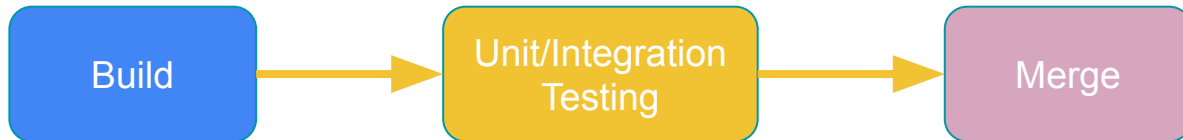


CI : Continuous Integration

Dans l'intégration continue, les développeurs font des merge de leurs changements de code sur la branche principale le plus souvent possible

Les changements sont validés en lançant un build et des tests automatiques unitaires

Cela évite les challenges d'intégration manuels et vérifie si l'application n'est pas abîmée par ce changement

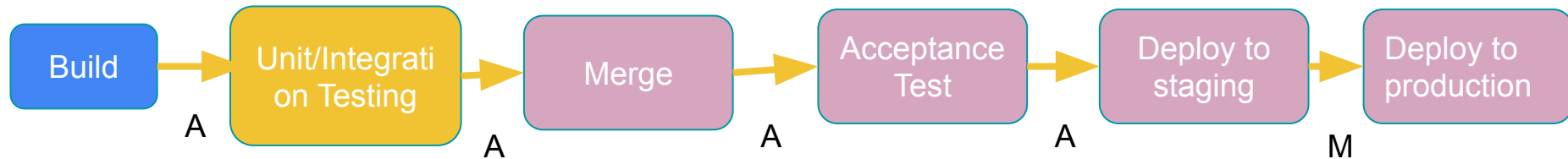


CD : Continuous Delivery

C'est une extension de l'intégration continue mais tout le code est automatiquement déployé dans un environnement de test (ou staging)

La mise en production finale est manuelle

Le but de la CD est de toujours avoir un code qui est prêt à être déployé



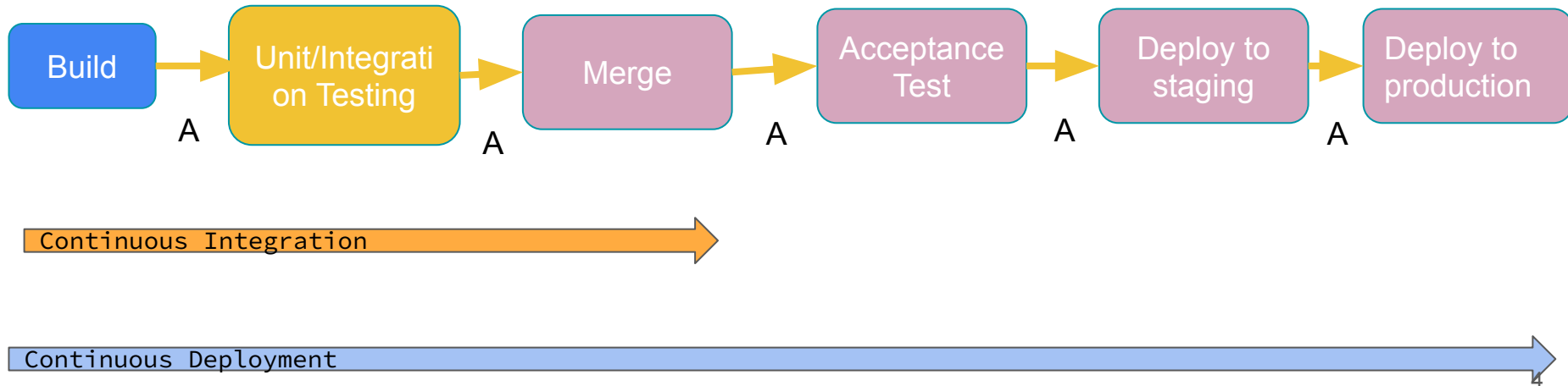
Continuous Integration →

Continuous Delivery →

CD : Continuous Deployment

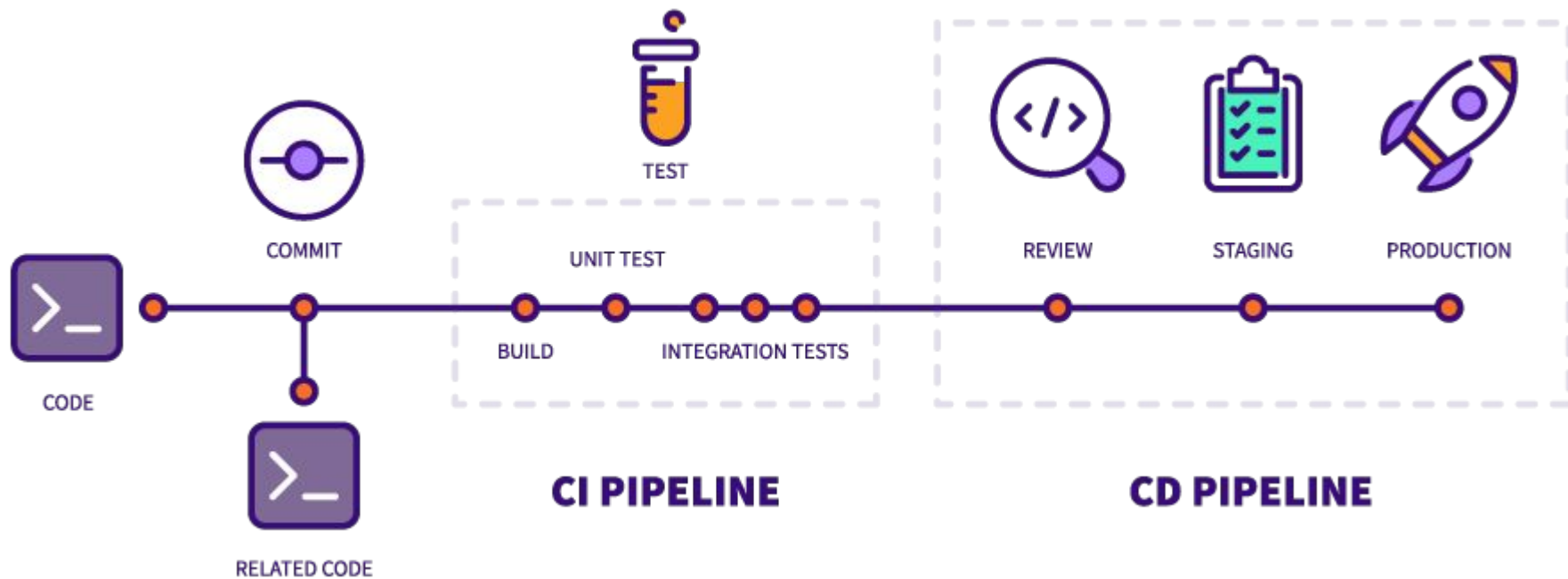
Les modifications qui ont passé toutes les étapes du pipeline de production sont directement placées pour le client en production

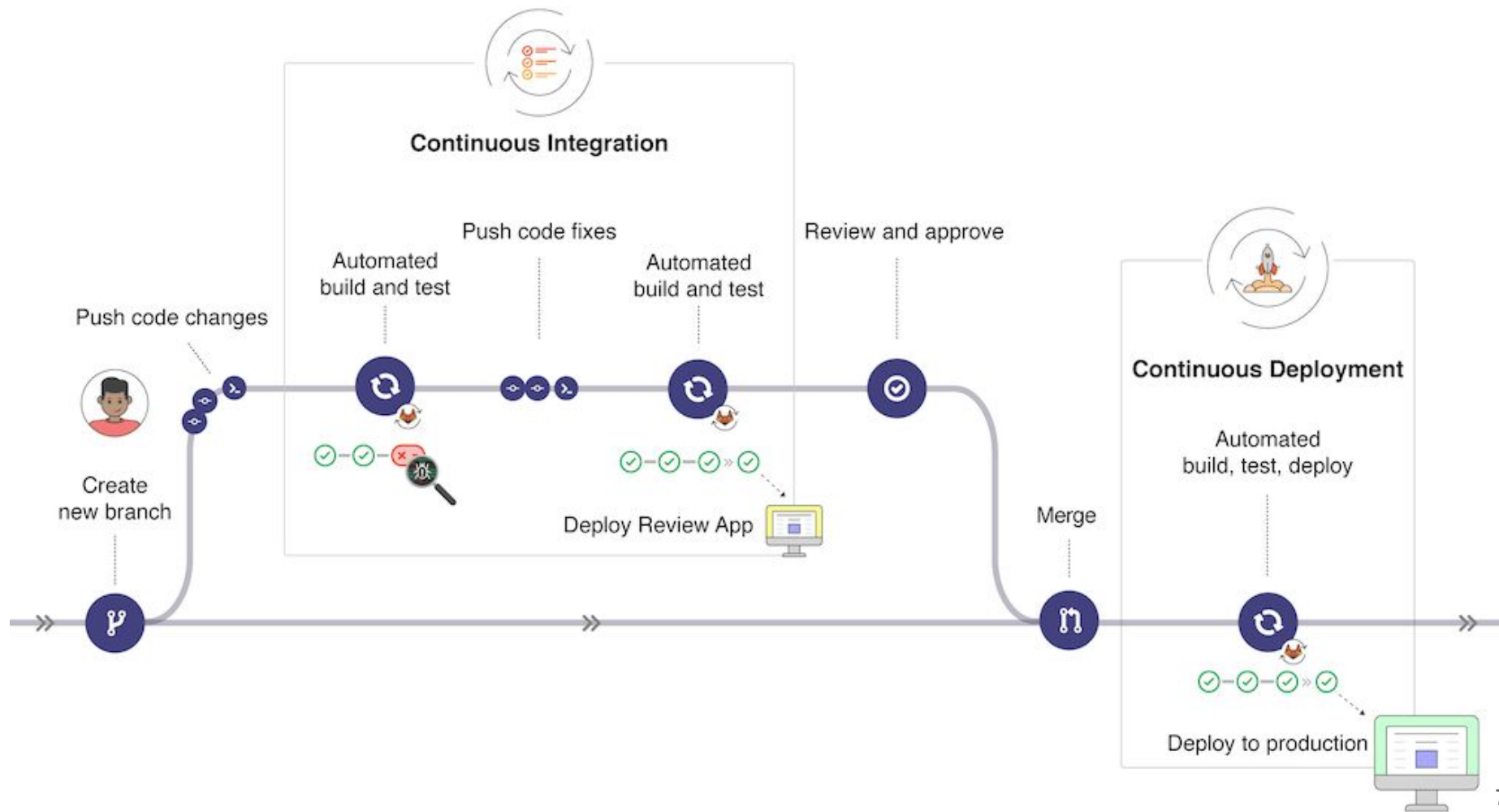
Aucune intervention humaine, complètement automatisé

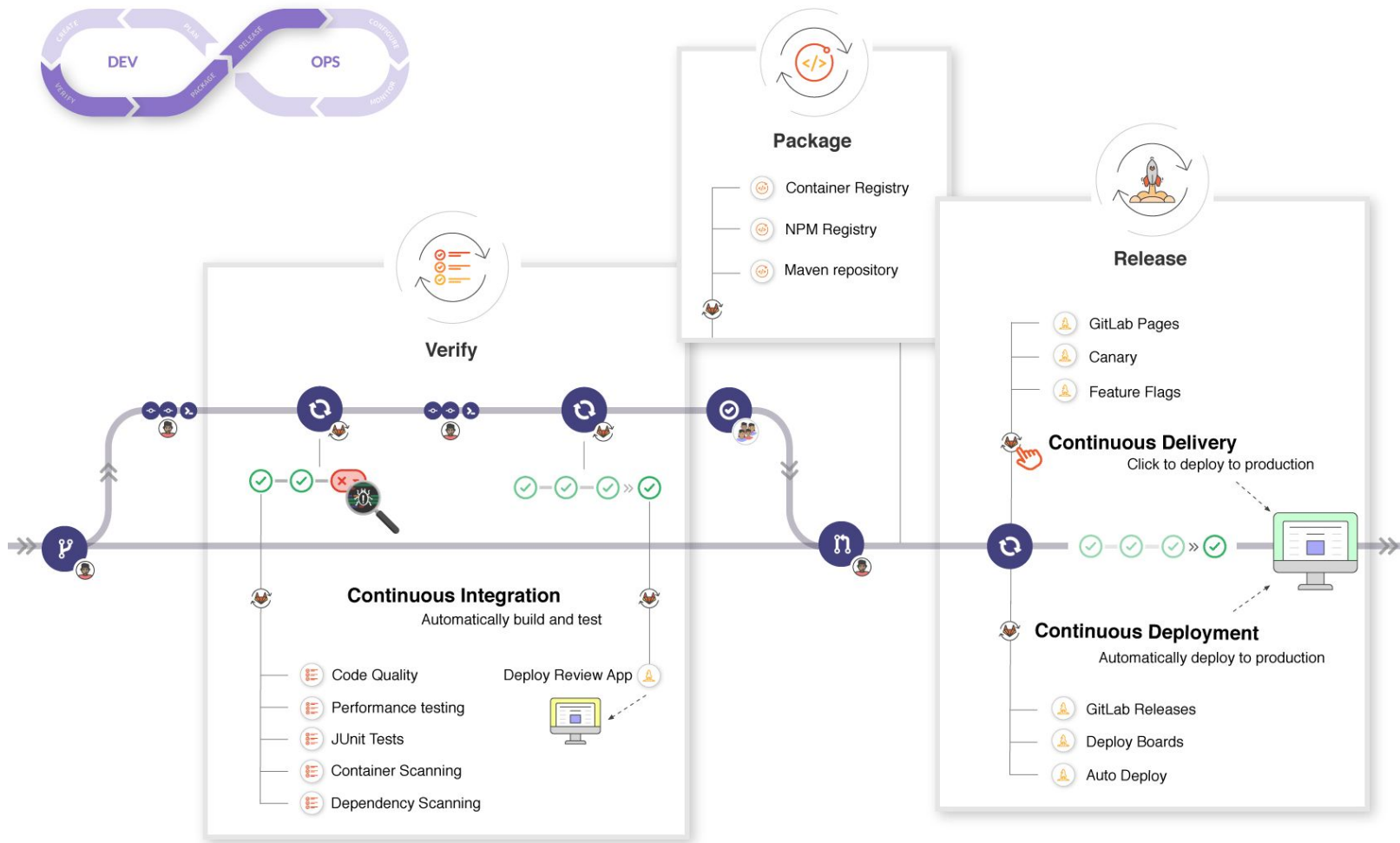




UAT : User Acceptance Testing







Avantages CI/CD

- **Vitesse** : l'intégration continue est censée être rapide avec un retour instantané
- **Précision** : L'application de l'automatisation au processus de déploiement est un excellent point de départ
- **Fiabilité** : avoir un pipeline CI/CD fiable améliore considérablement la vitesse de création et de déploiement de nouveaux commits
- **Modifications de code plus petites** : à l'aide de tests continus, ces petites pièces peuvent être testées dès qu'elles sont intégrées dans le référentiel de code
- **Taux de libération plus rapide** : les défaillances sont détectées plus rapidement et, en tant que telles, peuvent être réparées plus rapidement, ce qui entraîne une augmentation des taux de libération

Étapes CI/CD

Il n'y a pas d'étapes définies de CI/CD. Cependant, CI/CD est une méthodologie très flexible et les développeurs peuvent définir ces étapes selon leur choix. Il y a ci-dessous un choix standard d'étapes CI/CD et celles-ci sont toutes en préparation si les étapes précédentes échouent, elles ne passeront pas à l'étape suivante.

Étapes CI/CD

1. **Contrôle de version** : Utilisé pour gérer les changements de code, de documentation, etc. Par exemple, GitLab, GitHub, TFS, etc.
2. **Build** : cette étape crée la solution du produit logiciel et stocke les artefacts
3. **Test unitaire** : Après avoir réussi à créer la solution à l'étape 2, elle passera aux tests unitaires s'il en a.
4. **Déployer** : Après avoir réussi les étapes de construction et de test unitaire, la solution sera déployée dans un environnement temporaire.
5. **Test automatisé** : cette étape exécutera tous les tests automatisés pour vérifier que le produit est capable de répondre à toutes les exigences avant son déploiement en production.
6. **Déployer en production** : après avoir exécuté avec succès tous les cas de test et la vérification, cette étape va maintenant déployer le produit dans un environnement de production.

Softwares CI/CD

Il existe de nombreux outils logiciels disponibles qui fournissent une implémentation de CI/CD, et certains sont ci-dessous et ceux-ci peuvent être utilisés par les développeurs ou le choix des entreprises et chacun a ses propres caractéristiques :

- Jenkins
- Circleci
- TeamCity
- Bambou
- GitLab

Configuration requise pour GitLab CI/CD

Il peut y avoir n'importe quel outil logiciel pour implémenter CI/CD comme mentionné ci-dessus. Cependant, GitLab fournit une implémentation CI/CD de manière flexible, et sans beaucoup de connaissances ou d'autres logiciels supplémentaires, toute personne ayant des connaissances de base sur le développement logiciel peut facilement configurer la configuration CI/CD avec GitLab.

Voici quelques exigences :

Configuration requise pour GitLab CI/CD

1 - Contrôle de version : GitLab est lui-même un contrôle de version pour implémenter CI/CD

2 - GitLab Runner : C'est un outil important pour implémenter CI/CD et il faut l'installer avant toute configuration de CI/CD. Il récupère et exécute des tâches pour GitLab lorsque quelqu'un valide les modifications. Il existe deux types de coureurs : les coureurs partagés et les coureurs spécifiques.

- **Runner partagé** : ces coureurs exécutent le code de différents projets sur le même coureur et utilisent des coureurs partagés lorsque vous avez plusieurs tâches avec des exigences similaires.
- **Runner spécifique** : Il s'agit d'un runner individuel et chaque projet sera exécuté par un runner séparé et géré par le seul développeur qui travaille sur ce projet spécifique.

Exécuteurs GitLab Runner : GitLab Runner implémente un certain nombre d'exécuteurs qui peuvent être utilisés pour exécuter des builds dans différents scénarios. Il existe plusieurs exécuteurs disponibles dans GitLab, qui peuvent être implémentés en fonction des exigences du projet ou de la disponibilité des outils logiciels. Certains exécuteurs sont ci-dessous:

- **SSH** : l'exécuteur SSH ne prend en charge que les scripts générés dans Bash et la fonction de mise en cache n'est actuellement pas prise en charge. Il s'agit d'un exécuteur simple qui vous permet d'exécuter des builds sur une machine distante en exécutant des commandes via SSH.
- **Shell** : Shell est l'exécuteur le plus simple à configurer. Toutes les dépendances requises pour vos builds doivent être installées manuellement sur la même machine que celle sur laquelle Runner est installé.
- **Parallels** : cette configuration d'exécution est utilisée avec Virtual Box.
- **VirtualBox** : Ce type d'exécuteur permet d'utiliser une machine virtuelle déjà créée, qui est clonée et utilisée pour exécuter votre build.

- **Docker** : une excellente option consiste à utiliser Docker car il permet un environnement de construction propre, avec une gestion facile des dépendances (toutes les dépendances pour la construction du projet peuvent être placées dans l'image Docker).
- **Docker Machine (auto-scaling)** : La Docker Machine est une version spéciale de l'exécuteur Docker avec prise en charge de l'auto-scaling. Il fonctionne comme l'exécuteur Docker normal, mais avec la construction, les hôtes sont créés à la demande par Docker Machine.
- **Exécuteur Kubernetes** : L'exécuteur Kubernetes permet d'utiliser un cluster Kubernetes existant pour vos builds.
- **Exécuteur personnalisé** : L'exécuteur personnalisé permet de spécifier vos propres environnements d'exécution

Fichier Yml : le fichier Yml est nécessaire pour exécuter le CI/CD et il devra être stocké dans un fichier source du projet, et il comprend la configuration de chaque technologie et méthode de traitement.



GitLab



Google Cloud





Setup

Test

Build

Release

Deploy



setup



test



build



release



deploy_produc...

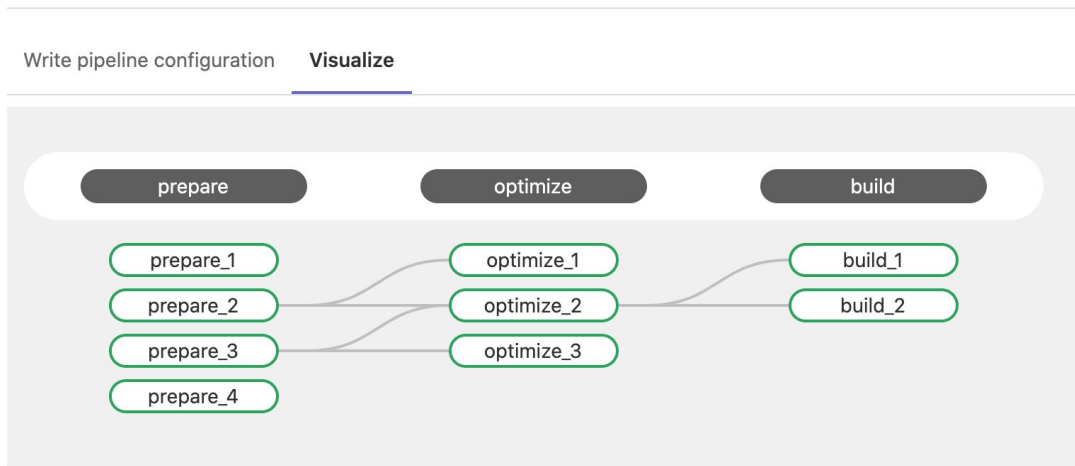


deploy_staging



Visualiser la configuration CI

Pour afficher une visualisation de votre configuration gitlab-ci.yml, dans votre projet, accédez à **CI/CD > Editor** , puis sélectionnez l'onglet **Visualize** . La visualisation montre toutes les étapes et tous les travaux. Toutes les relations de dépendance sont affichées sous forme de lignes reliant les travaux ensemble, montrant la hiérarchie d'exécution :



GitLab CI est un service open-source inclus dans GitLab intégré par défaut dans tout projet depuis la version 8.0.

Pour l'utiliser il faut avoir les droits selon que l'on soit admin, ou maintenir ou créateur du dépôt...

Pour créer un pipeline CI/CD, nous avons besoin de créer un fichier `.gitlab-ci.yml` à la racine du projet

Avant cela nous avons besoin d'utiliser des **Gitlab Runner**

Pour gérer vos pipelines, il faut mettre en place un GitLab Runner. Le GitLab Runner va gérer vos jobs et les lancer automatiquement quand une branche sera envoyée sur le dépôt ou lorsqu'elle sera mergée, par exemple. Vous pouvez également lancer les jobs à la main ou changer complètement la configuration.

Si vous utilisez Gitlab.com, plusieurs runners sont mis à votre disposition gratuitement. Vous pouvez alors lancer plusieurs pipelines en parallèles pour avoir les retours plus rapidement.

Ce sont des **Shared Runners**, ce qui veut dire qu'ils sont utilisés sur l'ensemble de vos projets, ce qui permet de limiter le nombre de runners à installer. Mais cela peut aussi poser problème, si vous avez beaucoup d'activité sur un projet. Les pipelines des autres projets seront alors en attente le temps que les **Shared runners** soient à nouveau disponibles.

En fonction de vos besoins, cela ne sera peut-être pas suffisant. Selon le nombre de projets que vous avez ou de l'activité que vous avez sur un projet, il vous faudra davantage de runners ou beaucoup de patience.

GitLab Runner

C'est un service open-source inclus dans Gitlab <https://docs.gitlab.com/runner/>

Utilisé pour lancer des jobs et envoyé les résultats en retour à GitLab

Nous allons

1. installer GitLab Runner
2. l'enregistrer
3. et le démarrer

1 - Aller sur le lien de l'installation → Installer selon votre OS → vérifier avec `gitlab-runner --version` et vérifier que le service est démarré

GitLab Runner

2 - Enregistrement du gitlab runner

Suivre les steps de la docs gitlab

Pour avoir votre token pour votre runner :

- Aller dans votre projet
- Settings
- CI/CD
- Runners
- Expand
- Set up a specific runner manually

GitLab Runner

2 - Enregistrement du gitlab runner

```
gitlab-runner register
```

```
https://gitlab.com
```

```
qf5qsd6f4s6D5F46d4f
```

```
my-runner
```

```
ssh,ci
```

```
shell
```

GitLab Runner

3 - Démarrer le service gitlab-runner

4 - Vérifier si le runner est activé dans le projet

GitLab CI/CD

Nous pouvons maintenant créer notre fichier `.gitlab-ci.yml` à la racine de notre repository (le tags ici appelle le runner avec le tag assigné lors de sa déclaration)

```
demo_job_1:  
  tags:  
    - ci  
  script:  
    - echo Hello World
```

Vous pouvez vérifier votre syntaxe sur n'importe quel yaml file validator

exemple : <http://www.yamllint.com/>

Nous devons ensuite commiter et pousser le fichier vers notre repo gitlab

```
git init
```



.git



.gitlab-ci.yml

```
cd <project-folder>
```

```
git status
```

```
git add .
```

```
git commit -m "add .gitlab-ci.yml"
```

```
git add origin <url_repo>
```

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

Maintenant le fichier `.gitlab-ci.yml` est dans notre repo.

Démarrer maintenant le service gitlab-runner ou vérifier qu'il est démarré

Changer un élément dans le dépôt local et commiter avec un push

Ensuite aller dans CI/CD

Un pipeline avec du Go pour tester : utilise le dépôt

https://gitlab.com/flying_kiwi/go-ci-demo

```
├── src
│   ├── alpha.go
│   ├── alpha_test.go
│   └── beta
│       ├── beta.go
│       └── beta_test.go
├── .gitignore
├── .gitlab-ci.yml
└── README.md
```

Mettre en place et exécuter des tests CI et automatisés pour un projet Go est assez simple. La [publication officielle de Gitlab](#) est complète, mais compliquée car elle implique la création d'une image docker et l'utilisation de [Makefiles](#) comme niveau d'abstraction supplémentaire.

Voyons comment nous pourrions construire un projet Go très simple.

Ici les deux alpha.go et beta.go sont des paquets autonomes destinés à être exécutés avec go run(ou avoir un binaire construit avec go build).

Image de base

Tout d'abord, le programme d'exécution Gitlab devra récupérer une image docker avec les outils Golang installés. Nous utilisons [l'image officielle du Golang](#). Les balises par défaut 1.12.6, 1.12, 1, latest vous obtiendrez une image basée sur Debian (version 9 alias Stretch au moment de la rédaction), mais vous pouvez essayer les balises [Alpine Linux](#) (suffixe alpine) si vous voulez une image plus légère.

Nous allons verrouiller une version plutôt que d'utiliser latest pour que les futurs changements de Go ne cassent pas les choses. Le versioning sémantique devrait signifier que toutes les versions 1.x de Go seront rétrocompatibles, mais il est bon d'en être sûr.

```
image: golang:1.12.6
```

Etapes

Nous voulons seulement exécuter des tests, pas construire un binaire ou déployer, nous n'aurons donc qu'une seule étape dans notre pipeline CI.

```
stages:  
  - test
```


Dépendances

Nous devons maintenant extraire toutes les dépendances de nos fichiers Go. Pour pouvoir utiliser `go get`, nous devons mettre le référentiel dans le **\$GOPATH**, qui par défaut est `/go`. De plus, nos projets doivent s'inscrire dans le `/src/<git domain>/<namespace>/<project>annuaire`

before_script:

- `mkdir -p /go/src/gitlab.com/flying_kiwi /go/src/_/builds`
- `cp -r $CI_PROJECT_DIR /go/src/gitlab.com/flying_kiwi/go-ci-demo`
- `ln -s /go/src/gitlab.com/flying_kiwi /go/src/_/builds/flying_kiwi`
- `go get -v -d ./...`

Essais

Pour exécuter les tests sur tous les fichiers .go dans votre projet, vous pouvez exécuter **go test ./...** à partir du dossier racine.

```
unit_tests:  
  stage: test  
  script:  
    - go test -v ./..
```

Couverture

A ce stade, tous nos tests sont en cours. Qu'en est-il de la couverture des tests ? Nous pouvons utiliser le flag `-coverprofile` pour obtenir les données de couverture pour nos tests.

```
script:  
- go test -v ./... -coverprofile .testCoverage.txt
```

Pour laisser Gitlab analyser la sortie de couverture de go test, nous ajoutons l'expression régulière suivante sous Settings > CI/CD > General pipelines > Test coverage parsing

total:\s+\(statements\)\s+(\d+\.\d+\%)

Vous pouvez désormais ajouter des [badges](#) à votre fichier README afin de voir l'état et la couverture du pipeline directement depuis la page du référentiel !

Regarder sous Settings > CI/CD > General pipelines et faites défiler vers le bas pour trouver le Pipeline status et Coverage report sections. Prenez le markdown snippet, insérez-le dans votre README.md, et voilà !

Golang Gitlab CI/CD Demo

pipeline

passed

coverage

55.60%

A simple project to demonstrate how to use Gitlab CI/CD with Go.

Install dependencies with:

```
go get -v -d ./...
```

Run the scripts with:

```
go run src/alpha.go  
go run src/beta/beta.go
```

Run the tests with

```
go test -v ./...
```

Mise en cache

Le [didacticiel officiel Gitlab](#) pour exécuter CI/CD avec Golang suggère ce qui suit :

```
cache:  
  paths:  
    - /apt-cache  
    - /go/src/github.com  
    - /go/src/golang.org  
    - /go/src/google.golang.org  
    - /go/src/gopkg.in
```

Mais *cela ne fonctionne pas* . Pourquoi? Gitlab autorise uniquement la mise en cache des fichiers à l' *intérieur* de votre référentiel. Ainsi, les chemins ci-dessus résolvent par exemple **/home/user/toto/demo-go-ci/go/src/github.com**.

Cette mise en cache de sous-répertoire fonctionne très bien pour les projets Node, par exemple, où node_modules sera un sous-répertoire de votre projet. Mais pour Go, toutes les dépendances sont stockées sur le gopath /go. Et évidemment /apt-cache est complètement sorti.

Variables

Enfin, nous allons refactoriser ce script avec quelques variables afin qu'il soit plus facilement portable vers d'autres projets. Nous utiliserons également la variable `$GOPATH` au cas où le chemin par défaut de notre image de base changerait de `/go/`.

Le fichier final `.gitlab-ci.yml` ressemble à ceci :

image: golang:1.12.6

variables:

REPO: gitlab.com

GROUP: toto

PROJECT: go-ci-demo

stages:

- test

before_script:

- mkdir -p \$GOPATH/src/\$REPO/\$GROUP \$GOPATH/src/_/builds
- cp -r \$CI_PROJECT_DIR \$GOPATH/src/\$REPO/\$GROUP/\$PROJECT
- ln -s \$GOPATH/src/\$REPO/\$GROUP \$GOPATH/src/_/builds/\$GROUP
- go get -v -d ./...

unit_tests:

stage: test

script:

- go test -v ./... -coverprofile .testCoverage.txt

Déployer une application Node.js avec Gitlab CI/CD



- 1 - Créer un dépôt sur GitLab
- 2 - Le lier avec un repo local, et y créer deux fichiers :
 README et .gitignore
- 3 - Commiter les changements
- 4 - push sur repo distant (push -u origin master)

5 - initialiser le projet node

6 - installer l'extension express

7 - fichier index.js à la racine :

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send({ hello: "world" });
});

const PORT = process.env.PORT || 5000;
app.listen(PORT, function() {
  console.log(`App listening on port ${PORT}`);
});
```

8 - lancer le serveur

9 - vérifier sur le browser

10- Couper le serveur

11 - Utilisation de Heroku






Jump to Favorites, Apps, Pipelines, Spaces...



 Personal ▾

New ▾

 node-heroku-dpl

-  Create new app
-  Create new pipeline

11 - Utilisation de Heroku

Avant de lancer le CI/CD de GitLab, il faut préciser à Heroku quelle version de Node.js et de npm nous disposons

```
...  
"engines": {  
  "node": "8.1.1",  
  "npm": "5.0.3"  
},  
"scripts": {  
  "start": "node index.js"  
},  
...
```



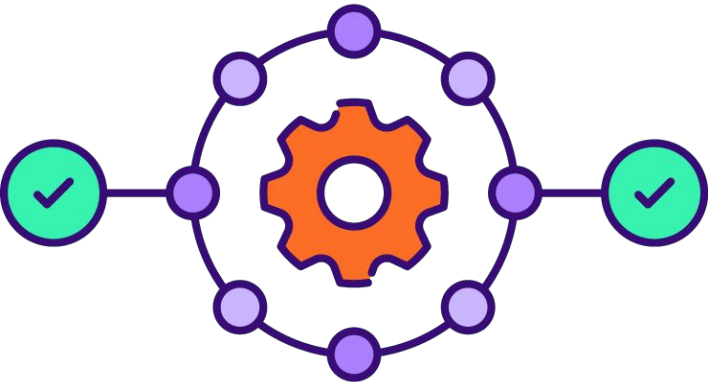
12 - Relancer le serveur

Heroku se lance en utilisant npm



13 - Configuration de Gitlab CI/CD

Créer un fichier `.gitlab-ci.yml` à la racine

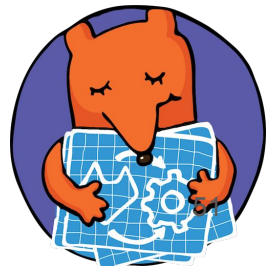


14 - Fichier .gitlab-ci.yml

```
image: node:latest

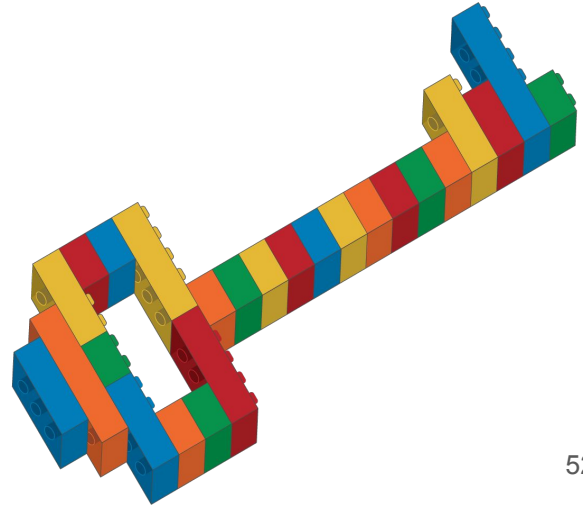
stages:
  - production

production:
  type: deploy
  stage: production
  image: ruby:latest
  script:
    - apt-get update -qy
    - apt-get install -y ruby-dev
    - gem install dpl
    - dpl --provider=heroku --app=node-heroku-dpl --api-key=$HEROKU_API_KEY
  only:
    - master
```



15 - Insérer l'API key de Heroku dans
Gitlab (voir account settings de Heroku)

Settings GitLab > CI/CD > Secret
Variables





N



Seulkiro Park > node-heroku-dpl > CI / CD Settings

General pipelines settings

[Expand](#)

Update your CI/CD configuration, like job timeout or Auto DevOps.

Runners settings

[Expand](#)

Register and see your runners for this project.

Secret variables ?

[Collapse](#)

Variables are applied to environments via the runner. They can be protected by only exposing them to protected branches or tags. You can use variables for passwords, secret keys, or whatever you want.

HEROKU_API_KEY

Protected



All environments



Input variable key

Input variable value

Protected



All environments

[Save variables](#)[Reveal value](#)

16 - .gitignore


```
$ echo node_modules > .gitignore
```









17 - Activer les shared Runners


Settings GitLab > CI/CD > Runners


Settings > Activate


 **GitLab** Projects ▾ Groups More ▾


 ▾   10    ▾


N























Runners settings

Collapse

Register and see your runners for this project.

A 'Runner' is a process which runs a job. You can setup as many Runners as you need. Runners can be placed on separate users, servers, and even on your local machine.

Each Runner can be in one of the following states:

- active** - Runner is active and can process any new jobs
- paused** - Runner is paused and will not receive any new jobs

To start serving your jobs you can either add specific Runners to your project or use shared Runners

Specific Runners

Setup a specific Runner automatically

You can easily install a Runner on a Kubernetes cluster. [Learn more about Kubernetes](#)

1. Click the button below to begin the install process by navigating to the Kubernetes page
2. Select an existing Kubernetes cluster or create a new one
3. From the Kubernetes cluster details view, install Runner from the applications list

Install Runner on Kubernetes

Shared Runners

[Shared Runners on GitLab.com](#) run in **autoscale mode** and are powered by DigitalOcean. Autoscaling means reduced wait times to spin up builds, and isolated VMs for each project, thus maximizing security.


They're free to use for public open source projects and limited to 2000 CI minutes per month per group for private projects. Read about all [GitLab.com plans](#).







Disable shared Runners

 for this project










18 - Commiter et pusher les changements

19 - Validation sur Heroku. Aperçu du projet dans Home de GitLab indique un déploiement actif (icône orange) → Cliquez dessus → logs

 GitLab Projects ▾ Groups More ▾

 ▾   10    ▾

N






        


Seulkiro Park > node-heroku-dpl > Details

N

node-heroku-dpl

Node.js express app example demonstrating Heroku deployment with GitLab CI/CD

 Star 0  Fork 0 SSH ▾ git@gitlab.com:seulkiro/node-heroku-dpl   ▾  ▾

 Global ▾



Files (72 KB) Commits (2) Branch (1) Tags (0) CI/CD configuration


Add Changelog



Add License

Add Contribution guide

Add Kubernetes cluster

master ▾ node-heroku-dpl /  ▾ History Find file  ▾

 Initial app deploying to Heroku
Seulkiro Park authored less than a minute ago

 808c82c4 

20 - Consulter le tableau de bord Heroku.
L'appli doit être déployée - Ouvrir l'URL
avec "Ouvrir l'application"





Second exemple de pipeline avec Node.js

1 - Créer un dépôt sur
GitLab

2 - Créer le fichier
.gitlab-ci.yml à la racine
du dépôt

```
image: node:latest

stages:
  - build
  - test

cache:
  paths:
    - node_modules/

install_dependencies:
  stage: build
  script:
    - npm install
  artifacts:
    paths:
      - node_modules/

testing_testing:
  stage: test
  script: npm test
```

Explication du fichier yaml

Le fichier de configuration commence par déclarer une image docker qui permet de spécifier une certaine version de NodeJS que vous souhaitez utiliser pendant la construction.

```
image: node:latest
```

Ensuite, nous définissons les différents processus d'intégration continue qu'il exécutera.

```
stages:
```

```
- build
```

```
- test
```

Maintenant que nous avons défini les étapes, la configuration comprend un cache qui spécifie les fichiers qui doivent être enregistrés pour être utilisés ultérieurement entre les exécutions ou les étapes.

```
cache:
```

```
  paths:
```

```
    - node_modules/
```

Ensuite **install_dependencies**, permet de poser l'interaction entre les étapes, nous extrayons cette étape pour exécuter sa propre étape. Typiquement, en cours d'exécution npm install peut être combiné avec les prochaines étapes de test.

```
install_dependencies:  
  stage: build  
  script:  
    - npm install  
  artifacts:  
    paths:  
      - node_modules/
```


Dernièrement, **testing_testing** (vous décidez du nom) déclare la commande qui exécutera la combinaison de test, puisqu'il s'agit de la dernière étape, elle accédera à ce qui est produit par le stage build, qui sont les dépendances du projet dans notre cas.

```
testing_testing:  
  stage: test  
  script: npm test
```

Installation de GitLab Runner

Étant donné que notre référentiel comprend un fichier `.gitlab-ci.yml`, tout nouveau commit déclenchera une nouvelle exécution de CI. Si aucun runner n'est disponible, le statut CI sera définie sur « en attente ».

Comme mentionné, dans GitLab, les Runners exécutent les tâches que vous définissez dans `.gitlab-ci.yml`.

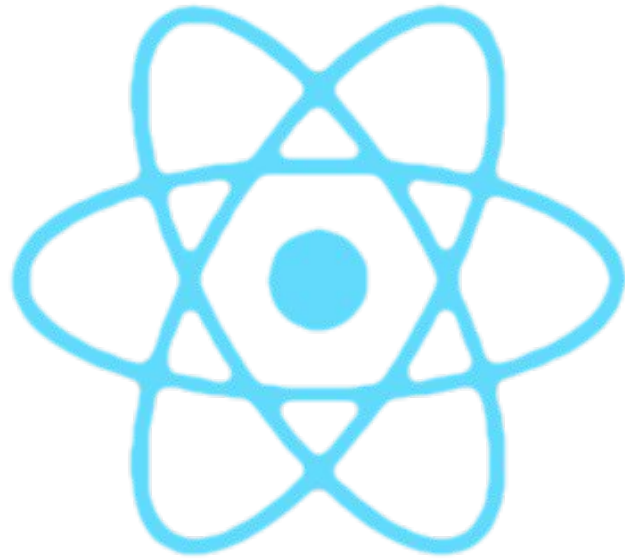
<https://docs.gitlab.com/runner/install/>

Enregistrer GitLab Runner

<https://docs.gitlab.com/runner/register/index.html>

```
$ gitlab-runner install  
$ gitlab-runner start
```

```
$ gitlab-runner status
```



React

Test d'un yaml avec une app react vierge

1 - Créer un nouveau repo GitLab

2 - En local

```
npx create-react-app my-app  
cd my-app  
npm start
```

3 - Push sur le repo distant

4 - GitLab → Setting > General > Visibility > Pages → Activer pour tout le monde

```
image: node:10.16.3 # change to match your project's node version
```

```
cache:
```

```
  paths:
```

```
    - node_modules/
```

```
before_script:
```

```
  - rm -rf build
```

```
  - CI=false npm install
```

```
pages:
```

```
  stage: deploy
```

```
  script:
```

```
    - CI=false npm run build
```

```
    - rm -rf public
```

```
    - cp build/index.html build/404.html
```

```
    - mv build public
```

```
artifacts:
```

```
  paths:
```

```
    - public
```

```
only:
```

```
  - master
```

Test d'un yaml avec une app react vierge

5 - Juste après aller voir dans CI/CD --> Pipelines

6 - Mettre à jour le package.json avec :

```
"homepage": "https://{your-username}.gitlab.io/{project}"
```

puis pusher la modif

7 - Vérifier le lien du site sur Setting → Pages

Exemples de pipelines CI/CD GitLab à pratiquer

<https://docs.gitlab.com/ee/ci/examples/>

Exercice :

Mettre en place un fichier `.gitlab-ci.yml` permettant de checker

1 - si le build d'une appli Node est cohérent

2 - la qualité du code

3 - les failles de sécurité