pipeline est effectivement et correctement déployée, sans intervention manuelle. Dans les environnements professionnels, le déploiement peut également inclure des actions beaucoup plus avancées.
Toutefois, nous ne mettrons pas en place de processus de déploiement dans ce TP. Le déploiement automatisé implique en effet de disposer d'une infrastructure serveur réelle ou simulée. Ces aspects dépassent le cadre de cette séance d'introduction, dont l'objectif est de se concentrer sur la compréhension des mécanismes fondamentaux de la CI, sans aborder la configuration d'une infrastructure de production.