

Maitriser la gestion de cycle de vie des développements applicatifs avec GitLab CI/CD

Brahim Hamdi
Consultant Devops & Cloud
brahim.hamdi.consult@gmail.com

Formateur

Brahim Hamdi

- Consultant/formateur
- Expert DevOps & Cloud



Objectifs pédagogiques

- Connaître l'offre GitLab
- Mettre en place l'intégration continue (CI) et le déploiement continu (CD) avec GitLab
- Appréhender les éléments constitutifs d'une usine logicielle DevOps

Public concerné

- Développeurs,
- Chefs de projet,
- Administrateurs systèmes,
- Architectes

Prérequis

- Connaissances de base du système Linux.
- Connaissances de base de la gestion de versions avec Git.

Le plan de formation

- Présentation de GitLab
- GitLab CI/CD
- Plus loin dans l'utilisation de GitLab

Présentation de GitLab

Qu'est ce que GitLab ?

- Outil open source de gestion de projets **git** (licence MIT)
- Application Web développée en langage Ruby par GitLab Inc
 - Dépôt : gitlab.com/gitlab-org/gitlab
- Dernière version : 17.0 (16 Mai 2024)
- Les principales fonctionnalités :
 - Gérer le cycle de vie de projets Git
 - Gérer les participants aux projets et leurs droits (rôles, groupes, etc ...)
 - Déposer des Issues pour lister les bugs
 - Gérer la communication entre ces participants
 - Proposer des Merges Requests pour fusionner les branches
 - Lancer des pipelines d'intégration et de déploiement continus via GitLab CI/CD
 - Fournir un Wiki pour la documentation

L'offre GitLab



- GitLab SaaS (<https://gitlab.com/>) : l'offre de logiciel en tant que service de GitLab. Vous n'avez rien besoin d'installer pour utiliser GitLab SaaS, il vous suffit de vous inscrire et de commencer à utiliser GitLab immédiatement.
- GitLab Dedicated : un service SaaS à locataire unique pour les grandes entreprises hautement réglementées.
- GitLab autogéré : installez, administrez et gérez votre propre instance GitLab.

Les différentes distributions (autogéré)

- GitLab Community Edition (CE) : version libre
- GitLab Enterprise Edition (EE) : version payante

Feature	CE	EE
file manager, issues, wiki	✓	✓
Online code changes	✓	✓
GitHub import	✓	✓
LDAP/AD authentication	✓	✓
CI and Docker support	✓	✓
Support	×	✓
Kerberos authentication	×	✓
Merge Request Approvals	×	✓
Issues/Merge Requests templates	×	✓

GitLab vs GitHub

 GitHub	 GitLab
Les issues peuvent être suivies dans plusieurs repositories	Les issues ne peuvent pas être suivies dans plusieurs repositories
Repositories privés payants	Repositories privés gratuits
Pas d'hébergement gratuit sur un serveur privé	Hébergement gratuit possible sur un serveur privé
Intégration continue uniquement avec des outils tiers (Travis CI, CircleCI, etc.)	Intégration continue gratuite incluse
Aucune plateforme de déploiement intégrée	Déploiement logiciel avec Kubernetes
Suivi détaillé des commentaires	Pas de suivi des commentaires
Impossible d'exporter les issues au format CSV	Exportation possible des issues au format CSV par e-mail
Tableau de bord personnel pour suivre les issues et pull requests	Tableau de bord analytique pour planifier et surveiller le projet

Inscription sur gitlab.com

- Créer un compte GitLab et authentifiez-vous avec votre login.

The screenshot shows the GitLab website homepage. At the top, there's a navigation bar with links: Why GitLab, Platform, Solutions, Pricing, Resources, Company, and Contact us. A search icon and a 'Get free trial' button are also present. Below the navigation bar, there's a Gartner badge stating 'GitLab named a Leader in the Gartner® Magic Quadrant™ for DevOps Platforms' with a link to 'Read the report'. The main hero section features the text 'Software. Faster.' and 'GitLab is the most comprehensive AI-powered DevSecOps Platform.' with a 'Get free trial' button. To the right, there are three overlapping cards: a 'Vulnerability Report' card showing counts for Critical (49), High (35), Medium (56), and Low (2) vulnerabilities; an 'Approve' card with a 'Requires 2 approvals' status; and a 'Python' card showing a code snippet for translating languages.

Software. Faster.

GitLab is the most comprehensive AI-powered DevSecOps Platform.

[Get free trial](#)

Vulnerability Report

Severity	Count
Critical	49
High	35
Medium	56
Low	2

Approve Requires 2 approvals

Python

```
Translating  
languages =  
'en': {'He'  
'hello': 'He'  
'goodbye': '  
'es': {'
```

Créer un nouveau projet

- Cliquez sur **Create a projet** puis sur **Create blank Projectct.**



- Indiquez un **Project Name** : **Car assembly line**
- Cochez qu'il s'agit d'un projet **Private**, puis validez en cliquant sur **Create Project**.

The screenshot shows the 'Create new project' form at https://gitlab.com/projects/new#blank_project. The form has the following fields and options:

- Project name**:
- Project URL**:
- Project slug**:
- Project deployment target (optional)**:
- Visibility Level**: ☒ Private, ☐ Public
- Project Configuration**: ☒ Initialize repository with a README, ☐ Enable Static Application Security Testing (SAST)

The 'Create project' button is at the bottom right of the form.

Page d'accueil du projet

Accès rapide aux merges requests, issues et tasks

Accès à la page d'accueil (projets, groupes, ...)

Gestion des paramètres du compte

Infos générales du projet

La branche en cours de consultation

Dernier commit sur la branche sélectionnée

Clone du dépôt

Accès à tous les outils du projet

Liste des fichiers de la branche sélectionnée

The screenshot shows the GitLab project page for 'Car assembly line'. The left sidebar contains navigation links: Project, Pinned, Issues, Merge requests, Manage, Plan, Code, Build, Secure, Deploy, Operate, and Monitor. The main content area displays project information: 'Car assembly line' (Project ID: 52099193), 1 Commit, 1 Branch, 0 Tags, and 3 KiB Project Storage. Below this, the 'main' branch is selected, showing the 'Initial commit' by 'leadingit consult' just now. A table lists the files in the selected branch: README.md. The 'Clone' button is highlighted in the top right corner.

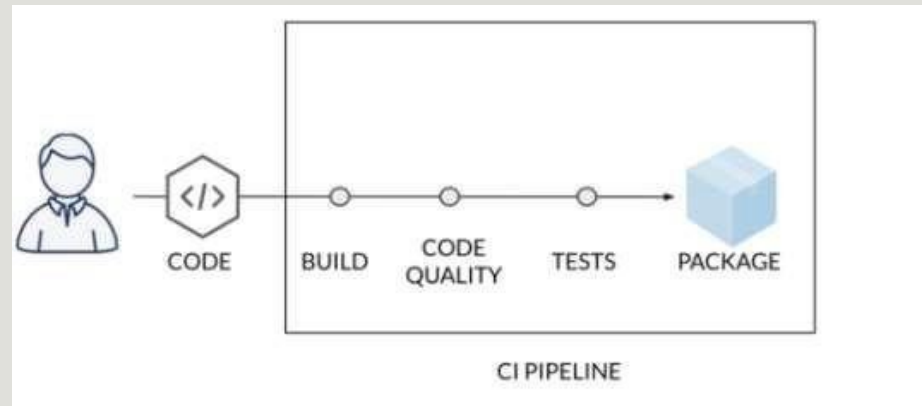
Name	Last commit	Last update
README.md	Initial commit	just now

GitLab CI/CD

Continuous Integration CI

■ Intégration continue (CI = Continuous Integration)

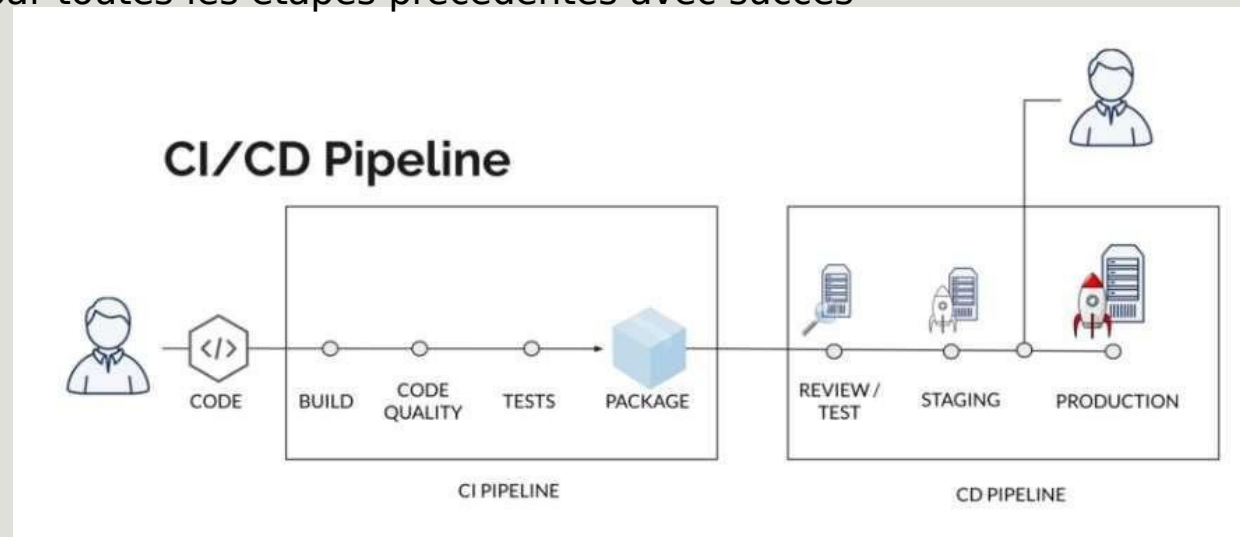
- ✓ CI est une pratique qui permet d'intégrer le code avec d'autres développeurs
- ✓ Le plus souvent, l'intégration du code est de vérifier si l'étape de construction (build stage) est toujours fonctionnelle
- ✓ Une pratique courante consiste à vérifier également si les tests unitaires (Unit Testing stage) sont toujours fonctionnels
- ✓ L'objectif du pipeline CI est de construire un **package** qui sera déployé par la suite



Continuous Delivery CD

■ Livraison continue (CD = Continuous Delivery)

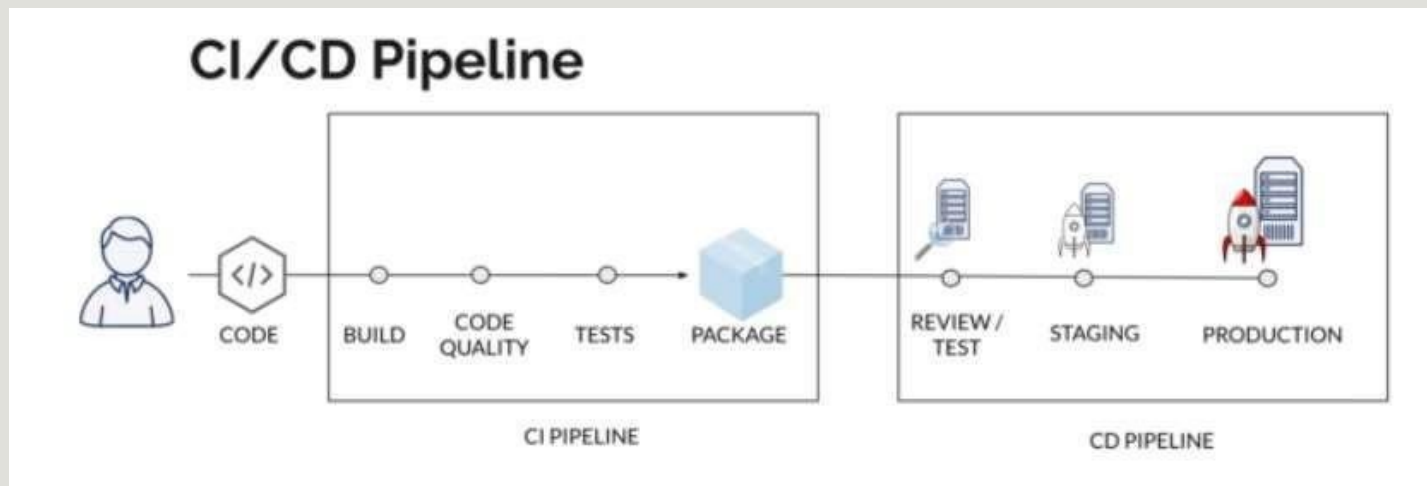
- ✓ C'est une extension de l'intégration continue
- ✓ L'objectif du CD est de prendre le package qui a été créé par le pipeline de CI et de tester son déploiement dans un **environnement de test (REVIEW et STAGING)**
- ✓ En ajoutant cette étape de pré-production et en exécutant certains tests, cela nous permet d'exécuter différents types de tests qui nécessitent la réponse de l'ensemble du système (généralement appelé tests d'acceptation)
- ✓ L'étape de déploiement en production (STAGE DEPLOY) est lancée **manuellement** si et seulement si le package a passé par toutes les étapes précédentes avec succès



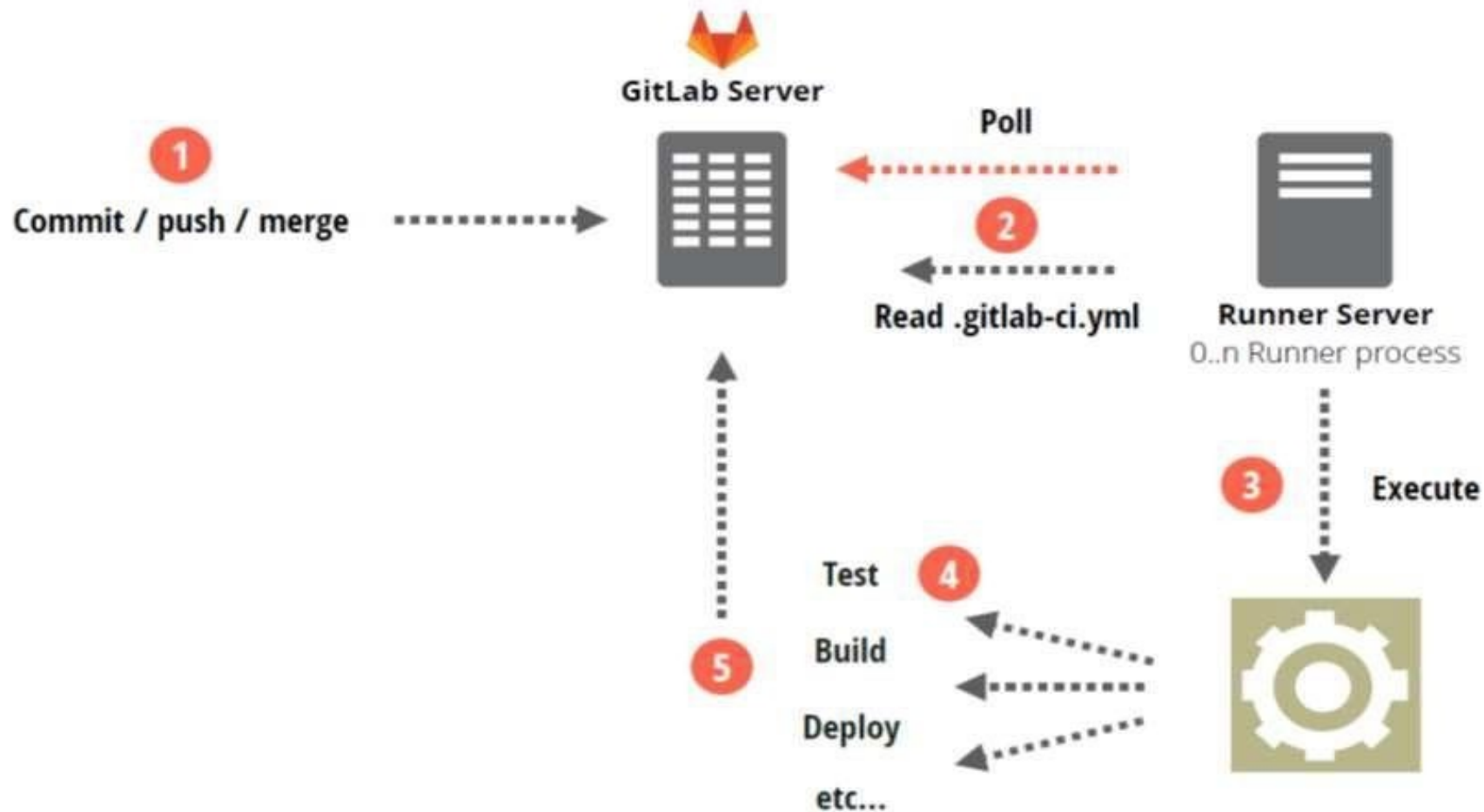
Continuous Deployment CD

- **Déploiement continu (CD = Continuous Deployment)**

- Le déploiement continu est la pratique d'automatisation complète de l'ensemble des processus du pipeline CI/CD dans un environnement de **production**.
- Le package doit d'abord passer par toutes les étapes précédentes avec succès
- Aucune intervention manuelle n'est requise: **c'est automatique.**

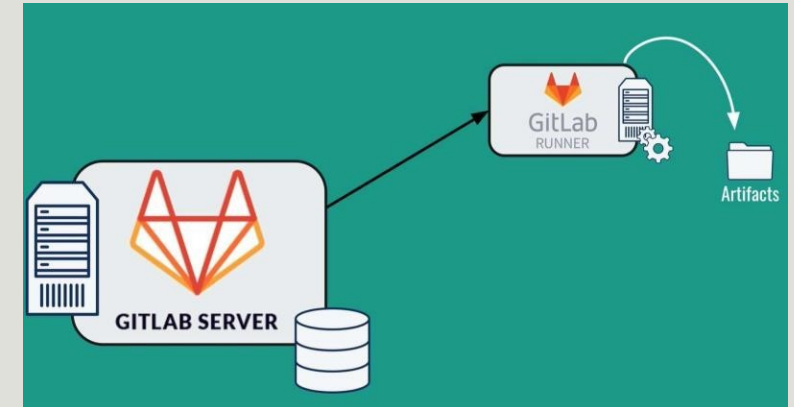
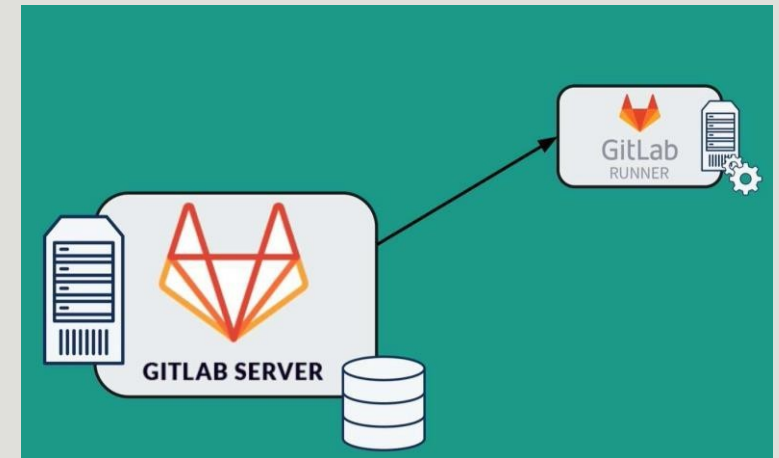


Gitlab CI/CD : Fonctionnement



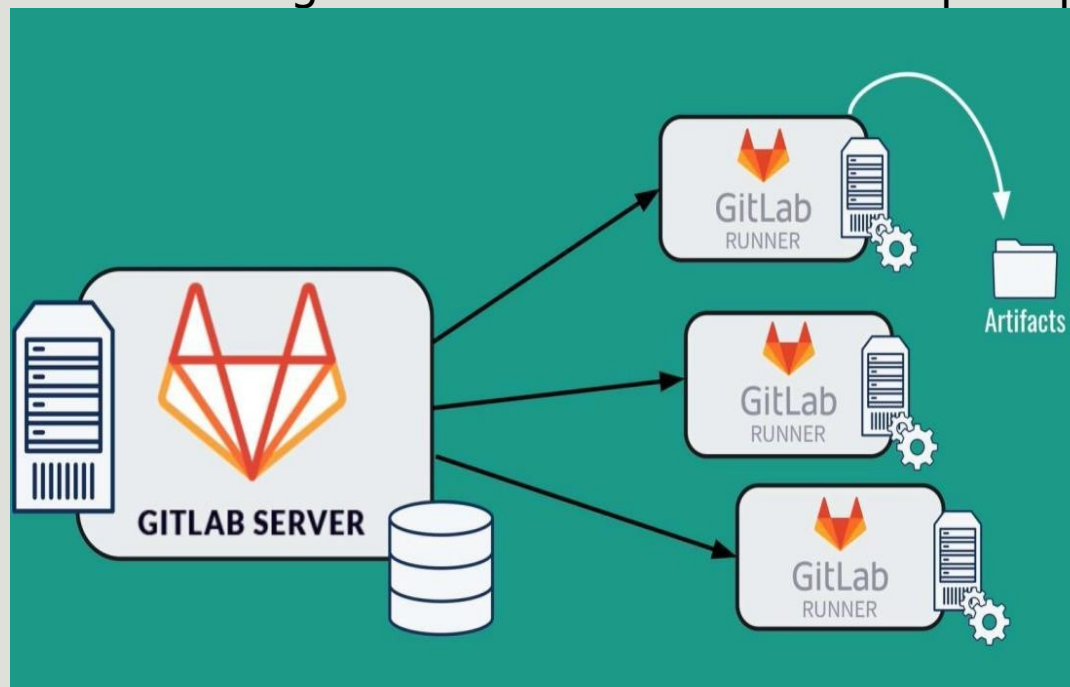
Gitlab CI/CD : Serveur + Runner

- Vous avez besoin d'au moins d'un **serveur Gitlab** pour installer l'interface Web, **vos dépôts Git** (référentiels) et un **Runner**.
- Pour que le serveur Gitlab n'exécute pas les travaux (Jobs) et pour avoir une architecture **évolutive**, facile à déployer et **scalable**, l'exécution de ces Jobs sera déléguée au **Runner**
- Si le Job a été exécuté avec succès, nous pouvons sauvegarder les résultats (les fichiers et/ou dossiers) dans des **Artifacts**. Ces derniers vont être stockés au sein des pipelines pour être utilisés par d'autres Jobs.



Gitlab Runner

- Un Runner Gitlab-CI est un simple démon qui attend les Jobs comme vus dans le diagramme précédent.
- Dans Gitlab, vous pouvez créer vos propres Runners sur votre propre infrastructure en fonction de vos besoins.
- Selon le nombre de projets que vous avez ou de l'activité que vous avez sur un projet, il vous faudra davantage des Runners ou beaucoup de patience.

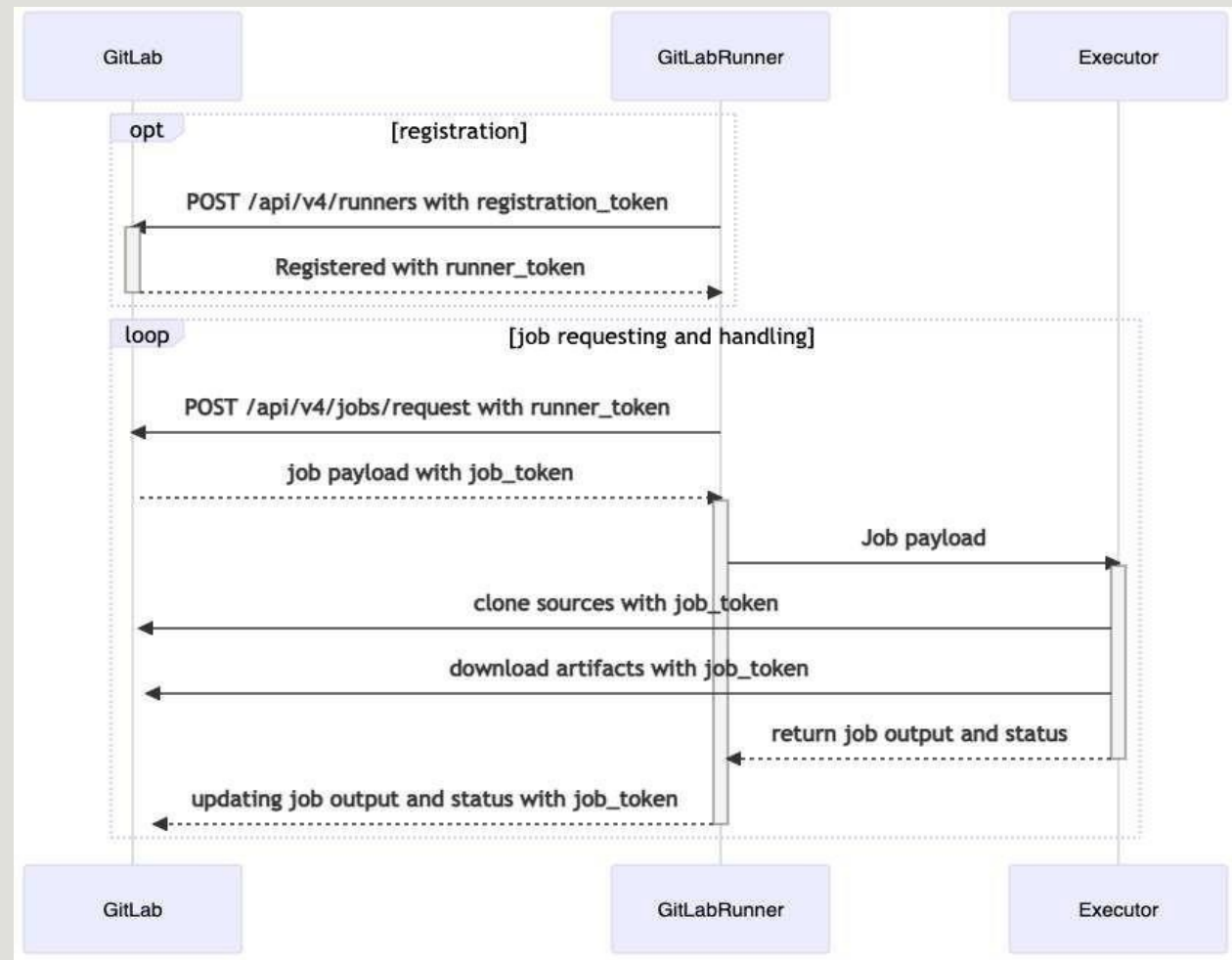


Gitlab Runner : Executor

- Une fois un Job reçu celui-ci va demander à « un exécuteur » de traiter la demande.
- Les **exécuteurs** sont des sous-processus qui vont se charger de faire les commandes (scripts) que vous avez définies dans votre gitlab-ci.
- Gitlab-CI est capable de fonctionner de différente manière :
 - ✓ SSH,
 - ✓ Shell,
 - ✓ Parallels,
 - ✓ VirtualBox,
 - ✓ Docker,
 - ✓ Docker Machine (auto-scaling),
 - ✓ Kubernetes
 - ✓ Personnalisé (Custom)

Gitlab Runner : Executor

- L'avantage est double :
 - ✓ Pas de limite en nombre de compilation.
 - ✓ Accès à vos ressources locales pour le déploiement.



Gitlab Runner : Executor

Executor	SSH	Shell	VirtualBox	Parallels	Docker	Kubernetes	Custom
Nettoyer l'environnement de build (pour chaque build)	✗	✗	✓	✓	✓	✓	conditional
Réutiliser le clone précédent s'il existe	✓	✓	✗	✗	✓	✗	conditional
Protéger l'accès au système de fichiers du Runner	✓	✗	✓	✓	✓	✓	conditional
Migrer la machine de Runner	✗	✗	partial	partial	✓	✓	✓
Prise en charge de zéro-configuration pour les builds simultanés	✗	✗	✓	✓	✓	✓	conditional
Environnements de construction très compliqués	✗	✗	✓	✓	✓	✓	✓
Débogage des problèmes de build	easy	easy	hard	hard	medium	medium	medium

<https://docs.gitlab.com/runner/executors/>

Comment choisir un exécuteur ?

- **Shell**
 - C'est le plus simple de tous.
 - Vos scripts seront lancés sur la machine qui possède le Runner.
- **Parallels, VirtualBox**
 - Le Runner va créer (ou utiliser) une machine virtuelle pour exécuter les scripts.
 - Pratique pour avoir un environnement spécifique (exemple macOS)
- **Docker**
 - Utilise Docker pour créer / exécuter vos scripts et traitement (en fonction de la configuration de votre .gitlab-ci.yml)
 - Solution la plus simple et à privilégier
- **Docker Machine (auto-scaling)**
 - Identique à docker, mais dans un environnement Docker multi-machine avec auto-scaling.
- **Kubernetes**
 - Lance vos builds dans un cluster Kubernetes.
 - Très similaire à Docker-Machine
- **SSH**
 - À ne pas utiliser. Il existe, car il permet à Gitlab-CI de gérer l'ensemble des configurations possibles.

Pipeline et YAML

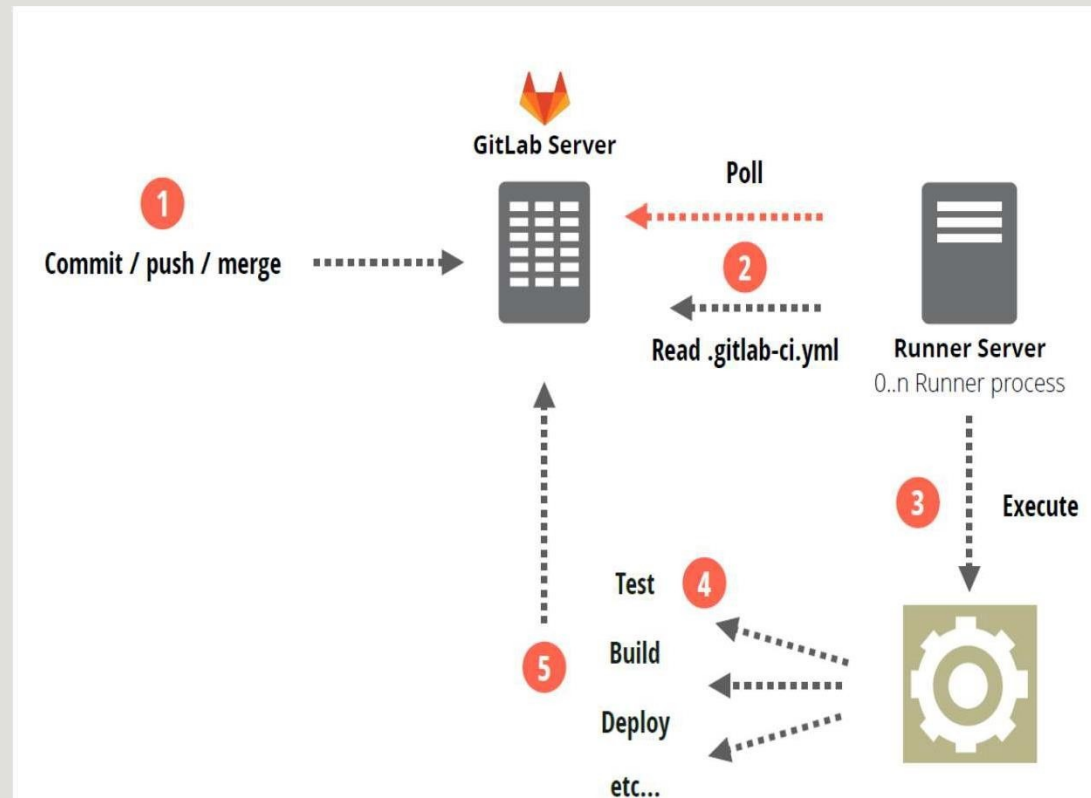
- La gestion de la pipeline CD/CD dans GitLab se fait simplement par l'ajout d'un fichier **YAML** «**.gitlab-ci.yml**» dans la racine de votre projet git concerné .
- **YAML** est un langage de sérialisation de données utilisé, par exemple, pour le déploiement de configurations par **Ansible** ou pour la configuration d'applications multi-containers via **Docker Compose**.

```
1  ---
2  #Blog about YAML
3
4  title: YAML Ain't Markup Language
5  author:
6    first_name: Lauren
7    last_name: Malhoit
8    twitter: "@Malhoit"
9  learn:
10    - Basic Data Structures
11    - Commenting
12    - When and How
```

- Pour plus de détails sur YAML:
 - <https://opensharing.fr/yaml-memo-bases>
 - https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html

Le manifeste .gitlab-ci.yml

- Pour que la CI/CD sur GitLab fonctionne il vous faut un **manifeste .gitlab-ci.yml** à la racine de votre projet
- Dans ce manifeste vous allez pouvoir définir des **stages**, des **jobs**, des **variables**, etc.
- Le pipeline est déclenché à chaque **commit** ou **push** et s'exécute dans le Runner
- Et .gitlab-ci.yml explique au(x) Runner(s) ce qu'il faut faire



LES JOBS

- Dans le manifeste de GitLab CI/CD vous pouvez définir un nombre **illimité** de jobs, avec des contraintes indiquant s'ils doivent être exécutés ou non.
- Voici comment déclarer un job le plus simplement possible :

```
job1:  
  script: echo 'my first job'  
  
job2:  
  script: echo 'my second job'
```

- Les noms des jobs doivent être uniques et ne doivent pas faire parti des mots réservés : *image, services, stages, types, before_script, after_script, variables, cache...*
- Dans la définition d'un job seule la déclaration **script** est obligatoire.

SCRIPT

- La déclaration script est donc la seule obligatoire dans un job. Cette déclaration est le cœur du job car c'est ici que vous indiquerez les actions à effectuer.
- Il peut appeler un ou plusieurs script(s) de votre projet, voire exécuter une ou plusieurs ligne(s) de commande.

```
job-script:  
  script: ./bin/script/my-script.sh ## Appel d'un script de votre projet  
  
job-scripts:  
  script: ## Appel de deux scripts de votre projet  
    - ./bin/script/my-script-1.sh  
    - ./bin/script/my-script-2.sh  
  
job-command:  
  script: printenv # Exécution d'une commande  
  
job-commands:  
  script: # Exécution de deux commandes  
    - printenv  
    - echo $USER'
```

BEFORE_SCRIPT ET AFTER_SCRIPT

- Ces déclarations permettront d'exécuter des actions avant et après votre script principal.
- Ceci peut être intéressant pour bien diviser les actions à faire lors des jobs, ou bien appeler ou exécuter une action avant et après chaque job.

```
before_script: # Exécution d'une commande avant chaque `job`  
- echo 'start jobs'
```

```
after_script: # Exécution d'une commande après chaque `job`  
- echo 'end jobs'
```

```
job-no_overwrite:
```

```
# Ici le job exécutera les actions du `before_script` et `after_script` par défaut
```

```
script:  
- echo 'script'
```

```
job-overwrite-before_script:
```

```
before_script:  
- echo 'overwrite' # N'exécutera pas l'action de `before_script` par défaut
```

```
script:  
- echo 'script'
```

```
job-overwrite-after_script:
```

```
script:  
- echo 'script'
```

```
after_script:  
- echo 'overwrite' # N'exécutera pas l'action définie dans le `after_script` par défaut
```

IMAGE

- Cette déclaration est simplement l'image docker qui sera utilisée lors d'un job ou lors de tous les jobs.

image: alpine # Image utilisée par tous les `jobs`, ce sera l'image par défaut

node: # Job utilisant l'image node

image: node

script: yarn install

alpine: # Job utilisant l'image par défaut

script: echo \$USER

STAGES

- Cette déclaration permet de **grouper** des jobs en **étapes**.
- Par exemple on peut faire une étape de build, de test, de deployment,
- Si vous n'avez pas défini à quel stage un job appartient, il sera automatiquement attribué au stage **Test**.



stages: # Ici on déclare toutes nos étapes

- build
- test
- deploy

build:

stage: build # Ce `job` fait partie de l'étape build
script: make build

test-functional:

stage: test # Ce `job` fait partie de l'étape test
script: make test-functional

test-unit:

stage: test # Ce `job` fait partie de l'étape test
script: make test-unit

deploy:

stage: deploy # Ce `job` fait partie de l'étape deploy
script: make deploy

ONLY ET EXCEPT

- Ces deux directives permettent de mettre en place des contraintes sur l'exécution d'une tâche.
- Vous pouvez dire qu'une tâche s'exécutera uniquement sur l'événement d'un push sur master ou s'exécutera sur chaque push d'une branche sauf master.

only-master:

script: make deploy

only:

- master # Le job sera effectué uniquement lors d'un événement sur la branche master

except-master:

script: make test

except:master:

- master # Le job sera effectué sur toutes les branches lors d'un événement sauf sur la branche master

ONLY avec schedules

- Pour l'utilisation de **schedules** il faut dans un premier temps définir des règles dans l'interface web.
- On peut les configurer dans l'interface web de Gitlab : **CI/CD -> Schedules** et remplir le formulaire.

Schedule a new pipeline

Description

Test schedule

Interval Pattern

☒ Custom (Cron syntax) ☐ Every day (at 4:00am) ☐ Every week (Sundays at 4:00am) ☐ Every month (on the 1st at 4:00am)

0 20 * * *

Cron Timezone

Paris

Target Branch

master

Variables

RELEASE staging

Input variable key Input variable value

Activated

☒ Active

WHEN

- Comme pour les directives `only` et `except`, la directive `when` est une contrainte sur l'exécution de la tâche. Il y a quatre modes possibles :

- **on_success** : le job sera exécuté uniquement si tous les jobs du stage précédent sont passés
- **on_failure** : le job sera exécuté uniquement si un job est en échec
- **always** : le job s'exécutera quoi qu'il se passe (même en cas d'échec)
- **manual** : le job s'exécutera uniquement par une action manuelle

stages:

- build
- test
- report

build:

stage: build
script:
- make build

test:

stage: test
script:
- make test

when: on_success #s'exécutera uniquement si le job `job:build` pass

report:

stage: report
script:
- make report

when: on_failure #s'exécutera si le job `job:build` ou `job:test` ne pa

ALLOW_FAILURE

- Cette directive permet d'accepter qu'un job échoue sans faire échouer la pipeline.

```
stages:  
  - build  
  - test  
  - report  
  - clean  
  
...  
  
stage: clean  
  script:  
    - make clean  
  when: always  
  allow_failure: true # Ne fera pas échouer la pipeline  
  
...
```

TAGS

- Avec GitLab Runner vous pouvez héberger vos propres runners sur un serveur ce qui peut être utile dans le cas de configuration spécifique.
- Chaque runner que vous définissez sur votre serveur à un nom, si vous mettez le nom du runner en tags, alors ce runner sera exécuté.

```
tag:  
script: yarn install  
tags:  
  - shell # Le runner ayant le nom `shell` sera lancé
```

SERVICES

- Cette déclaration permet d'ajouter des services (container docker) de base pour vous aider dans vos jobs.
- Par exemple si vous voulez utiliser une base de données pour tester votre application c'est dans services que vous le demanderez.

```
test-functional:  
  image: registry.gitlab.com/username/project/php:test  
  services:  
    - postgres # On appelle le service `postgres` comme base de données  
  before_script:  
    - composer install -n  
  script:  
    - codecept run functional
```

ENVIRONMENT

- Cette déclaration permet de définir un environnement spécifique au déploiement
- Il est possible de spécifier :
 - ✓ un **name**,
 - ✓ une **url**,
 - ✓ une condition **on_stop**,
 - ✓ une **action** en réponse de la condition précédente.

```
deploy-demo:
  stage: deploy
  environment: demo # Déclaration simple de l'environnement
  script:
    - make deploy

deploy-production:
  environment: # Déclaration étendue de l'environnement
    name: production
    url: 'https://blog.eleven-labs/fr/gitlab-ci/' # Url de l'application
  script:
    - make deploy
```

VARIABLES

- Cette déclaration permet de définir des variables pour tous les jobs ou pour un job précis.
- Ceci revient à déclarer des variables d'environnement.

```
variables: # Déclaration de variables pour tous les `job`  
  SYMFONY_ENV: prod  
  
build:  
  script: echo ${SYMFONY_ENV} # Affichera "prod"  
  
test:  
  variables: #Déclaration et réécriture de variables globales pour ce `job`  
    SYMFONY_ENV: dev  
    DB_URL: '127.0.0.1'  
  script: echo ${SYMFONY_ENV} ${DB_URL} # Affichera "dev 127.0.0.1"
```

- Il est aussi possible de déclarer des variables depuis l'interface web de GitLab **Settings > CI/CD > Variables** et de leur spécifier un environnement.

The screenshot shows the 'Variables' settings page in GitLab. It features a table with columns for 'Variable name', 'Variable value', 'Protected', and 'Environment'. The first three rows are pre-filled with 'BD_PASSWORD', 'DB_PASSWORD', and another 'DB_PASSWORD', each with a corresponding value and a 'production' environment. The last row is a template for 'Input variable key' and 'Input variable value'. The 'Protected' column has toggle switches, and the 'Environment' column has dropdown menus. At the bottom, there are 'Save variables' and 'Hide values' buttons.

Variable name	Variable value	Protected	Environment
BD_PASSWORD	db_password_production	<input checked="" type="checkbox"/>	production
DB_PASSWORD	db_password_demo	<input checked="" type="checkbox"/>	demo
DB_PASSWORD	db_password_no_env	<input type="checkbox"/>	All environments
Input variable key	Input variable value	<input type="checkbox"/>	All environments

[Save variables](#) [Hide values](#)

CACHE

- Le cache est intéressant pour spécifier une liste de fichiers et de répertoires à mettre en cache tout le long de votre pipeline. Une fois la pipeline terminée le cache sera détruit.
- Plusieurs sous-directives sont possibles :
 - ✓ **paths** : obligatoire, elle permet de spécifier la liste de fichiers et/ou répertoires à mettre en cache
 - ✓ **policy** : facultative, elle permet spécifier que le cache doit être récupéré ou sauvegardé lors d'un job (push ou pull).

```
build:
  stage: build
  image: node:8-alpine
  script: yarn install && yarn build cache:
    paths:
      - build # répertoire mis en cache
    policy: push # le cache sera juste sauvegardé, pas de récupération d'un cache existant

deploy:
  stage: deploy
  script: make deploy
  cache:
    paths:
      - build
    policy: pull # récupération du cache
```

ARTIFACTS

- Les artefacts sont un peu comme du cache mais ils peuvent être récupérés depuis une autre pipeline. Comme pour le cache il faut définir une liste de fichiers ou/et répertoires qui seront sauvegardés par GitLab. Les fichiers sont sauvegardés uniquement si le job réussit.
- Nous y retrouvons cinq sous-directives possibles :
 - ✓ **paths** : obligatoire, elle permet de spécifier la liste des fichiers et/ou dossiers à mettre en artifact
 - ✓ **name** : facultative, elle permet de donner un nom à l'artifact. Par défaut elle sera nommée artifacts.zip
 - ✓ **untracked** : facultative, elle permet d'ignorer les fichiers définis dans le fichier .gitignore
 - ✓ **when** : facultative, elle permet de définir quand l'artifact doit être créé. Trois choix possibles on_success, on_failure, always. La valeur on_success est la valeur par défaut.
 - ✓ **expire_in** : facultative, elle permet de définir un temps d'expiration

```
job:  
  script: make build  
  artifacts:  
    paths:  
      - dist  
    name: artifact:build  
  when: on_success  
  expire_in: 1 weeks
```

RETRY

- Cette déclaration permet de **ré-exécuter** le job en cas d'échec.
- Il faut indiquer le nombre de fois où vous voulez ré-exécuter le job

```
job:retry:  
  script: echo 'retry'  
  retry: 5
```

Plus loin dans l'utilisation de GitLab

Les runners partagés

- Sur GitLab.com, on retrouve notamment les Runners par défaut qui sont aussi couramment appelés Runners partagés (**Shared runners**).
- Ils ne nécessitent aucune installation, ni configuration et peuvent être amenés à exécuter vos tâches.
- Chaque Runner dispose d'un ensemble de tags permettant d'exécuter des jobs spécifiques en fonction de ceux mentionnés dans le fichier « .gitlab-ci.yml ».
 - Assigner à un Runner des tâches particulières en fonction de plusieurs paramètres :
 - système d'exploitation,
 - environnement d'exécution,
 - capacités techniques (CPU et RAM) de la machine,
 - ...

Les runners auto-gérés

- Il est possible d'installer de créer des runners sur une infrastructure privée
- Plusieurs avantages :
 - Minimiser le temps d'exécution de la pipeline;
 - Avoir la main ou contrôler la configuration du Runner, mais aussi les couches plus basses comme le système d'exploitation;
 - Déployer au plus proche de l'environnement cible.
 - Héberger sa propre installation de GitLab. Dans cette situation, une installation d'au moins un Runner est requise car il n'y en a pas par défaut.

Installation de Gitlab Runner

Les recommandations de <https://docs.gitlab.com/runner/install/>

■ Les OS supportés

- ✓ CentOS,
- ✓ Debian
- ✓ Ubuntu
- ✓ RHEL
- ✓ Fedora
- ✓ Mint
- ✓ **Windows**
- ✓ **macOS**
- ✓ FreeBSD

Repositories

- [Install using the GitLab repository for Debian/Ubuntu/CentOS/RedHat](#)

Binaries

- [Install on GNU/Linux](#)
- [Install on macOS](#)
- [Install on Windows](#)
- [Install on FreeBSD](#)
- [Install nightly builds](#)

Containers

- [Install as a Docker service](#)
- [Install on Kubernetes](#)
- [Install using the Kubernetes Agent](#)
- [Install on OpenShift](#)

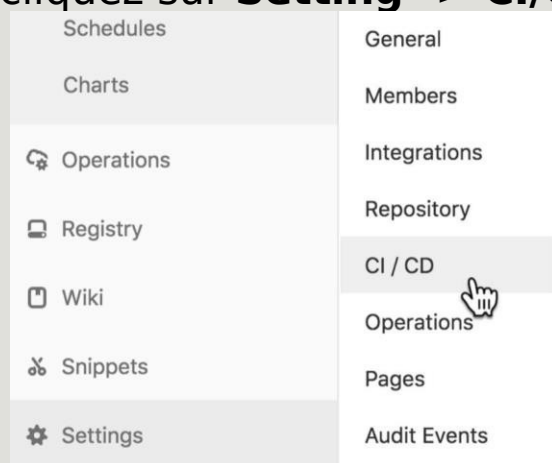
Autoscale

- [Install in autoscaling mode using Docker machine](#)
- [Install the registry and cache servers](#)

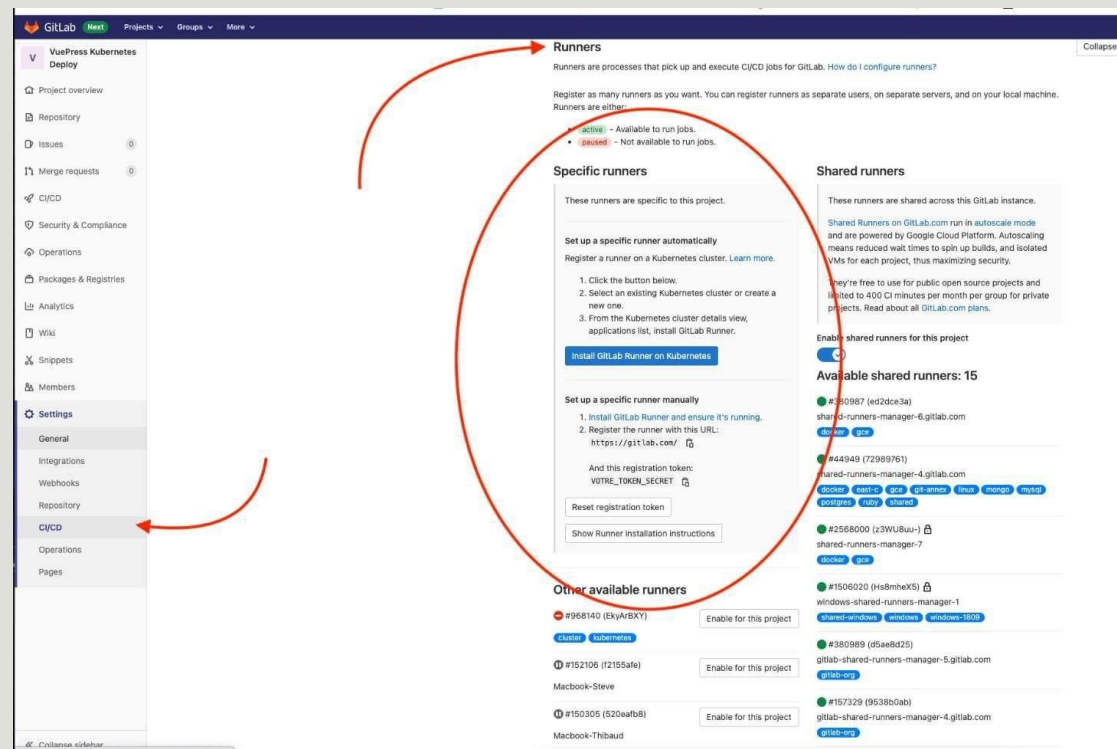
Installation d'un runner avec Docker

■ Étape 1 : Récupérer les informations des runners spécifiques (Specific runners) de votre projet

- Dans le projet **Car assembly line**, cliquez sur **Setting -> CI/CD**



- Dans l'interface **Runners**, cliquez sur le bouton **Expand**



Runners

Register and see your runners for this project.





Installation d'un runner avec Docker

■ Étape 2 : Enregistrement du Runner avec Gitlab-CI

- L'étape d'enregistrement n'est à réaliser qu'une seule fois. Elle a pour but d'autoriser Gitlab à communiquer avec votre runner, elle s'assure aussi que seuls vos jobs vont être lancés sur votre Runner.
- Pour enregistrer un Runner tapez la commande suivante:

```
docker run --rm -it -v $(pwd)/config:/etc/gitlab-runner gitlab/gitlab-runner register
```

```
stage@kubernetes-worker:~$ docker run --rm -it -v $(pwd)/config:/etc/gitlab-runner gitlab/gitlab-runner register
Runtime platform                                arch=amd64 os=linux pid=7 revision=4b9e985a version=14.4.0
Running in system-mode.

Enter the GitLab instance URL (for example, https://gitlab.com/):
https://gitlab.com/
Enter the registration token:
fUbjSxyFexFwKy9rBSq
Enter a description for the runner:
[100d8abcc0c0]: Démo Docker runner
Enter tags for the runner (comma-separated):

Registering runner... succeeded                  runner=fUbjSxy
Enter an executor: docker-ssh, shell, ssh, virtualbox, docker-ssh+machine, kubernetes, custom, docker, parallels, docker+machine:
docker
Enter the default Docker image (for example, ruby:2.6):
alpine
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!
```



Installation d'un runner avec Docker

■ Étape 3 : Lancer le runner

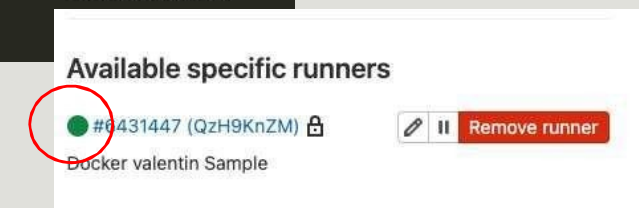
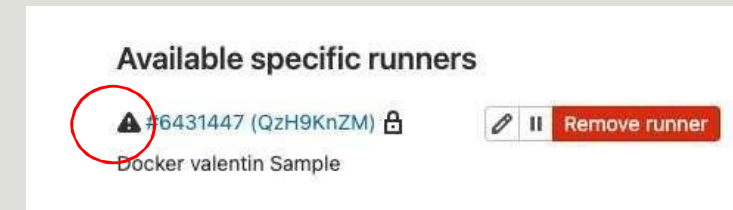
- Notre runner est maintenant connu de Gitlab, il n'est pour l'instant par contre pas encore en fonction.
- Pour le lancer on réutilise évidemment Docker, via la commande suivante :

```
docker run -d --name gitlab-runner --restart always \
-v $(pwd)/config:/etc/gitlab-runner \
-v /var/run/docker.sock:/var/run/docker.sock \
gitlab/gitlab-runner:latest
```

- Cette action lance un Container Docker visible via la commande docker ps :

```
runner ➤ docker ps
CONTAINER ID   IMAGE                COMMAND                  CREATED        STATUS        PORTS        NAMES
61569fcfdda5   gitlab/gitlab-runner:latest  "/usr/bin/dumb-init ..."  7 seconds ago  Up 6 seconds  -           gitlab-runner
runner ➤ |
```

- Votre runner est maintenant actif sur Gitlab-CI :





Installation d'un runner avec Docker

- **Étape 4 : Configuration du runner**

- Dans la dernière version des runners, il existe un Bug au niveau de la configuration des volumes.
- Pour corriger ce Bug, éditez le fichier « **config/config.toml** »

```
sudo nano config/config.toml
```

- Ajouter dans la déclaration volumes la valeur suivante: ***"/var/run/docker.sock:/var/run/docker.sock"***

```
[runners.docker]
...
volumes = ["/cache", "/var/run/docker.sock:/var/run/docker.sock"]
```

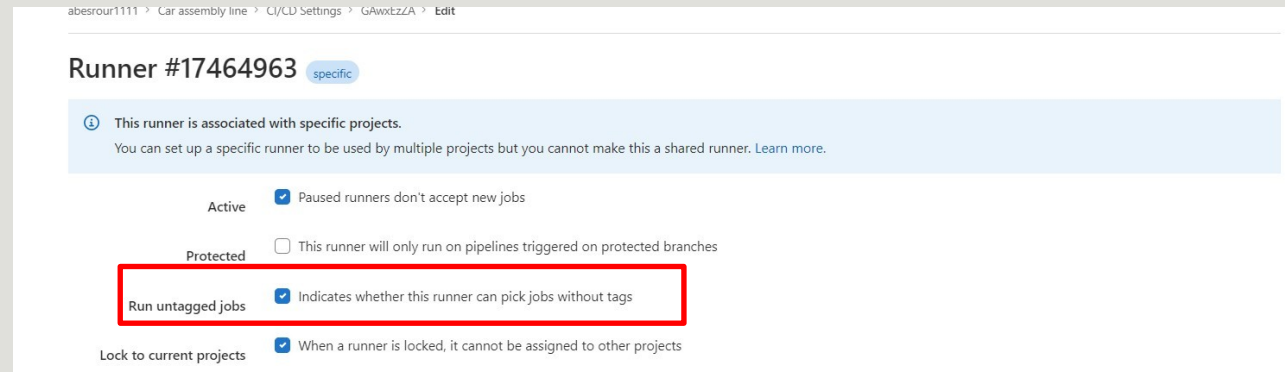
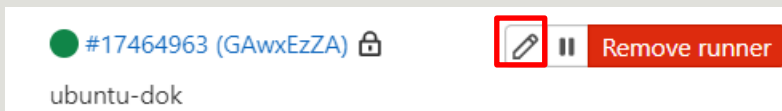
- Redémarrez le conteneur avec la commande:

```
docker restart gitlab-runner
```

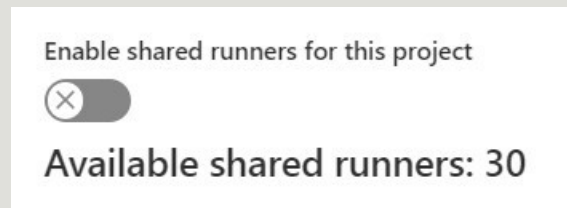


Installation d'un runner avec Docker

- **Étape 5 : Configuration du runner depuis gitlab**
- En dernier lieu, certaines configurations doivent être terminées dans le site gitlab:
 - Activer la prise en compte des jobs non taggués par le runner:



- Désactiver les runners partagés (shared runners) depuis l'interface des settings CI/CD



Scalabilité automatique (autoscaling) des runners

- Le **Runner standard**, qui utilise l'exécuteur **Docker, Shell, SSH, VirtualBox** ou **Parallels**, exécute une seule instance directement sur le serveur GitLab Runner.
- Le **Runner autoscaler** est différent du premier concept car ce n'est pas l'instance même qui exécutera les conteneurs et donc vos jobs GitLab CI, mais elle pilotera comme une tour de contrôle la création d'autres instances (agents):
 - D'autres machines virtuelles si l'exécuteur est **Docker+Machine**
 - D'autres conteneurs si l'exécuteur est **Docker-autoscaler** (Béta)
- Une fois les jobs terminés ou si le Runner n'est plus sollicité, les instances "agents" seront détruites.

Autoscaling des runners avec Docker Machine

- **Docker machine** existe depuis les débuts de Docker, et qui a pris son essor lorsque **Docker Desktop** n'existait pas encore.
- Cet outil permet de gérer un parc de machines faisant tourner docker engine, et de configurer le client docker pour se connecter directement vers l'une des machines de ce parc.
- Docker a obsolète Docker Machine et l'a remplacé par **Docker Desktop**.
- GitLab maintient un fork Docker Machine pour les utilisateurs de GitLab Runner qui s'appuient sur l'exécuteur **Docker+Machine**.
- **GitLab Runner Autoscaler** est le successeur de Docker Machine

```
# curl -O "https://gitlab-docker-machine-downloads.s3.amazonaws.com/main/docker-machine-Linux-x86_64"  
cp docker-machine-Linux-x86_64 /usr/local/bin/docker-machine  
chmod +x /usr/local/bin/docker-machine
```

Autoscaling des runners avec Docker Autoscaler

- **Docker Autoscaler** crée des instances à la demande pour prendre en charge les jobs exécutés par le runner.
- Il encapsule **Docker executor** afin que toutes les options et fonctionnalités de l'exécuteur Docker soient prises en charge.
- Il utilise des plugins de **Fleeting** pour l'autoscaling. **Fleeting** est une abstraction pour un groupe d'instances *autoscalé*, qui en charge les fournisseurs de cloud, tels que **GCP**, **AWS** et **Azure**.
- Il est en version **Béta** depuis **GitLab Runner 16.6**

Kubernetes executor

- **Kubernetes executor** utilise les clusters kubernetes pour exécuter les jobs de GitLab ci.
- Un **cluster kubernetes** est un ensemble de nœuds qui exécute des Pods
- Un **Pod** est est un ensemble composé d'un ou plusieurs conteneurs (Docker ou autre)
- Un nouveau Pod est crée pour l'exécution de chaque job

Monitoring de Gitlab

- La solution utilisée pour la surveillance des instances GitLab et GitLab Runner est **Prometheus**
- Prometheus permet le monitoring de n'importe quelle système et pas seulement GitLab
- Les agents de Prometheus, appelés **exporters**, exportent les métriques sur des ports standards.
- Il sauvegarde les métriques collectés régulièrement dans sa base de données de type time-series
- Prometheus et ses exporters sont activés par défaut sur le serveur GitLab
- Parmi les exporters utilisés par Gitlab :
 - Node exporter
 - Web exporter
 - Redis exporter
 - PostgreSQL exporter

Merci pour votre attention

