INTEGRATION CONTINUE

Cafés calcul

Baptiste MARY

18 Février 2021

Introduction

Définition

Définition vague

Application d'une chaine d'opérations à chaque modification d'un code source.

Définition wikipedia

Ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée.

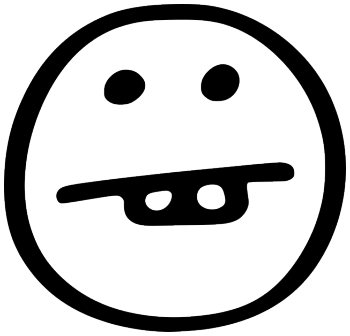
Définition (trop) concrète

Exécution, par un runner, d'un pipeline de test unitaires lors d'un push sur une forge logicielle.

Plan

- Contexte
- Fonctionnement
- Runner
- Déploiement Continue
- Démonstration

Audience

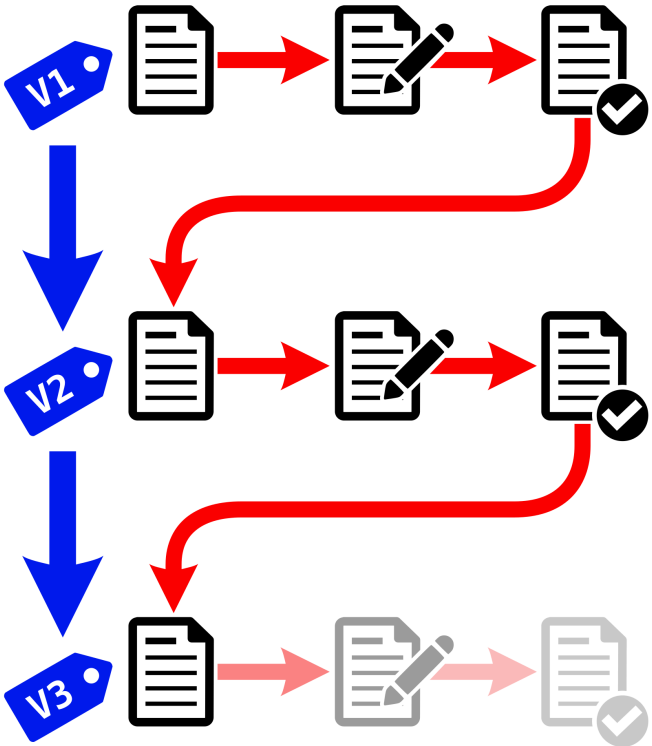


Contexte

Versioning

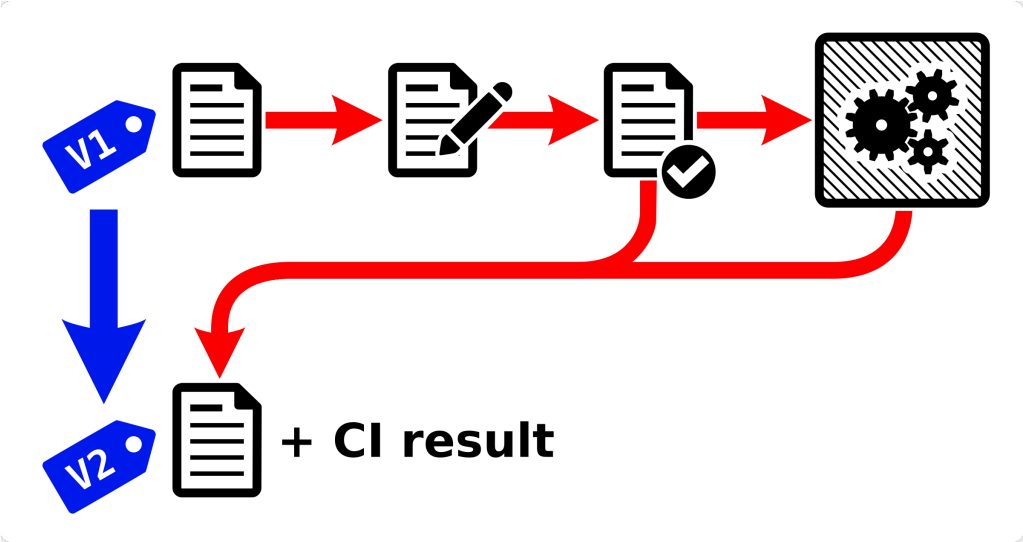
Étapes fondamentales

Étapes de versioning



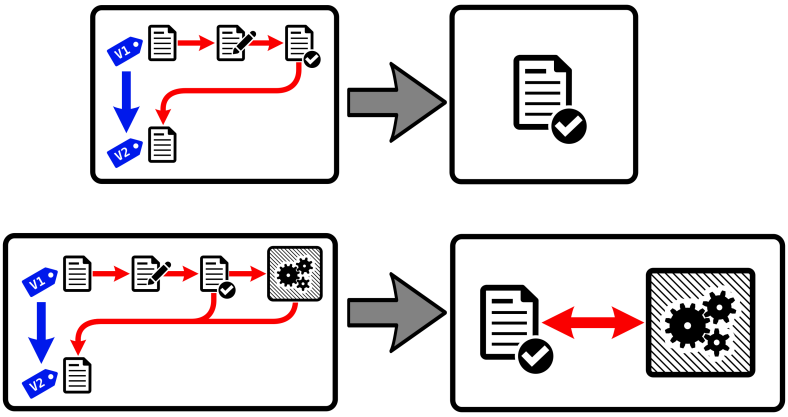
Intégration continue

Étape supplémentaire de versioning

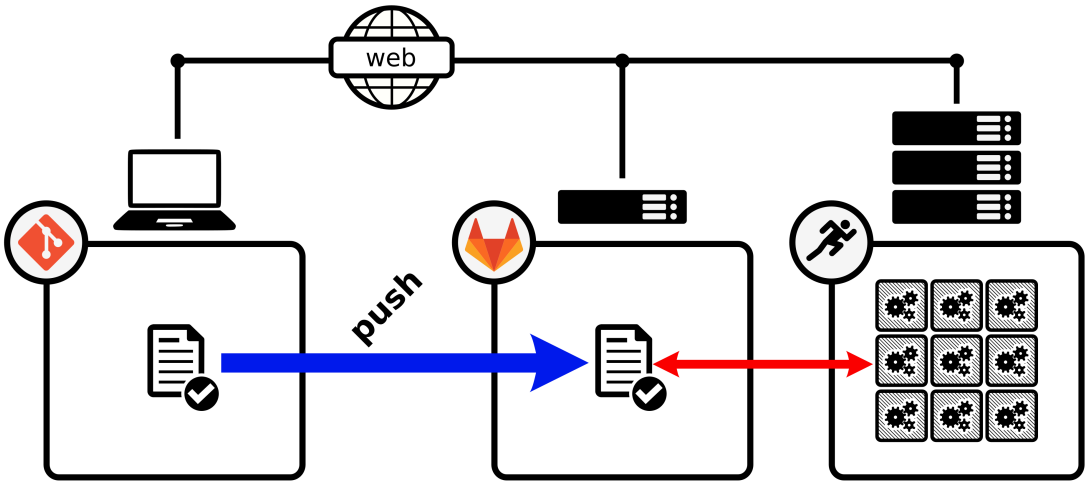


Runners

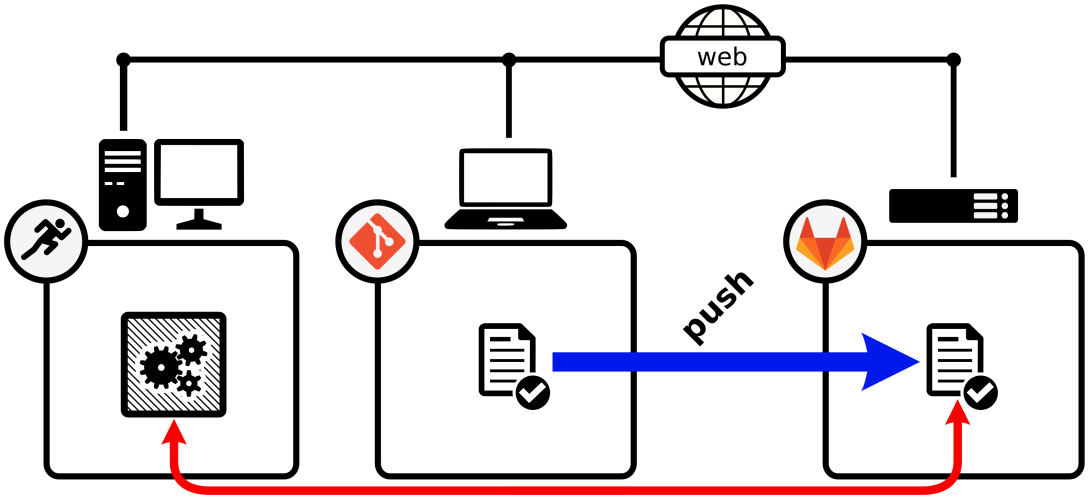
Convention graphique



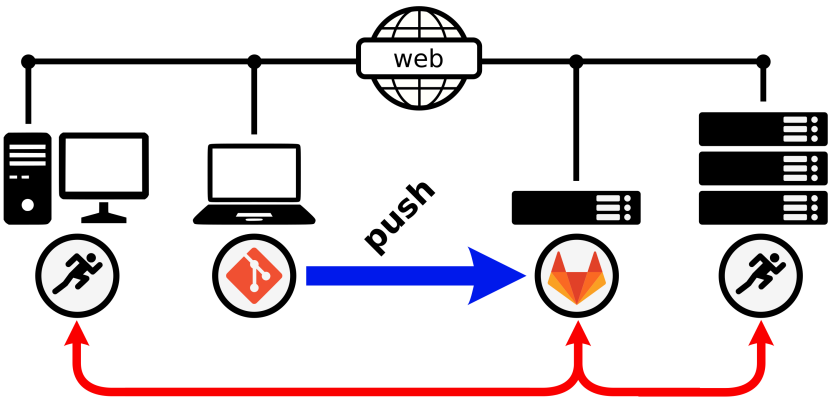
Différents acteurs



Runner local



Bilan



Un peu de recul

Quels besoins ?

Tout ça peut être réalisé à la main :

- versionner : copier coller + renommage + archivage ;
- build manuel ;
- exécution manuelle de la chaine de test.

You are CI friendly !

Automatiser

- Logiciel de versionning (git)
- Build et chaine de tests
  - script bash
  - make
  - git alias

You do CI !

Avantages :

- évacuer les efforts redondants,
- assurance de qualité.

Centraliser

Centraliser les étapes communes

- serveur de forge pour centraliser les contributions (gitlab)
- serveur exécutant le build et les tests (runner gitlab).

You do good CI !

Effets secondaires positifs :

- process normalisé facilitant le travail d'équipe
- ne bloque pas votre machine pendant les build+tests
- utiliser plusieurs runners pour accélérer les build+tests
- force à tester son code sur une autre machine que la sienne

Retour sur les définitions

Définition (trop) concrète

> Exécution, par un runner, d'un pipeline de test unitaires lors d'un push sur une forge logicielle.

Définition wikipedia

> Ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée.

Définition vague

> Application d'une chaine d'opérations à chaque modification d'un code source.

Fonctionnement

Activer la CI

Étapes d'activation

> Starting from version 8.0, GitLab Continuous Integration (CI) is fully integrated into GitLab itself and is enabled by default on all projects.

Dernière version (11 fev. 2021) : 13.8.4

- Être *Maintainer* ou *Owner* du projet
- Configurer le(s) runner(s) qui sera associé au projet
- Ajouter un fichier `.gitlab-ci.yml` dans la racine du projet

`gitlab-ci.yml`

https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html

Recette

Le fichier `.gitlab-ci.yml` contient une liste d'actions

```
test-code-job1:
  script:
    - echo "Test some files with one command:"
    - make test1

test-code-job2:
  script:
    - echo "Test other files with a different command:"
    - make test2
```

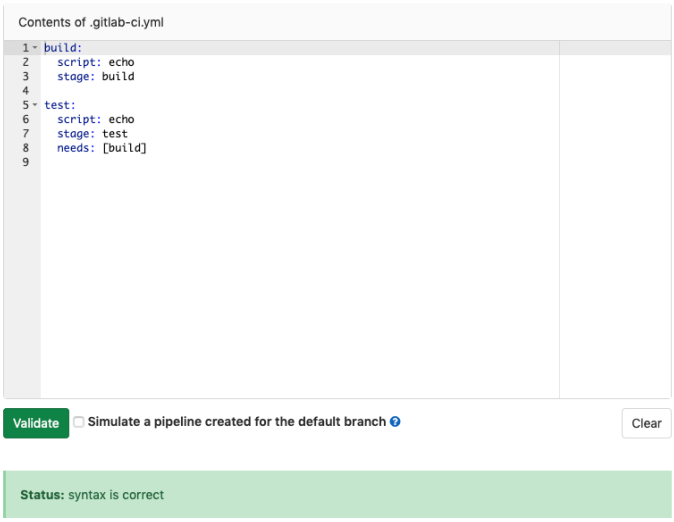
Mots clefs

```
after_script          image               script
allow_failure         include             services
artifactsartifacts    interruptible       stage
before_script         only                tags
cache                 pages               timeout
```

```
coverage                parallel              trigger
dependencies            release               variables
environment             resource_group        when
except                  retry
extends                 rules
```

CI Lint tool

Valider la syntaxe de son `.gitlab-ci.yml` avec le *CI Lint tool*

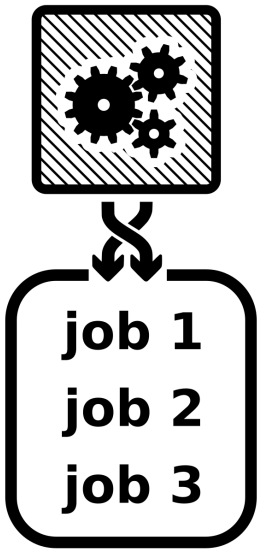


Depuis votre projet :

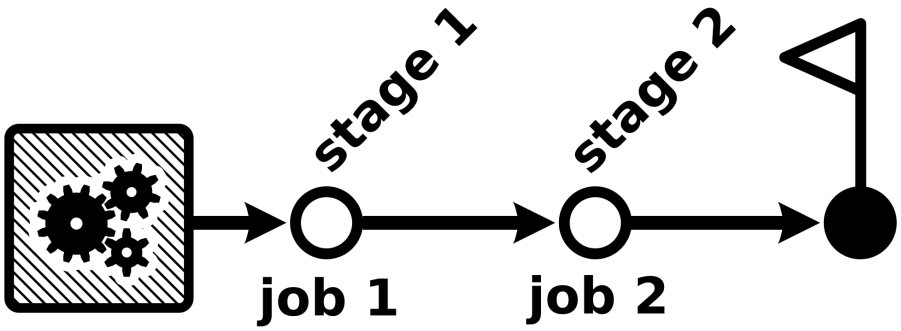
- `CI/CD > Pipelines` ou `CI/CD > Jobs`
- Clicker sur `CI lint`

Pipeline

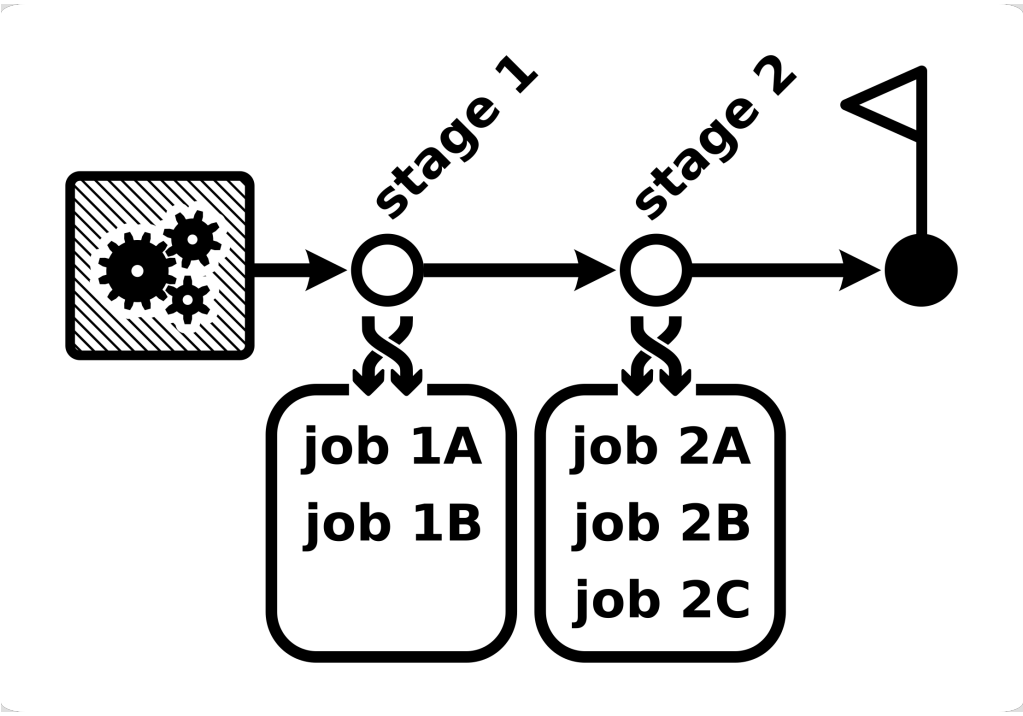
Jobs concurrents



Stages



Stages & jobs concurrents

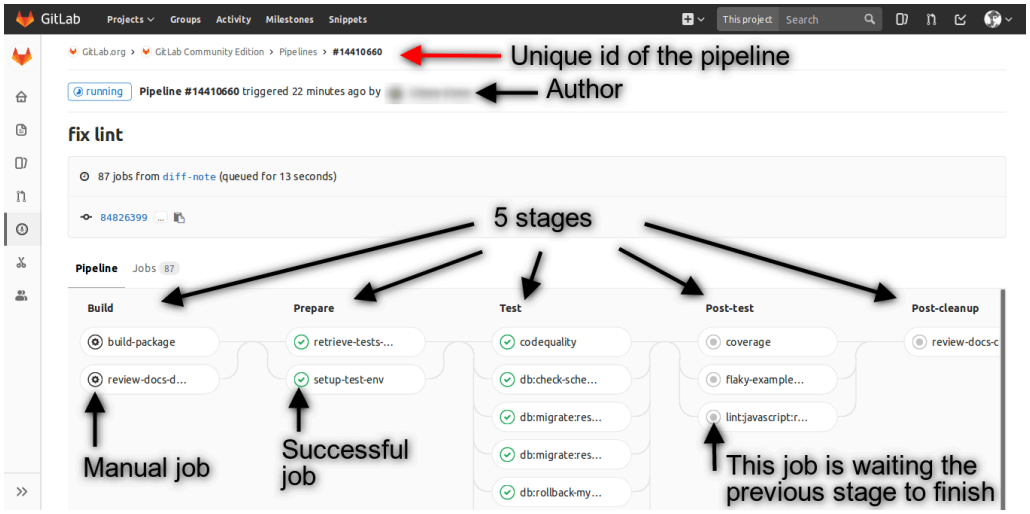


Exemple yaml

```yaml
stages:
  - build
  - test

build-code-job:
  stage: build
  script:
    - echo "Build the project files:"
    - make

test-code-job1:
  stage: test
  script:
    - echo "If the files are built successfully, run test suite 1:"
    - make test1

test-code-job2:
  stage: test
  script:
    - echo "If the files are built successfully, run test suite 2:"
    - make test2
```

Résumé

- Un pipeline peut être composé d'un ou de plusieurs jobs.
- Les jobs sont exécutés de façon concurrente.
- Les jobs peuvent être organisés en étapes (stage), exécutées séquentiellement.



Runners

Installation

Nature du Runner

Un Runner peut être

- une machine virtuelle,
- un VPS (Virtual Private Server),
- une machine *bare-metal*,
- un container docker,
- un cluster de containeurs (Kubernetes).

Prérequis :

- un accès réseau au server gitlab (API)

Support officiel

Architectures :

```
x86, AMD64, ARM64, ARM, s390x
```

Systèmes d'exploitation :

```
Linux (rpm, deb), Windows, macOS, FreeBSD
```

Containeurs

```
Docker, Kubernetes, OpenShift
```

Exemple sur Debian

Ajouter le dépot officiel de *GitLab* :

```
curl -L "https://packages.gitlab.com/install/repositories/runner/
gitlab-runner/script.deb.sh" | sudo bash
```

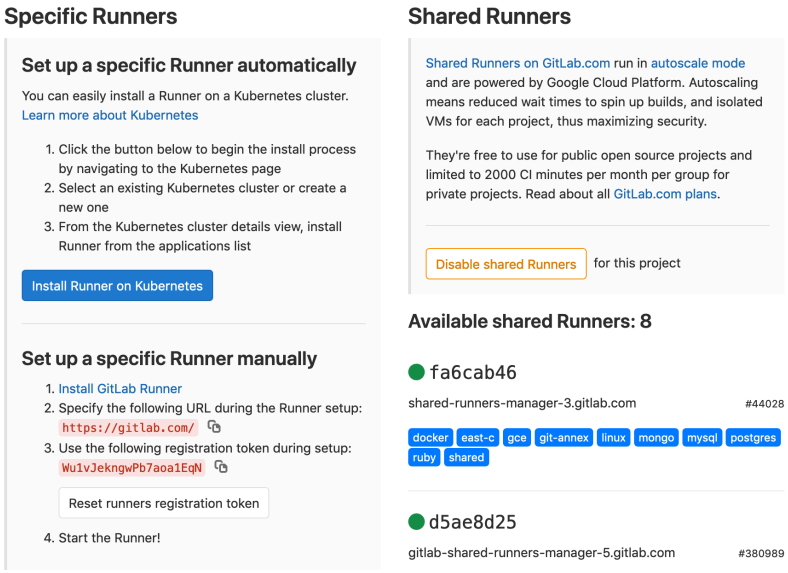
Installer la dernière version de *GitLab Runner*,

```
export GITLAB_RUNNER_DISABLE_SKEL=true; sudo -E apt-get install
```

```
gitlab-runner
```

Connection à Gitlab

Obtenir un *token*



- Runner partagé : depuis l'interface administrateur, clicker sur `Overview > Runners`
- Runner privé : depuis le projet entrer dans `Settings > CI/CD`, section `Runners`

Enregistrement

Depuis le runner …

Linux

```
sudo gitlab-runner register
```

Windows

```
./gitlab-runner.exe register
```

Docker

```
docker run --rm -it -v /srv/gitlab-runner/config:/etc/gitlab-
runner gitlab/gitlab-runner register
```

Étapes d'enregistrement

1. URL de l'instance *GitLab*.
2. Token fournit par l'instance *GitLab*.
3. Description du runner.
4. Tags associés au runner, separés par des virgules.
5. Précisier l'exécuteur du runner (shell, docker, …).
    - Image docker par défaut.

One liner

```
sudo gitlab-runner register \
  --non-interactive \
  --url "https://gitlab.com/" \
  --registration-token "PROJECT_REGISTRATION_TOKEN" \
  --executor "docker" \
  --docker-image alpine:latest \
  --description "docker-runner" \
  --tag-list "docker,aws" \
  --run-untagged="true" \
  --locked="false" \
  --access-level="not_protected"
```

Tags

Usage

- Spécifier un environnement pré-installé (python, ruby, postgress, docker …).
- Permet la réservation pour un usage spécifique.
- La réservation se fait dans le `.gitlab-ci.yml` avec le mot clef `tags`

```
compile_job:
    stage: build
    script:
        - echo Working...
    tags: [llvm]
    only: [master]
```

Ne pas confondre

La notion de tag pour *git* et pour *gitlab* :

- tag git : référence vers un commit
- tag gitlab : référence vers un runner

Il est possible de se référer à un tag *git* dans *gitlab-ci* :

```
compile_job:
    stage: build
    script:
        - echo Working...
    tags: [llvm]
    only: [tags]
```

Docker

Spécifier un image

Utiliser une image disponible sur *DockerHub* :

```
image: alpine:latest

stages: [build, test]

compile_job:
    stage: build
    script: [make]

test_job:
    stage: test
    script: [make test]
```

Utiliser une image locale

- Configurer le runner pour l'autoriser à utiliser une image locale (`gitlab-runner/config.toml`)
- Construire l'image depuis le runner (admin)

```
image: my-docker-image

stages: [build, test]

compile_job:
    stage: build
    script: [make]

test_job:
    stage: test
    script: [make test]
```

Construire une image

Construire l'image et l'utiliser dans le pipeline :

```
image: docker:latest

variables:
  TEST_IMAGE: my-custom-image

before_script:
  - docker build --no-cache -t $TEST_IMAGE .
  - docker run $TEST_IMAGE make

test_job_1:
  script:
    - docker run $TEST_IMAGE make test1

test_job_2:
  script:
    - docker run $TEST_IMAGE make test2
```

Documentation

https://docs.gitlab.com/runner/

Déploiement continue

Artifacts

Description

- Résultat d'un job dans un pipeline.
- Chaque job du pipeline peut renvoyer un ou plusieurs artifacts.
- Peuvent être sotckés par gitlab et peuvent être téléchargés par les utilisateurs.

```
test:
 script: ["echo 'test' > file.txt"]
 artifacts:
   paths: ['file.txt']
```

Mots clefs associés :

```
exclude       public
expire_in     reports
expose_as     untracked
name          when
paths
```

Environments

Description

- Environnement logiciel créé par un job
- L'environnement d'exécution du job est copié sur l'environnement
- Peut être accédé par un utilisateur via URL

```
deploy_master:
  stage: deploy
  script: make deploy
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com/
  only:
    - master
```

Mots clefs associés :

```
action
auto_stop_in
name
on_stop
url
```

Démonstration