IUT PARIS CITE
BUT VCOD
2025-2026

TP : Versioning and CI-CD

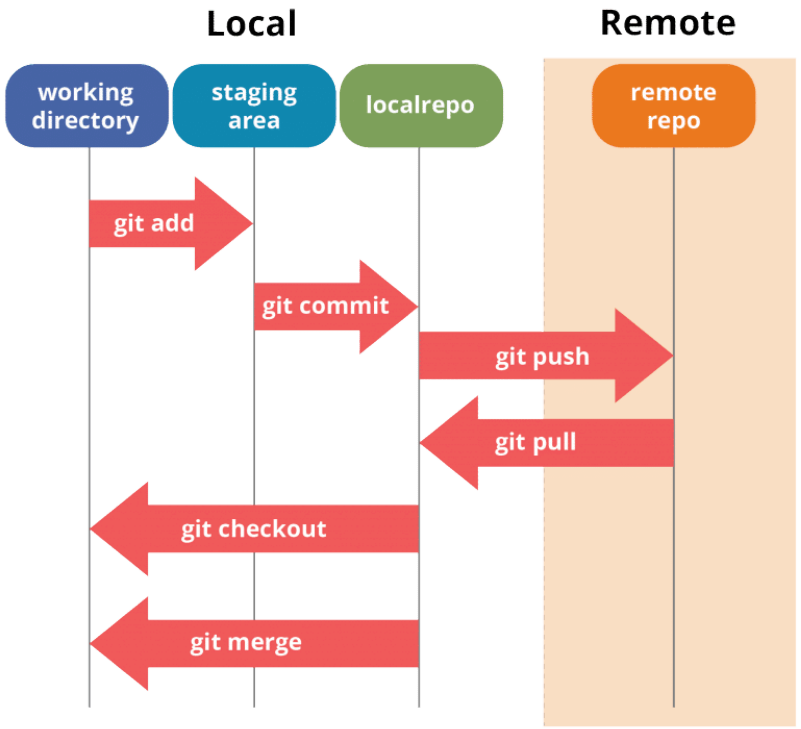
Pré-requis : afin de réaliser ce TP, vous devez :
Disposer de python et d'un IDE type vscode https://code.visualstudio.com/
Avoir installé GIT sur votre machine Install GIT
Avoir un compte sur github et gitlab

===== LAB1 : Initiation à GIT (via Vscode) =====

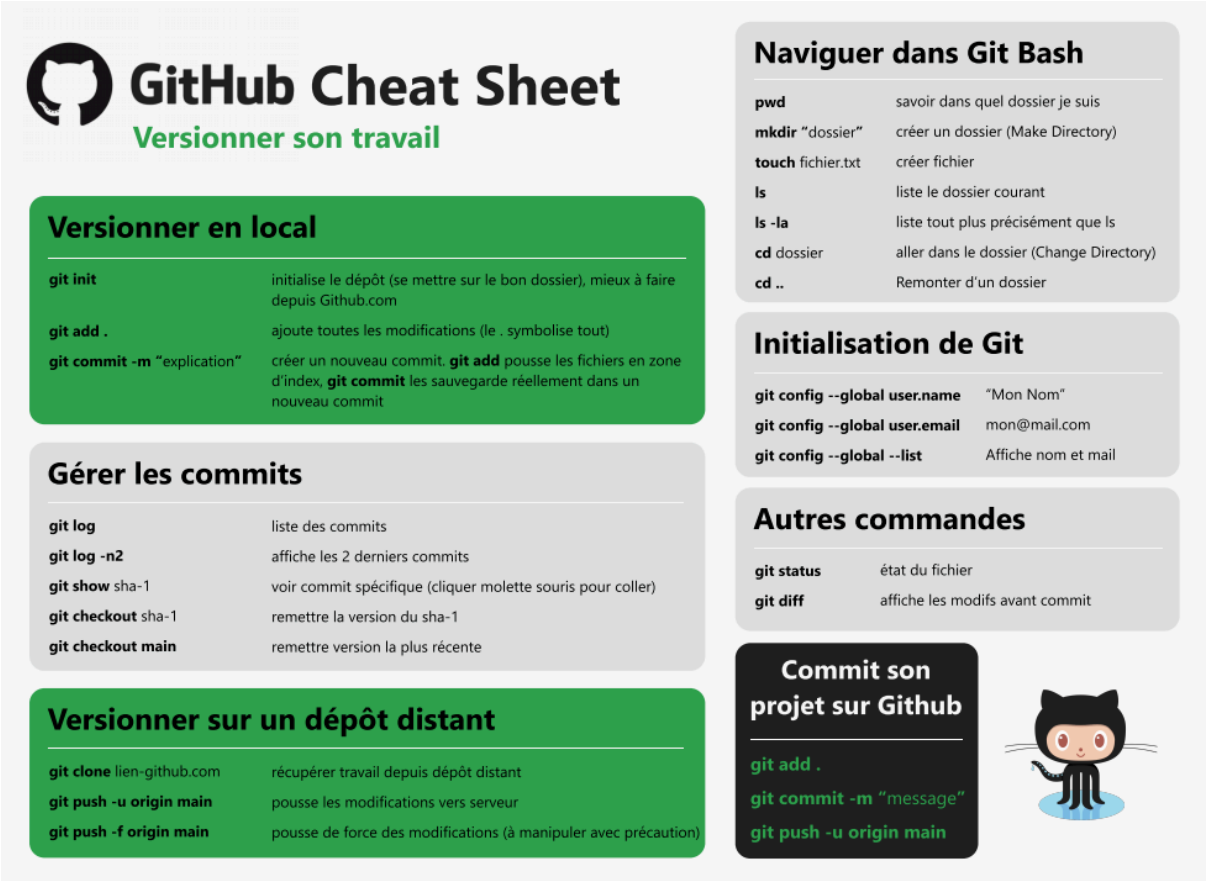
L'objectif principal de ce TP est une initiation au contrôle de version à l'aide de **Git**, en particulier depuis un IDE comme Visual Studio Code (VSCode) au travers de :

- ➔ Mise en place d'un projet Python simple,
- ➔ Initialisation d' un répertoire Git,
- ➔ Effectuer des modification de code dans le projet
- ➔ Envoyer les modifications de code vers GitHub.

Rappel des bases de GIT et GitHub



source



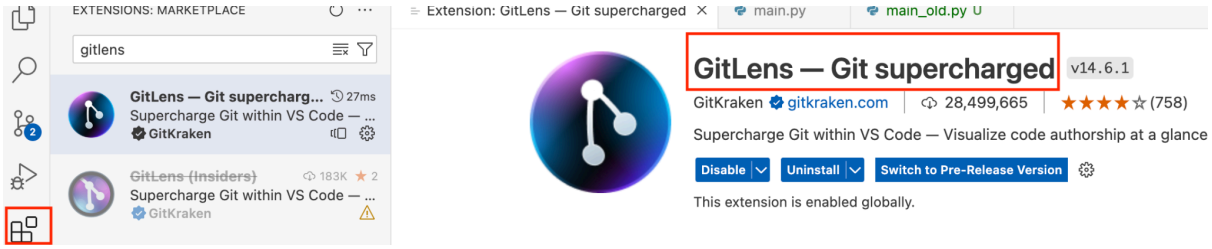
[Github Cheatsheet]

Etape 1: Paramétrage de VSCode et de Git

⇒Si ce n'est pas déjà installé :

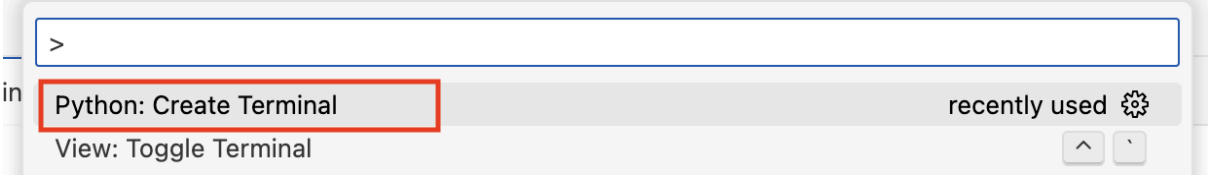
- Installer Visual Studio Code (VSCode)
- Installer GIT [Install GIT]

⇒Dans VSCode, installer l'extension "GitLens" depuis la vue extensions



Etape 2: Créer votre projet python

Créer un nouveau répertoire pour votre projet python en utilisant le terminal VSCode :



```
mkdir test-git-vscode
cd test-git-vscode
```

Créer un fichier python "main.py" and ajouter un script python simple, par exemple :

```python
# Program to check if a number is prime or not

num =  67

# To take input from the user
#num = int(input("Enter a number: "))

# define a flag variable
flag = False

if num == 1:
    print(num, "is not a prime number")
elif num > 1:
    # check for factors
    for i in range(2, num):
        if (num % i) == 0:
            # if factor is found, set flag to True
            flag = True
            # break out of loop
            break

    # check if flag is True
    if flag:
        print(num, "is not a prime number")
    else:
        print(num, "is a prime number")
```

Etape 3: Initialiser un répertoire GIT

Initialiser un répertoire Git repository au sein de votre répertoire project "test-git-vscode" :

```
git init
```

Ajouter le fichier "main.py" à la staging area (il s'agit de faire un **add** au sens GIT, voir le cours pour rappel des concepts) :

```
git add main.py
```

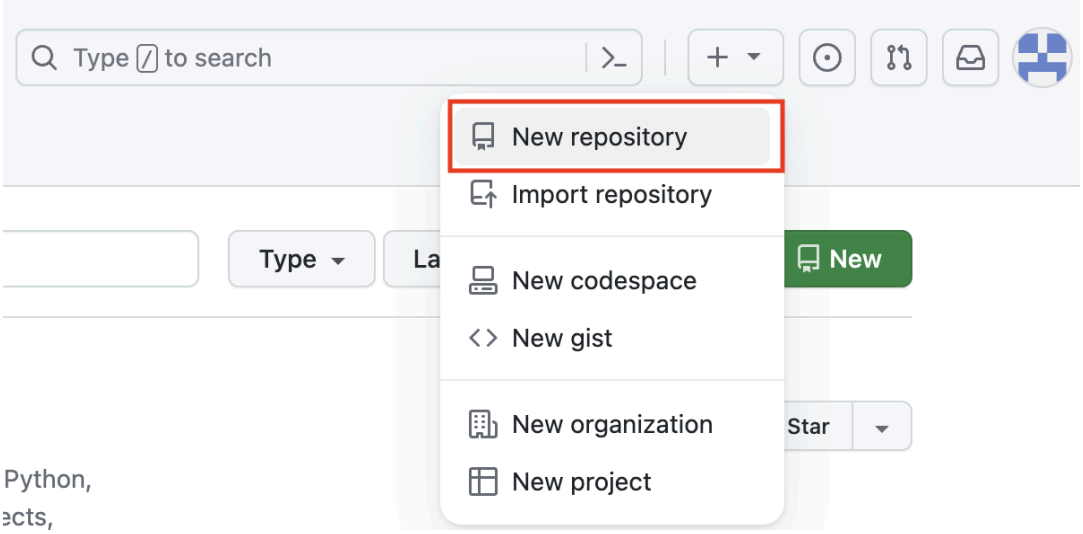
Faire un **commit** vos changements :

```
git commit -m "commit initial"
```

Etape 4: Créer un répertoire GitHub

Aller sur GitHub et créer un compte si ce n'est pas déjà fait ou connectez vous à votre compte existant.

Créer un nouveau répertoire via le bouton "+" en haut à droite :



Suivez les étapes pour créer un répertoire, utilisez le même nom que votre répertoire local "test-git-vscode"

Copier l'URL du répertoire créé qui est de la forme
https://github.com/votre-compte/test-git-vscode

Etape 5: Connecter votre répertoire local GitHub

Associer votre répertoire GitHub comme répertoire distant (remote) de votre répertoire local via la commande ci-dessous dans le terminal vscode :

```
git remote add origin <coller_url_repertoire_github>
```

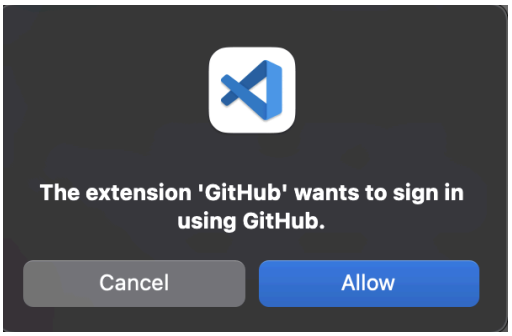
→*Remarque :*

pour changer l'URL du répertoire (github, gitlab..) il faut utiliser
git remote set-url origin <nouvelle_url>

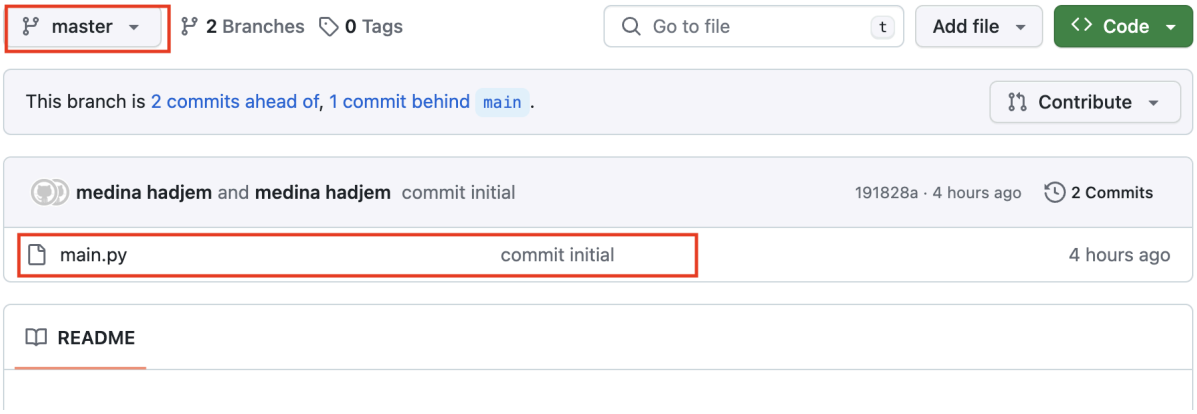
Pousser (Push) votre code depuis votre répertoire local vers GitHub via cette commande depuis le terminal vscode :

```
git push -u origin master
```

Cette commande nécessite que l'extension Github de vscode puisse avoir les permissions pour se connecter à votre compte Github, cliquer sur "Allow" lorsque cette fenêtre s'affiche :



Ceci va permettre d'envoyer le code "main.py" sur la branche master de votre répertoire github :



Etape 6: Faire des changements et mettre à jour le répertoire GitHub

Ouvrir le script "`main.py`" dans VSCode, effectuer quelques changements et enregistrer ces changements (ajouter un commentaire par exemple).

Ajouter ces changements à la staging area (add) et faire un commit

```
git add main.py
git commit -m "Update main.py"
```

Envoyer les changements (push) à votre répertoire GitHub, cette fois vous n'avez pas besoin de spécifier la branche (-u origin master), par défaut il va garder la même "master" :

```
git push
```

Etape 7: Créer une nouvelle branche Git et Merger avec la branche master

Créer une nouvelle branche git dans votre répertoire local :

```
git branch new-branch
```

Changer de branche pour aller vers la nouvelle depuis votre répertoire local :

```
git checkout new-branch
```

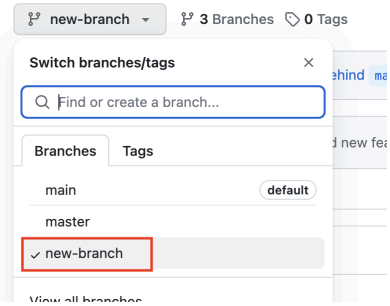
Faire de nouveau des changements dans “main.py” dans cette nouvelle branche et commit ces changements :

```
git add main.py
git commit -m "Add new feature"
```

Envoyer cette branche vers github

```
git push origin new-branch
```

Aller sur Github pour vérifier que la nouvelle branche **new-branch** a bien été ajoutée :



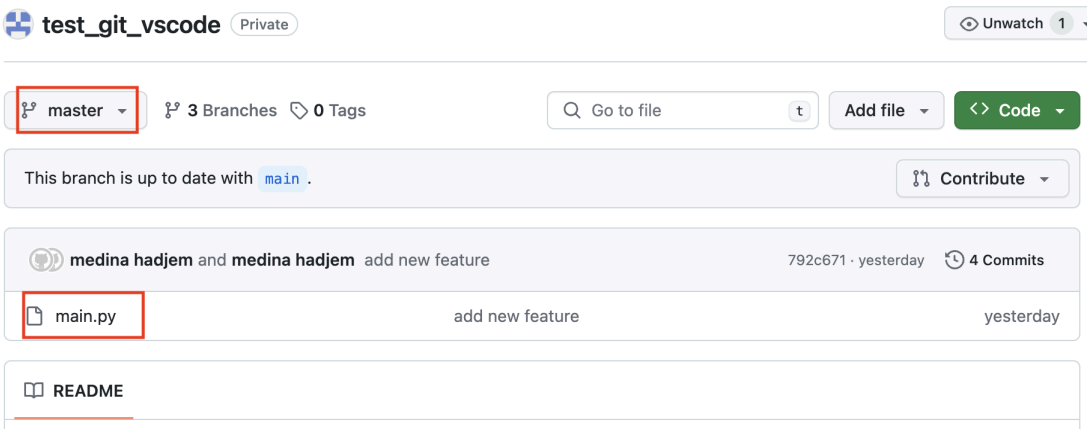
Maintenant on va merger les modifications de cette branche avec la branche principale master, il faut revenir vers la branche master :

```
git checkout master
```

Merger la branche **new-branch** dans la branche **master** et envoyer vers la branche master sur github (push):

```
git merge new-branch
git push origin master
```

Vérifier sur github que la branche master a été modifiée avec la nouvelle version de main.py



Etape 8: Créer une nouvelle branche Git et faire une pull request

Créer une nouvelle branche git dans votre répertoire local :

```
git branch new-branch2
```

Changer de branche pour aller vers la nouvelle depuis votre répertoire local :

```
git checkout new-branch2
```

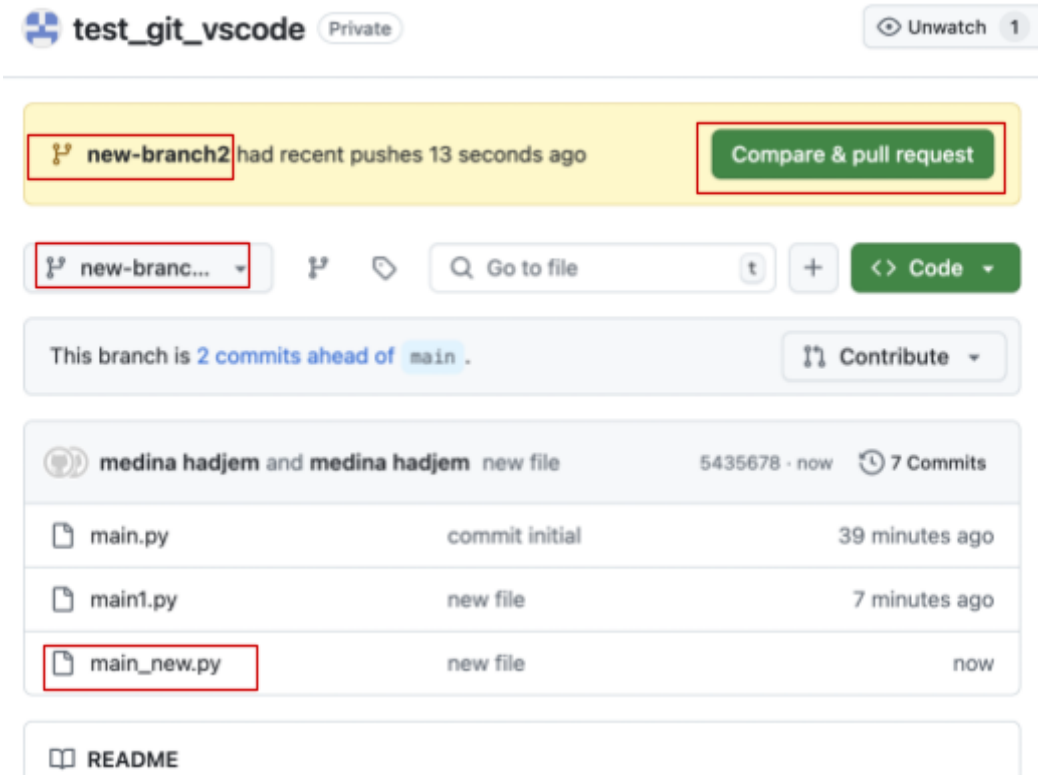
Créer un nouveau fichier “main_new.py” dans votre répertoire local et commit ces changements :

```
git add main_new.py
git commit -m "Add new code"
```

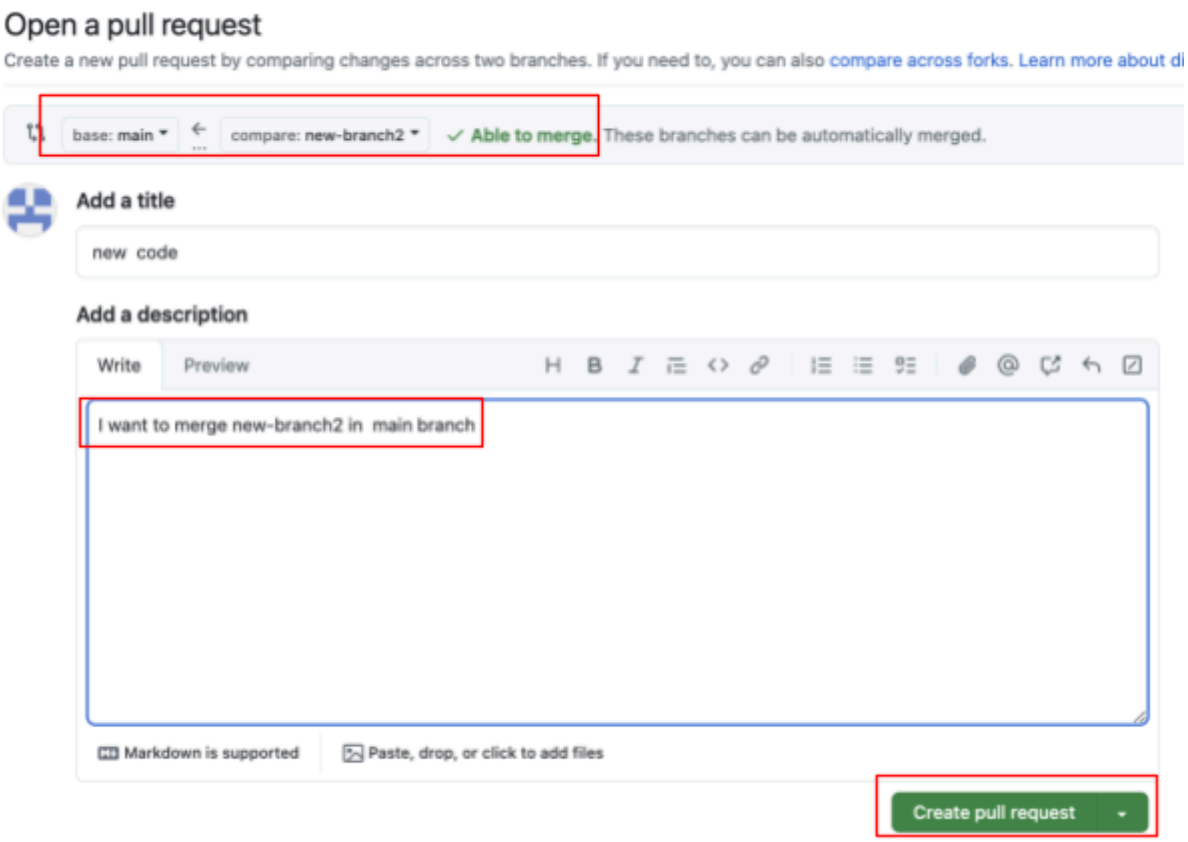
Envoyer cette branche vers github

```
git push origin new-branch2
```

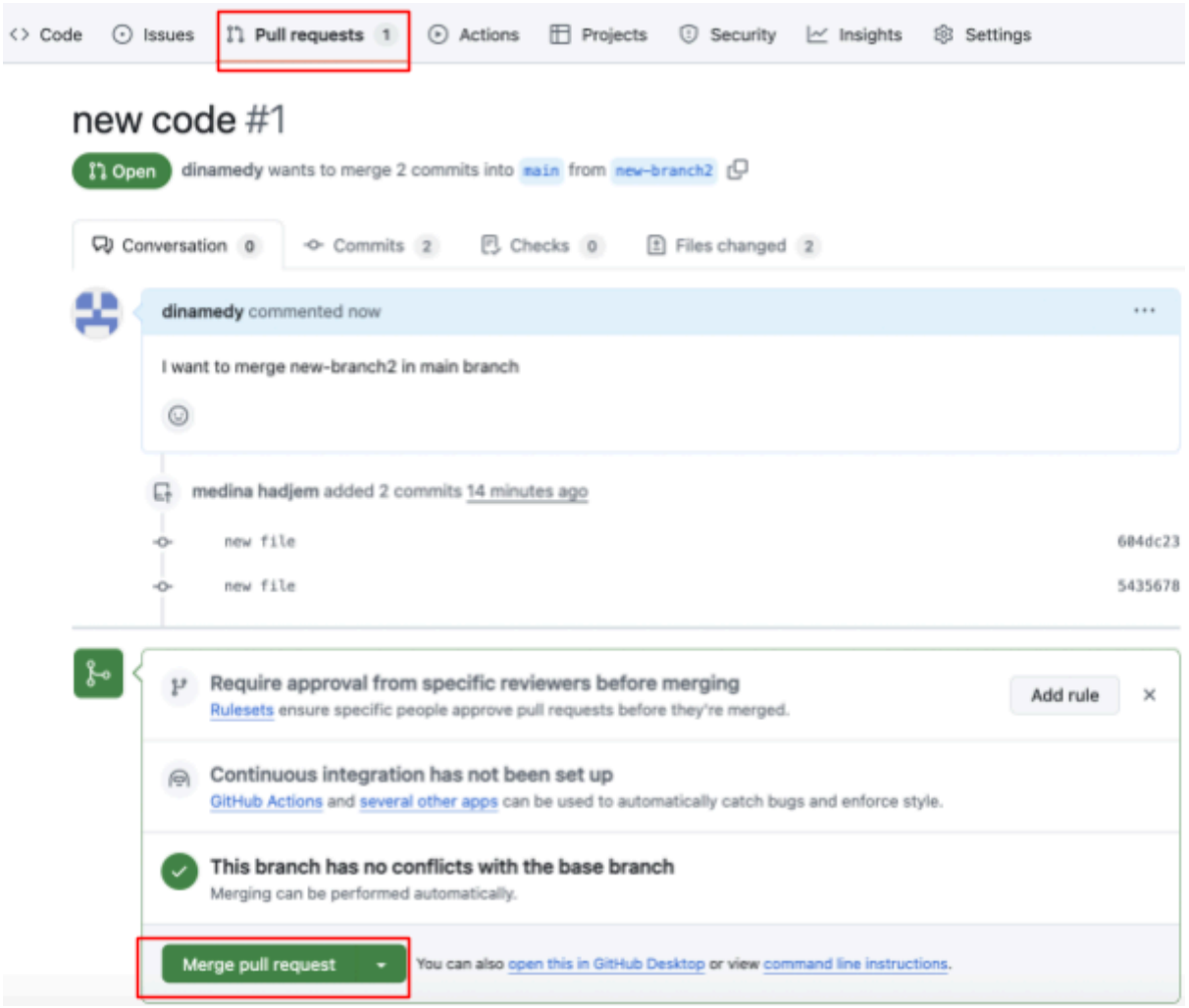
Vous allez voir sur le répertoire github distant :
la nouvelle branche **new-branch2** avec le fichier main_new.py
et l’option **“Compare & pull request”**



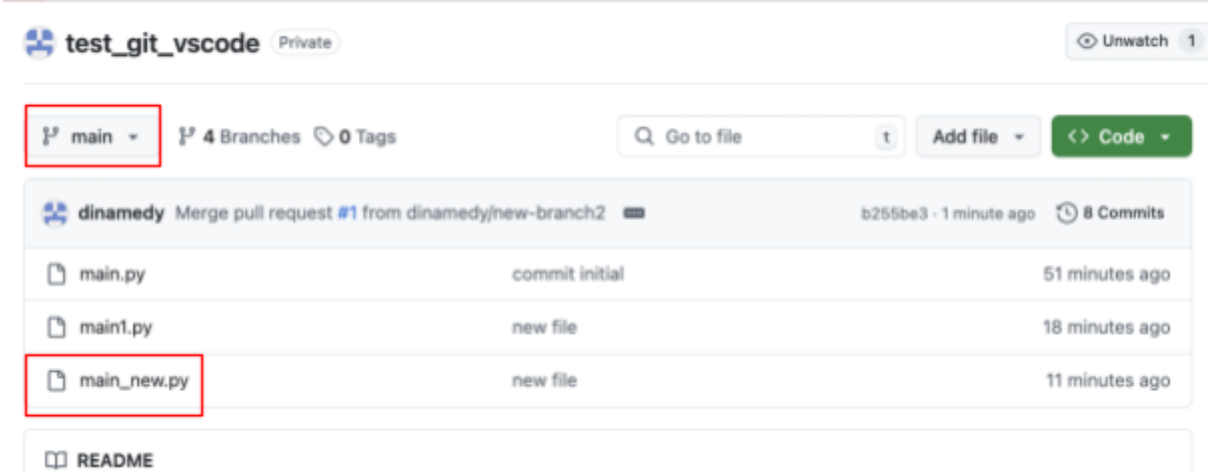
Cliquer sur **“Compare & pull request”**pour créer la pull request, mettez un titre et une description de votre pull request et cliquer sur **“create pull request”**



Allee sur **“pull requests”** pour visualiser la pull request et cliquer sur **“Merge pull request”** puis **“confirm merge”**



L'ensemble du code de new-branch2 a été mergé avec main au travers de cette pull request



== LAB2 : Introduction à GitHub Actions pour automatiser les tests ==

Objectifs :

1. Comprendre les bases de GitHub Actions.
2. Configurer un workflow CI (Intégration Continue) pour exécuter des tests Python automatiquement.
3. Appliquer les bonnes pratiques pour gérer un pipeline CI simple

Étape 1 : Créer un Projet Python Simple

- Créer un répertoire GitHub :
  - Allez sur GitHub et créez un nouveau dépôt (par exemple, `github-actions-lab`).
  - Initialisez-le avec un fichier `README.md`.
- Initialiser le projet localement en clonant le dépôt sur votre machine depuis le terminal vscode (remplacer username par votre compte) :

```
git  clone https://github.com/username/github-actions-lab.git
cd github-actions-lab
```

- Ajoutez du Code Simple au répertoire local `github-actions-lab` :

⇒Créez un fichier `math_utils.py` :

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

def subtract(a, b):
  return a - b
```

⇒Créez un fichier `test_math_utils.py` pour les tests :

```
from math_utils import add, subtract

def test_add():
  assert add(2, 3) == 5
  assert add(-1, 1) == 0

def test_subtract():
```

```
  assert subtract(5, 3) == 2
  assert subtract(0, 3) == -3
```

- Installer pytest :

Ajoutez pytest à votre environnement local :

```
pip install pytest
```

Testez le code localement :

```
pytest
```

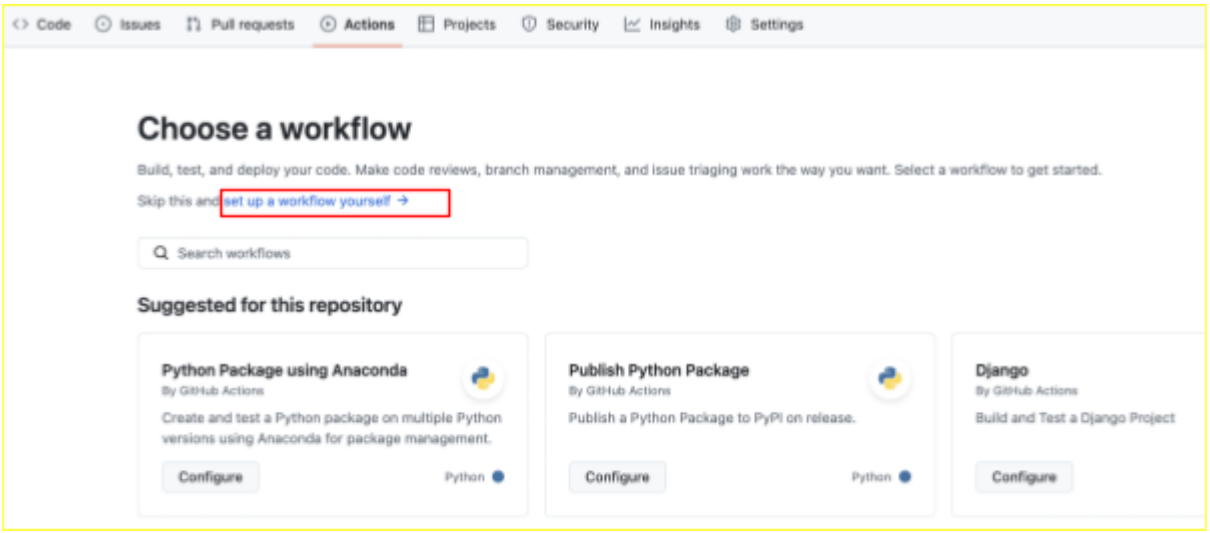
- Commit les fichiers :

Ajoutez les fichiers au dépôt GitHub :

```
git add .
git commit -m "Ajout du code et des tests de base"
git push origin main
```

Étape 2 : Configurer GitHub Actions pour votre répertoire

1. Accédez à l'onglet "Actions" sur github :
   - Allez sur la page de votre répertoire GitHub `github-actions-lab`.
   - Cliquez sur l'onglet Actions.
   - GitHub suggère plusieurs workflows, pour ce Lab, cliquez sur "set up a workflow yourself".



2. Créer un Workflow :
    - GitHub crée un fichier YAML “main.yml” par défaut dans `.github/workflows`.

Modifiez le fichier pour configurer un workflow Python CI :

```
name: Python CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Setup Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.9'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install pytest

    - name: Run tests
      run: pytest
```

3. Commit le Workflow :
    - Cliquez sur "Start Commit", ajoutez un message de commit “`ajout du workflow GitHub Actions`”, et commit le fichier directement sur la branche main de votre répertoire github.

Étape 3 : Observer l'Exécution

1. Déclenchement Automatique :
    - GitHub Actions s'exécute automatiquement à chaque push ou pull request sur la branche `main`.
    - Accédez à l'onglet Actions pour voir les workflows en cours d'exécution.
2. Vérifiez le Résultat :
    - Cliquez sur le workflow pour afficher les détails.
    - Assurez-vous que toutes les étapes (`Checkout code`, `Setup Python`, `Install dependencies`, `Run tests`) se terminent avec succès.

Étape 4 : Corriger les Erreurs

1. Simulation d'une Erreur :

Modifiez `math_utils.py` pour introduire une erreur :

```
def add(a, b):
  return a - b  # Erreur intentionnelle
```

Poussez le changement :

```
git commit -am "Simulation d'une erreur"
git push origin main
```

2. Observer l'Échec :
    - Retournez à l'onglet Actions et observez que le workflow échoue.
    - GitHub fournit des messages détaillés pour identifier l'erreur.
3. Corrigez le Code :

Réparez l'erreur et commit push à nouveau :

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

```
git commit -am "correction de l'erreur"
git push origin main
```

    - GitHub Actions relancera automatiquement le workflow .

Étape 5 : Amélioration Supplémentaire

1. Ajoutez du Linting avec `flake8` :

Modifiez le fichier YAML `.github/workflows/main.yml` pour inclure le linting :

```
- name: Run linter
 run: pip install flake8 && flake8 math_utils.py
```

=====LAB3 : Initiation à la CI-CD via GitLab =====

L'objectif principal de ce LAB est une initiation à l'intégration continue (CI) à l'aide de GitLab et en particulier en utilisant GitLab-CI.

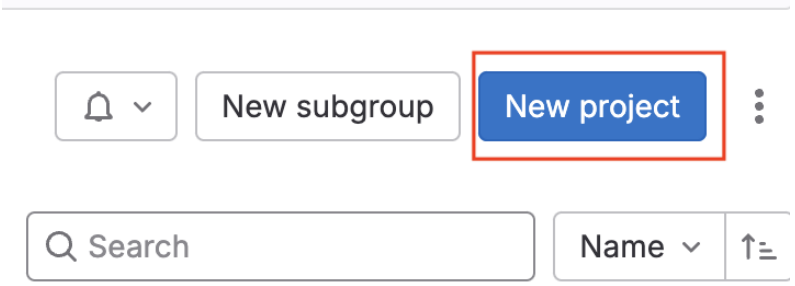
Il s'agit de mettre en place un pipeline CI de base pour un projet Python hébergé sur votre compte GitLab.

Etape 1: Créer votre compte sur GitLab

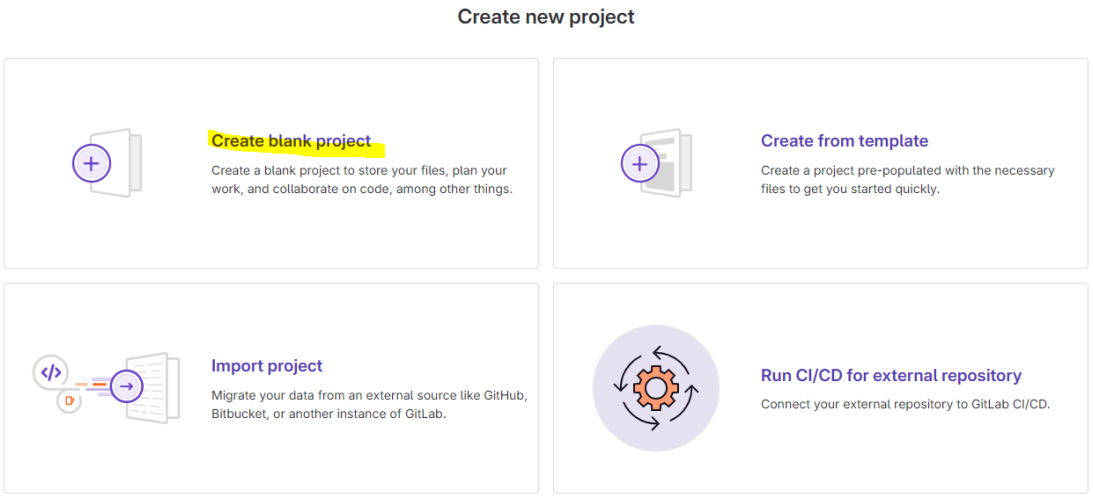
⇒Si ce n'est pas déjà fait créer votre compte gratuit sur gitlab sign up en suivant les étapes.

Etape 2 : créer un nouveau projet sur votre espace GitLab

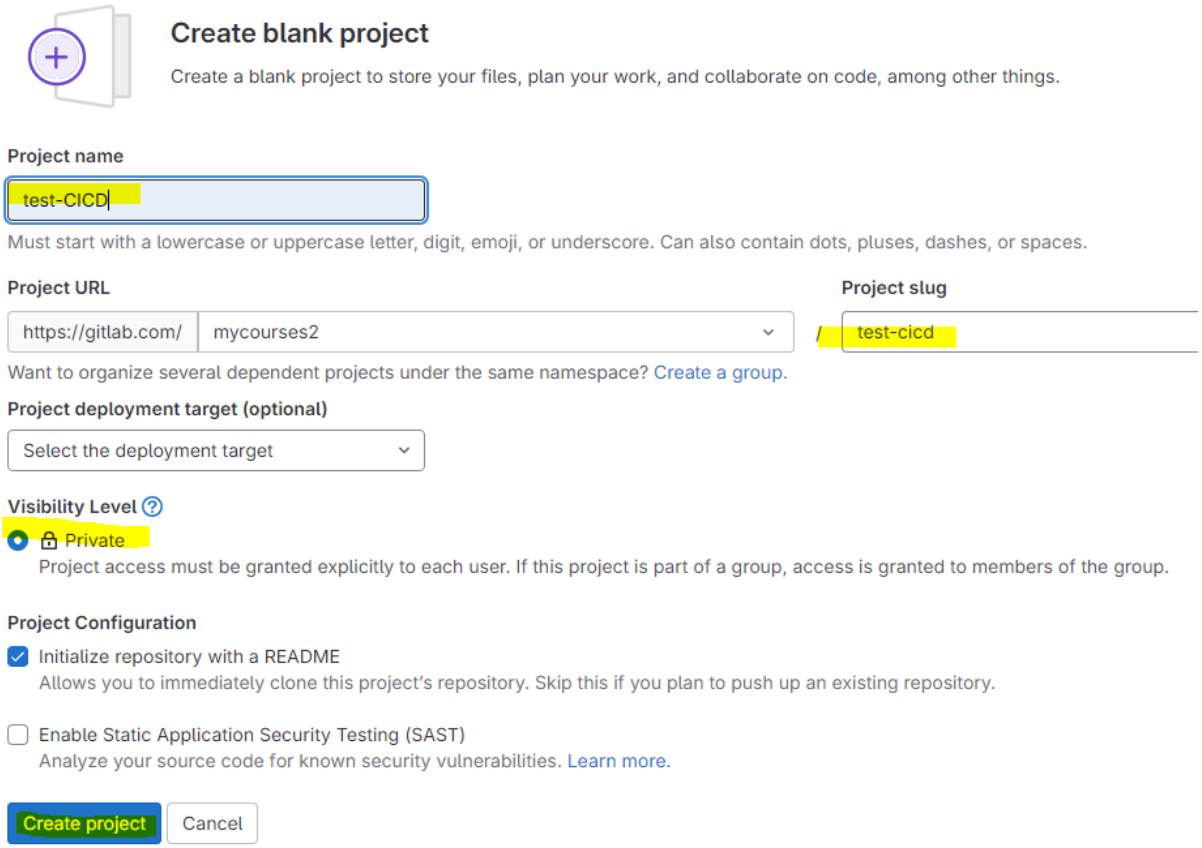
une fois votre espace gitlab crée, aller dans "New project" à droite :



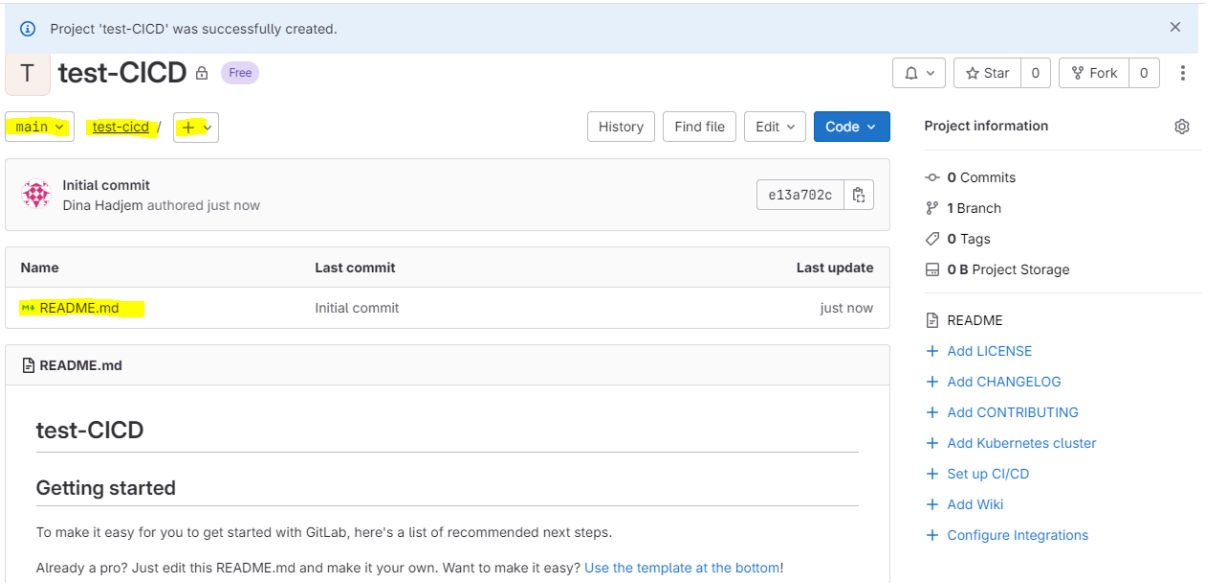
Plusieurs options sont possibles pour créer un projet, nous allons partir d'un projet vide :



Sélectionner **"Create blank project"** et choisir un nom pour votre projet puis cliquer sur **"Create project"** :



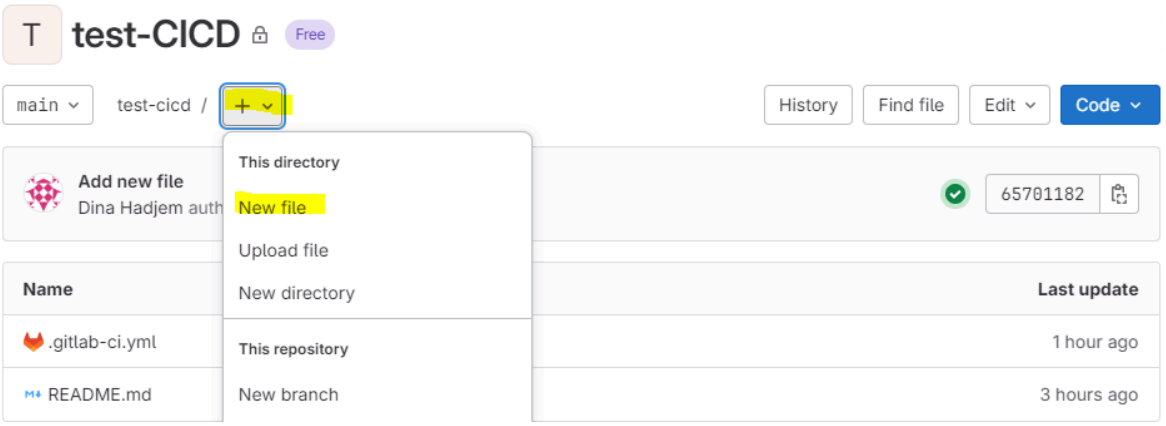
Votre projet va se créer avec une branche “main” et un fichier “README.md”.



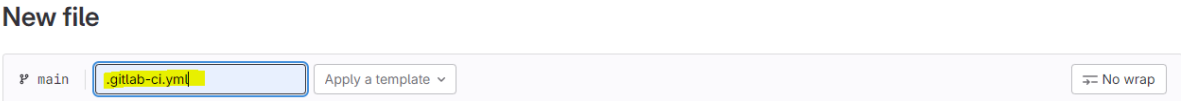
Etape 3 : créer un fichier “.gitlab-ci.yml” de paramétrage CICD

Le fichier “.gitlab-ci.yml” est un fichier au format YAML qui permet de paramétrer une CICD d’exemple pour ce projet ,

Cliquez sur le bouton “+” puis sur “**New file**”



Entrer "**.gitlab-ci.yml**" dans le nom du fichier (ne pas oublier le "." au début), ne pas appliquer de template :



dans ce cas nous allons créer un paramétrage basique d'exemple qui exécute des affichages de messages au lieu d'un script de code, pour ce faire, ajouter la configuration ci-dessous (disponible également depuis le tutorial https://docs.gitlab.com/ee/ci/quick_start/)

```
build-job:
  stage: build
  script:
    - echo "Hello, $GITLAB_USER_LOGIN!"

test-job1:
  stage: test
  script:
    - echo "This job tests something"

test-job2:
  stage: test
  script:
    - echo "This job tests something, but takes more time than test-job1."
    - echo "After the echo commands complete, it runs the sleep command
for 20 seconds"
    - echo "which simulates a test that runs 20 seconds longer than
test-job1"
    - sleep 20

deploy-prod:
  stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
  environment: production
```

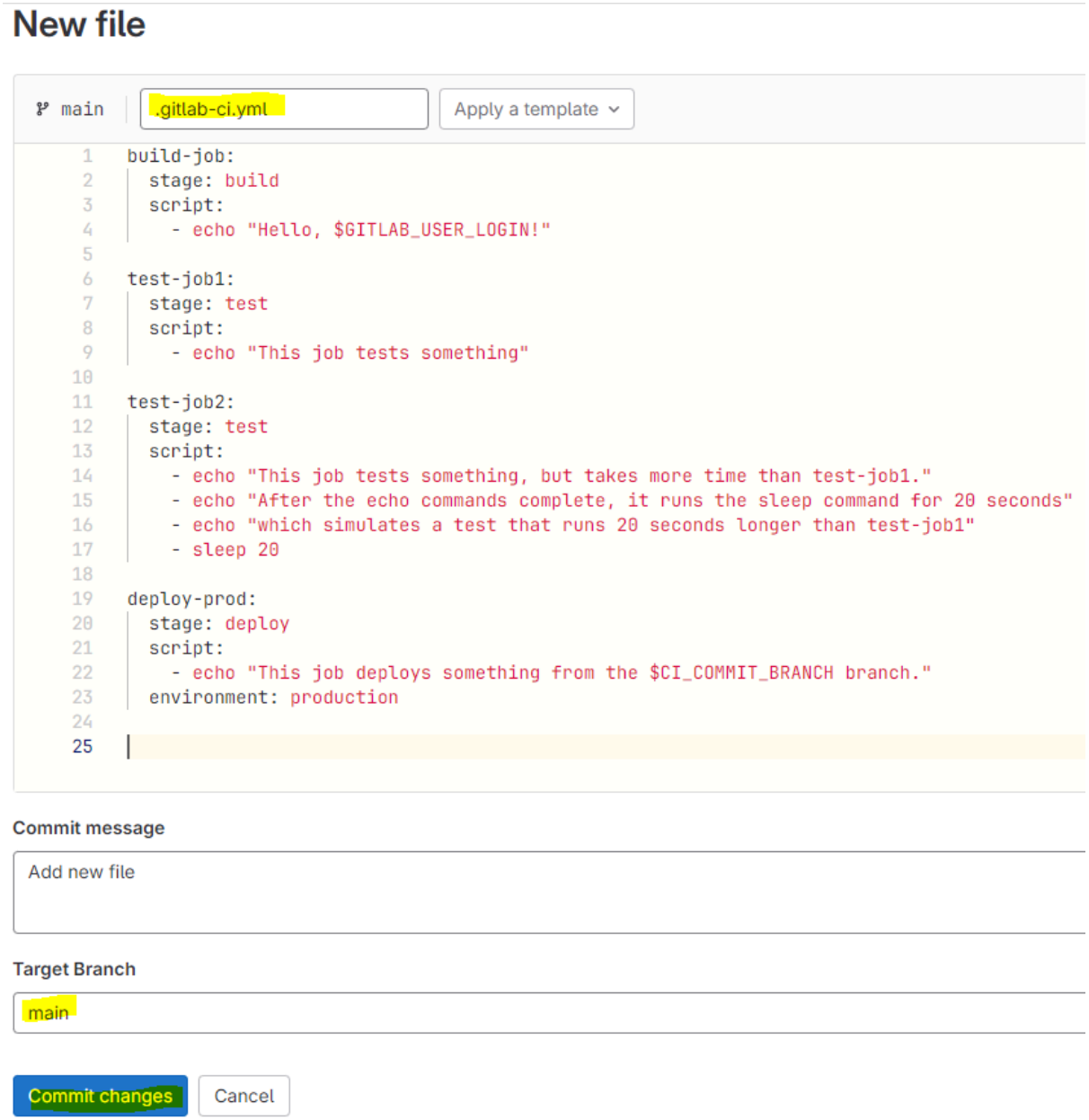
Dans cette configuration d'exemple, 4 jobs (tâches) sont déclarés :

build-job : tâche d'intégration (fait partie de la **CI, Continuous Integration** vu en cours), ici elle affiche le message : "Hello, $GITLAB_USER_LOGIN!"

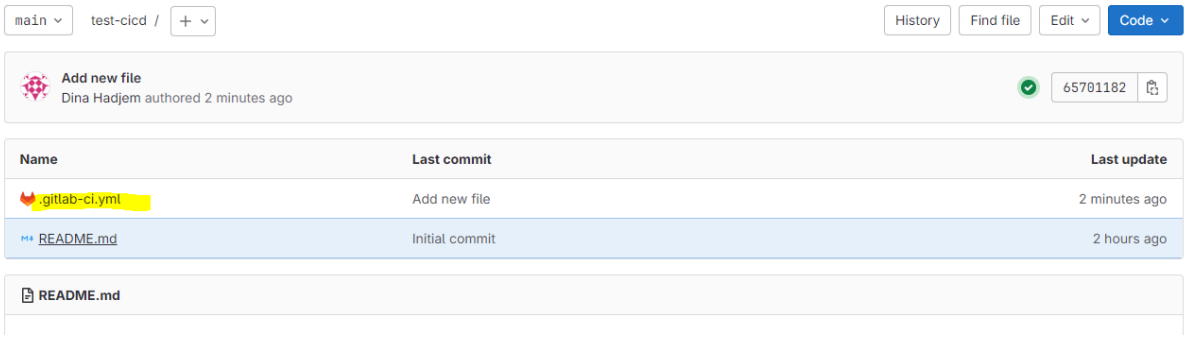
test-job1 et **test-job2** : tâche de test (fait partie de la **CI, Continuous Integration** vu en cours), ici ces deux tâches affichent des messages également

deploy-prod : tâche de déploiement (fait partie de la **CD, continuous Deployment** vu en cours)

Cliquez sur le bouton **"commit changes"** en bas :

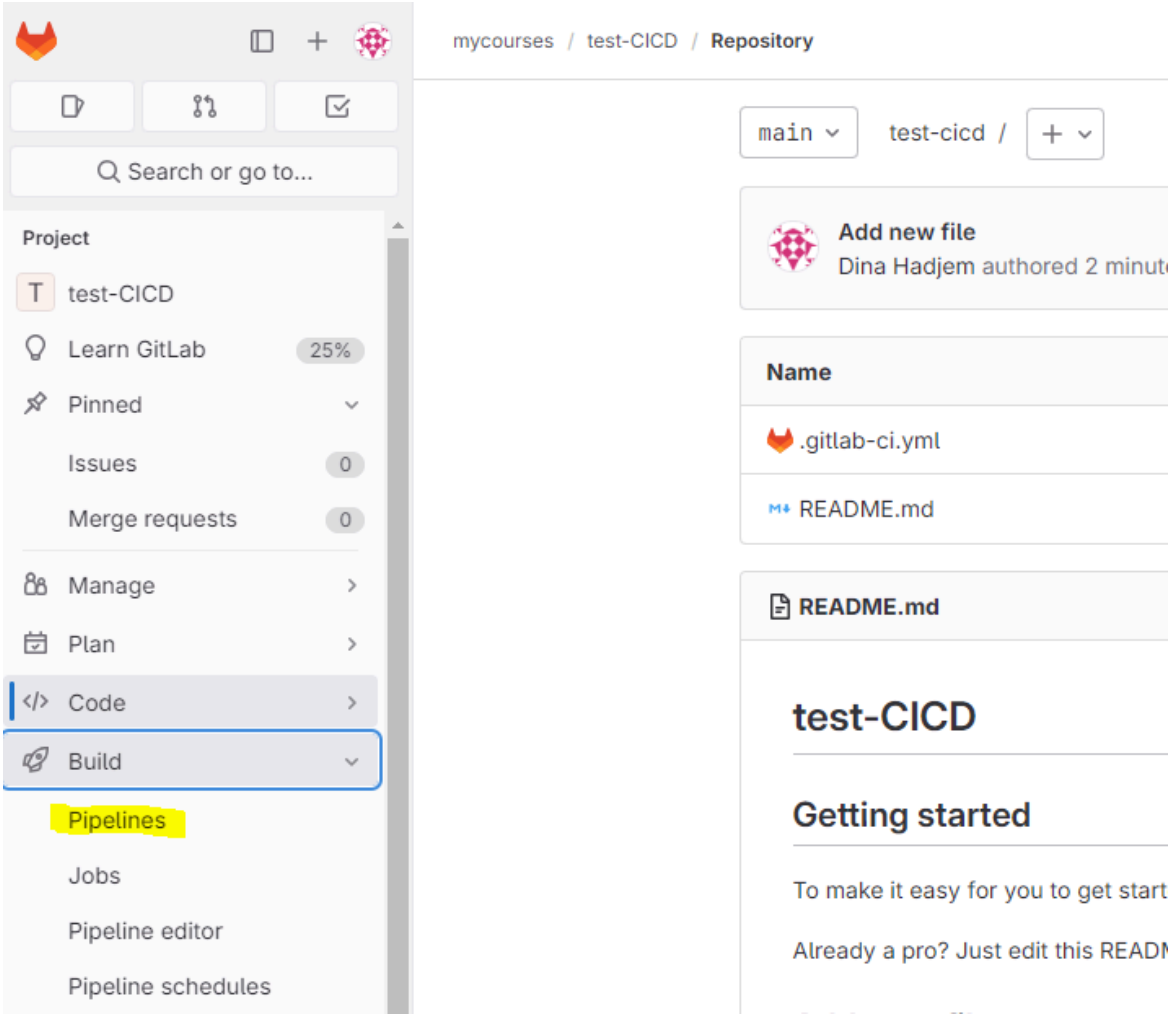


Le fichier "**.gitlab-ci.yml**" a été enregistré et exécuté :

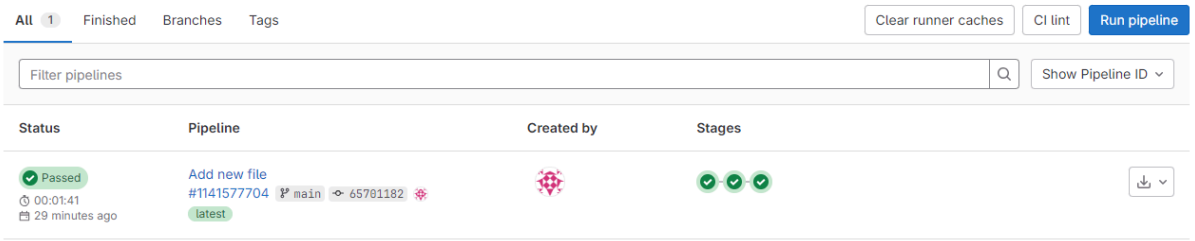


Etape 4 : Vérifier l'exécution de la pipeline CICD correspondant au ".gitlab-ci.yml"

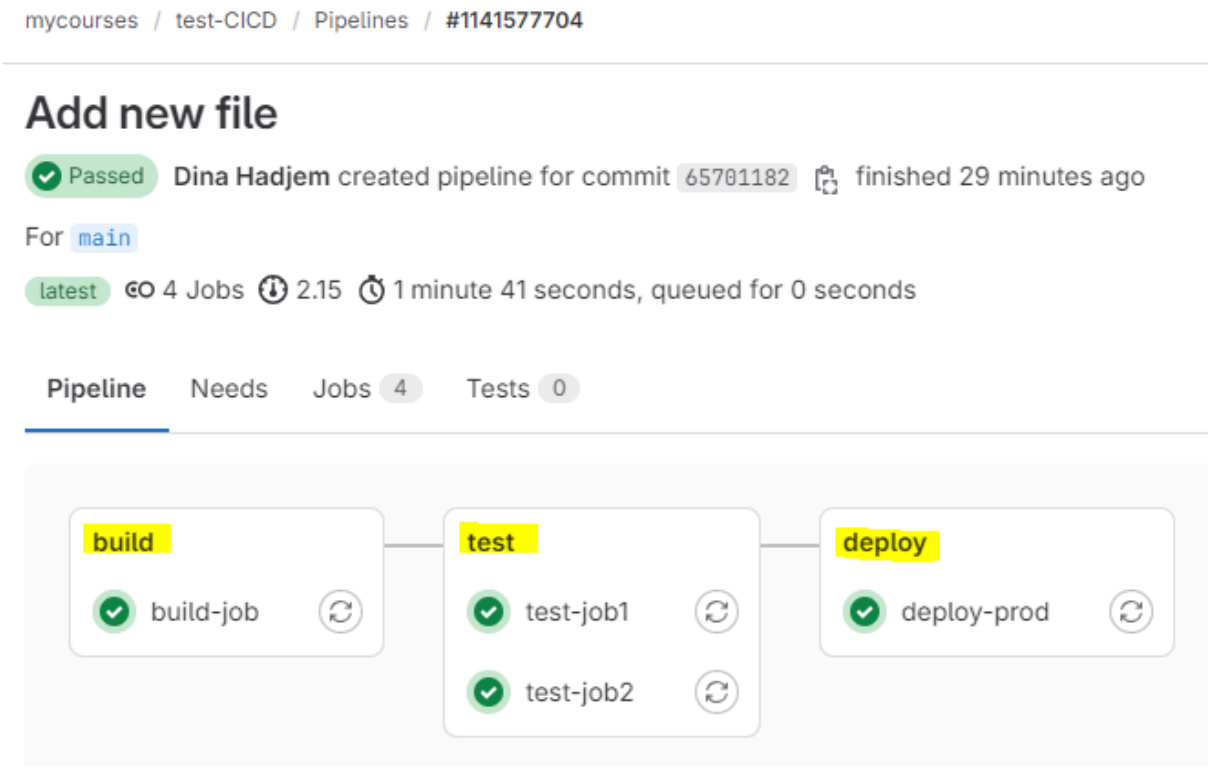
Aller dans le menu gauche dans "**Build**" puis "**Pipelines**" :



Après l'exécution du fichier "**.gitlab-ci.yml**" , l'ensemble des 4 tâches déclarées ont été exécutées, comme on peut le voir ci-dessous avec le statut "passed" et les stages tous en vert.



On peut aussi vérifier l'exécution de chaque job (tâche) en cliquant sur la pipeline :

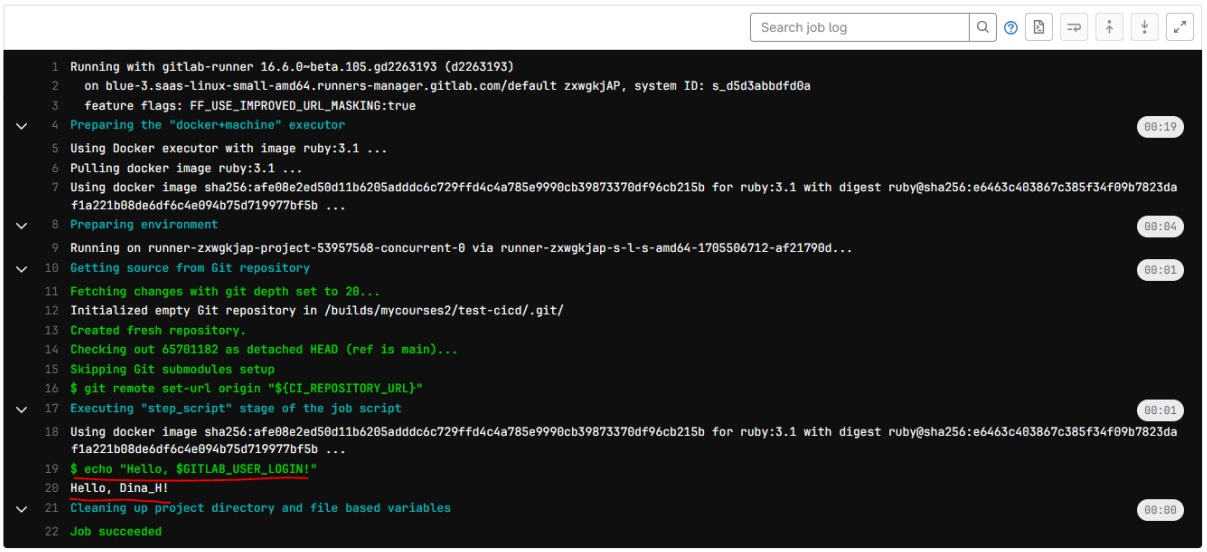


On peut consulter le détail (logs) de chaque job en cliquant sur son nom :

On peut notamment voir que les commentaires ajoutés dans les commandes **echo** du fichier "**.gitlab-ci.yml**" vont être visibles ici. Les valeurs des variables prédéfinies $GITLAB_USER_LOGIN et $CI_COMMIT_BRANCH sont renseignées lors de l'exécution des jobs.

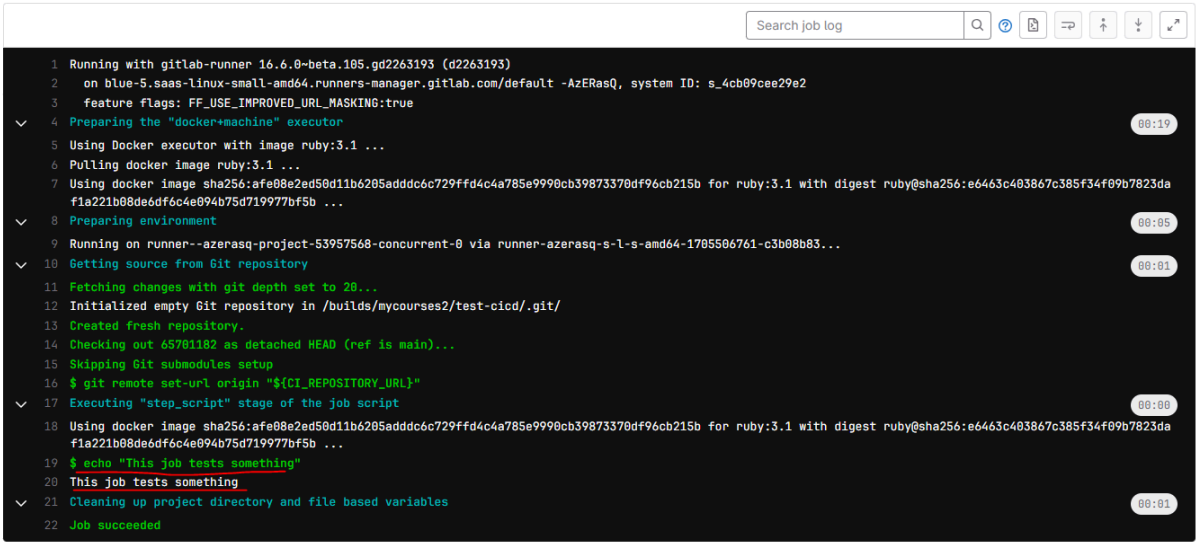
build-job

Passed Started 2 hours ago by Dina Hadjem



test-job1

Passed Started 2 hours ago by Dina Hadjem



deploy-prod

Passed Started 2 hours ago by Dina Hadjem

This job is deployed to production.

Search job log

```
Running with gitlab-runner 16.6.0~beta.105.gd2263193 (d2263193)
  on blue-5.saas-linux-small-amd64.runners-manager.gitlab.com/default -AzERasQ, system ID: s_4cb09cee29e2
  feature flags: FF_USE_IMPROVED_URL_MASKING:true
Preparing the "docker+machine" executor                                    00:19
Using Docker executor with image ruby:3.1 ...
Pulling docker image ruby:3.1 ...
Using docker image sha256:afe08e2ed50d11b6205adddc6c729ffd4c4a785e9990cb39873370df96cb215b for ruby:3.1 with digest ruby@sha256:e6463c403867c385f34f09b7823daf1a221b08de6df6c4e094b75d719977bf5b ...
Preparing environment                                                      00:05
Running on runner--azerasq-project-53957568-concurrent-0 via runner-azerasq-s-l-s-amd64-1705506816-4e3c1687...
Getting source from Git repository                                         00:00
Fetching changes with git depth set to 20...
Initialized empty Git repository in /builds/mycourses2/test-cicd/.git/
Created fresh repository.
Checking out 65701182 as detached HEAD (ref is main)...
Skipping Git submodules setup
$ git remote set-url origin "${CI_REPOSITORY_URL}"
Executing "step_script" stage of the job script                            00:01
Using docker image sha256:afe08e2ed50d11b6205adddc6c729ffd4c4a785e9990cb39873370df96cb215b for ruby:3.1 with digest ruby@sha256:e6463c403867c385f34f09b7823daf1a221b08de6df6c4e094b75d719977bf5b ...
$ echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
This job deploys something from the main branch.
Cleaning up project directory and file based variables                      00:00
Job succeeded
```