





Gitlab une plateforme DevOps complète

David THIBAU – 2023

david.thibau@gmail.com



Agenda

Introduction

- Agilité et DevOps
- CI et CD
- Architecture des systèmes
- La plateforme Gitlab

Pilotage de projet

- Projets et membres
- Issues et Milestones

Gestion des sources et collaboration

- L'unique source de vérité
- Gitlabflow et MergeRequest
- Déclinaisons

Pipelines CI/CD

- Introduction
- Jobs et Runners
- UI pipelines
- Définition de Pipelines

Phases de build

- Construction et tests développeur
- Analyses statiques
- Dépôts d'artefacts
- Déploiement et release
- Tests post-déploiements
- Gestion de l'infrastructure



Introduction

Agilité et DevOps

Pipelines CI/CD

Architecture et Infrastructure

La plateforme Gitlab



L'agilité

- Le terme agile (regroupant de nombreuses méthodes) est consacré par le manifeste Agile : <http://agilemanifesto.org/> 2001
 - Personnes et interactions plutôt que processus et outils
 - Logiciel fonctionnel plutôt que documentation complète
 - Collaboration avec le client plutôt que négociation de contrat
 - Réagir au changement plutôt que suivre un plan



Méthodes agiles

2 facteurs communs à toutes les méthodes agiles :

- la mise en place de pratiques **itératives**,
 - des projets plus **petits**
=> des équipes de dév. de plus en plus petites, des périmètres fonctionnels limités
- C'est ce l'on retrouve dans *RUP, XP Programming, Scrum, Safe, Spotify, etc..*



Contraintes sur la fréquence des déploiements

L'agilité suppose d'augmenter la fréquence des déploiements dans les différents environnements : intégration, recette, production afin :

- De fiabiliser les processus de déploiement
- De mieux piloter le projet
- De prendre en compte rapidement les retours utilisateurs

Problème : Avant DevOps, l'organisation des services informatiques ne facilitait pas les déploiements



Le constat DevOps

Les différents objectifs donnés à des équipes qui se parlent peu créent des tensions et des dysfonctionnements dans le processus de mise en production d'un logiciel.

=> Pour l'équipe Ops, l'équipe de développement devient responsable des problèmes de qualité du code et des incidents survenus en production.

=> L'équipe Dev blâme son alter ego Ops pour sa lenteur, les retards et leur méconnaissance des livrables qu'elle manipule



Approche *DevOps*

DevOps vise l'alignement des équipes par la réunion des "Dev engineers" et des "Ops engineers" .

Cela impose :

- la réunion des équipes
- la montée en compétence des différents profils.
=> Arrivée des profils *full-stack*

Une équipe *DevOps* est une équipe qui regroupe toutes les compétences nécessaires à un projet¹ et qui est complètement indépendante



Pratiques *DevOps* (1)

- Un déploiement régulier/continu des applications dans les différents environnements.
La répétition fiabilise le processus ;
- Un décalage des tests "vers la gauche".
Autrement dit de tester au plus tôt ;
- Une pratique des tests dans un environnement similaire à celui de production ;
- Une intégration continue incluant des "tests continus" (à chaque commit);



Pratiques *DevOps* (2)

- Une boucle d'amélioration courte
i.e. un feed-back rapide des utilisateurs ;
- Une surveillance étroite de l'exploitation et de la qualité de production actualisée par des métriques et indicateurs "clé".
- La production de métriques reflétant la qualité du projet
- Une unique source de vérité : Le dépôt de source inclut le code de build, de test, d'infrastructure et ... le code source



« As Code »

Les outils *DevOps* permettent d'automatiser/exécuter toutes les tâches nécessaires à la construction d'un logiciel de qualité (build, test, provisionnement) à partir de l'unique point central de vérité

On parle alors de *Build As Code*,
Infrastructure As Code, *Pipeline As Code*,
Load Test As Code, ...



Objectif ultime

- Déployer souvent et rapidement
- Automatisation complète
- Zero-downtime des services
- Possibilité d'effectuer des roll-backs
- Fiabilité constante de tous les environnements
- Possibilité de scaler sans effort
- Créer des systèmes résilients, capable de se reprendre en cas de défaillance ou erreurs



Introduction

Agilité et DevOps

Pipelines CI/CD

Architecture et Infrastructure

La plateforme Gitlab



En continu

A chaque ajout de valeur dans le dépôt de source (*push*), l'intégralité des tâches nécessaires à la mise en service d'un logiciel (intégration, tests, déploiement) sont essayées.

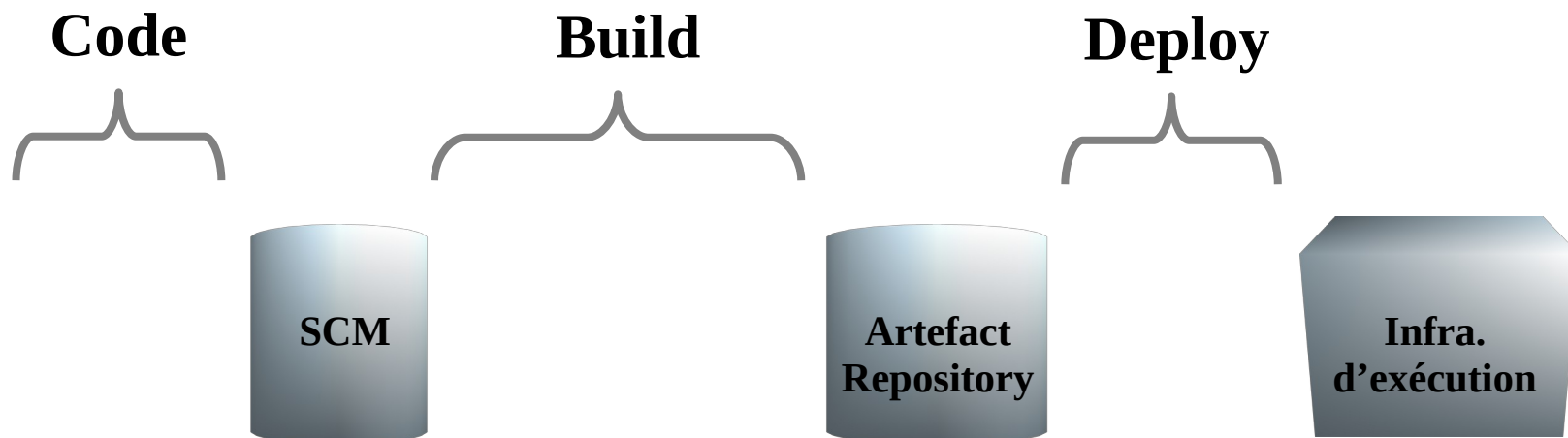
En fonction de leurs succès, l'application est déployée dans les différents environnements (intégration, staging, production)



Cycle de vie du code Build / Release / Run

La pipeline suit le cycle de vie du code.

- 1) Le code est testé localement puis poussé dans le **dépôt de source**
- 2) Le build construit l'artefact et si le build est concluant stocke une release dans un **dépôt d'artefact**
- 3) L'outil de déploiement accède aux différentes release et les déploie sur l'**infra d'exécution**





Build is tests !

La construction de l'application consiste principalement à :

- Packager le code source dans un format exécutable, déployable
- Effectuer toutes les vérifications automatiques permettant d'avoir confiance dans l'artefact généré et autoriser son déploiement
(Tests Unitaires/Intégration, Fonctionnel, Performance, Sécurité, Licenses, ...)



Pipelines

Les étapes de construction automatisées sont séquencées dans une **pipeline**.

- Une étape est exécutée seulement si les étapes précédentes ont réussi.
- Les plate-formes d'intégration/déploiement continu ont pour rôle de démarrer et observer l'exécution de ces pipelines



Distinction CI/CD





Outil de communication

La Plateforme a pour vocation de publier les résultats des builds :

- Nombre de tests exécutés, succès/échecs
- Couverture des tests
- Complexité, Vulnérabilités du code source
- Performance : Temps de réponse/débit
- Documentation, Release Notes
- ...

Les métriques sont visibles de tous en toute transparence

- => Confiance dans ce qui a été produit
- => Motivation pour s'améliorer



Phases de la mise en place

La mise en place de pipeline de CI/CD passe généralement par plusieurs phases :

1. Mise à disposition d'une PIC
2. Automatisation tests unitaires et d'intégration
3. Déploiement dans un environnement d'intégration (review app)
4. Mise en place d'analyse de code, de tests fonctionnels, performance, d'acceptation automatisés, Collecte des métriques
5. Renforcement des tests, Release automatique, déploiement en QA
6. Renforcement des tests d'acceptation, Validation de déploiements
7. Totale confiance dans les tests permet le Déploiement continu



Introduction

Agilité et DevOps
Pipelines CI/CD

Architecture et Infrastructure

La plateforme Gitlab



Introduction

Avec DevOps une nouvelle architecture de systèmes visant à améliorer la rapidité des déploiements des retours utilisateur est apparu : les « **micro-services** »

C'est le même objectif que l'approche *DevOps* : « *Déployer plus souvent* »

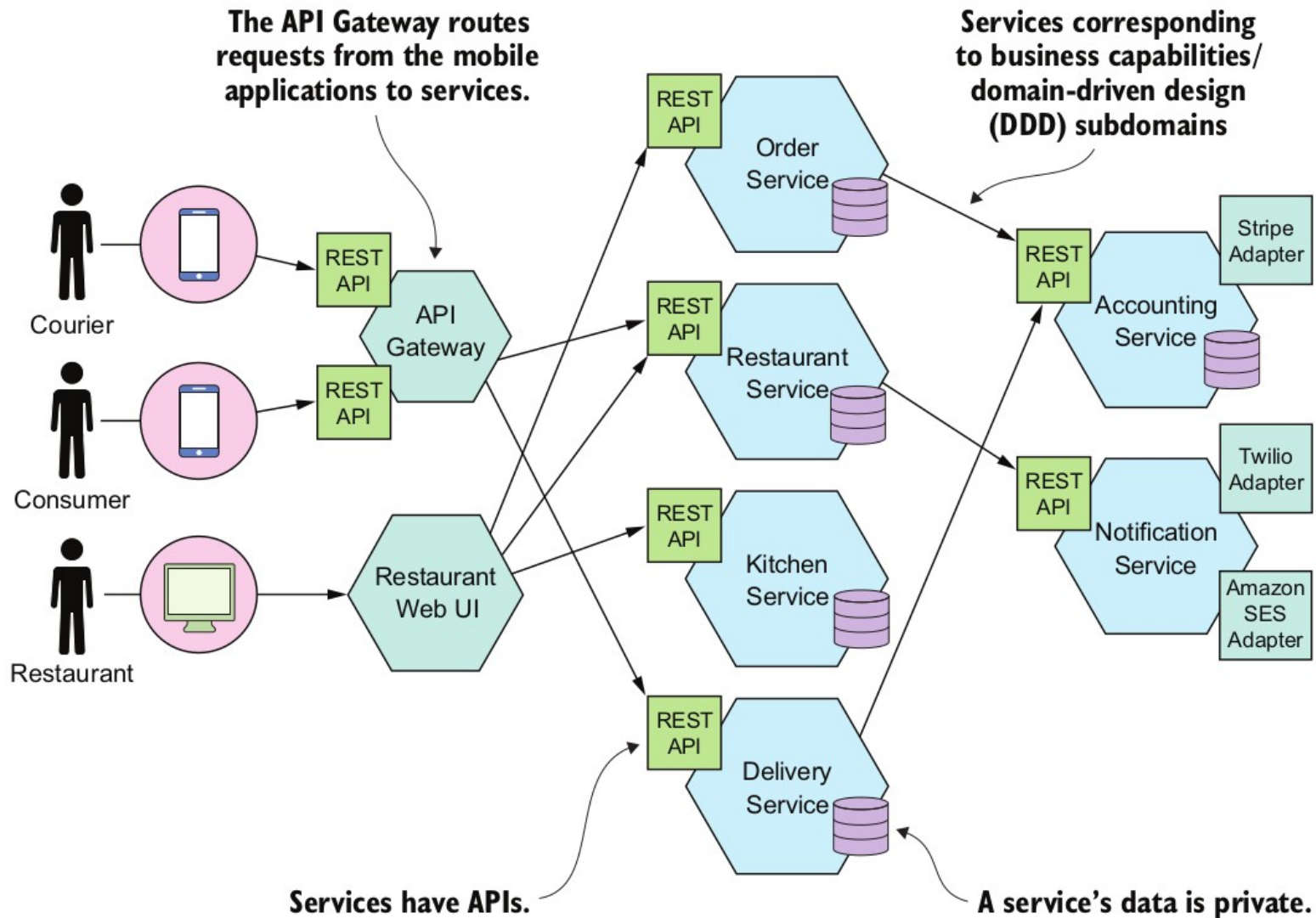


Architecture

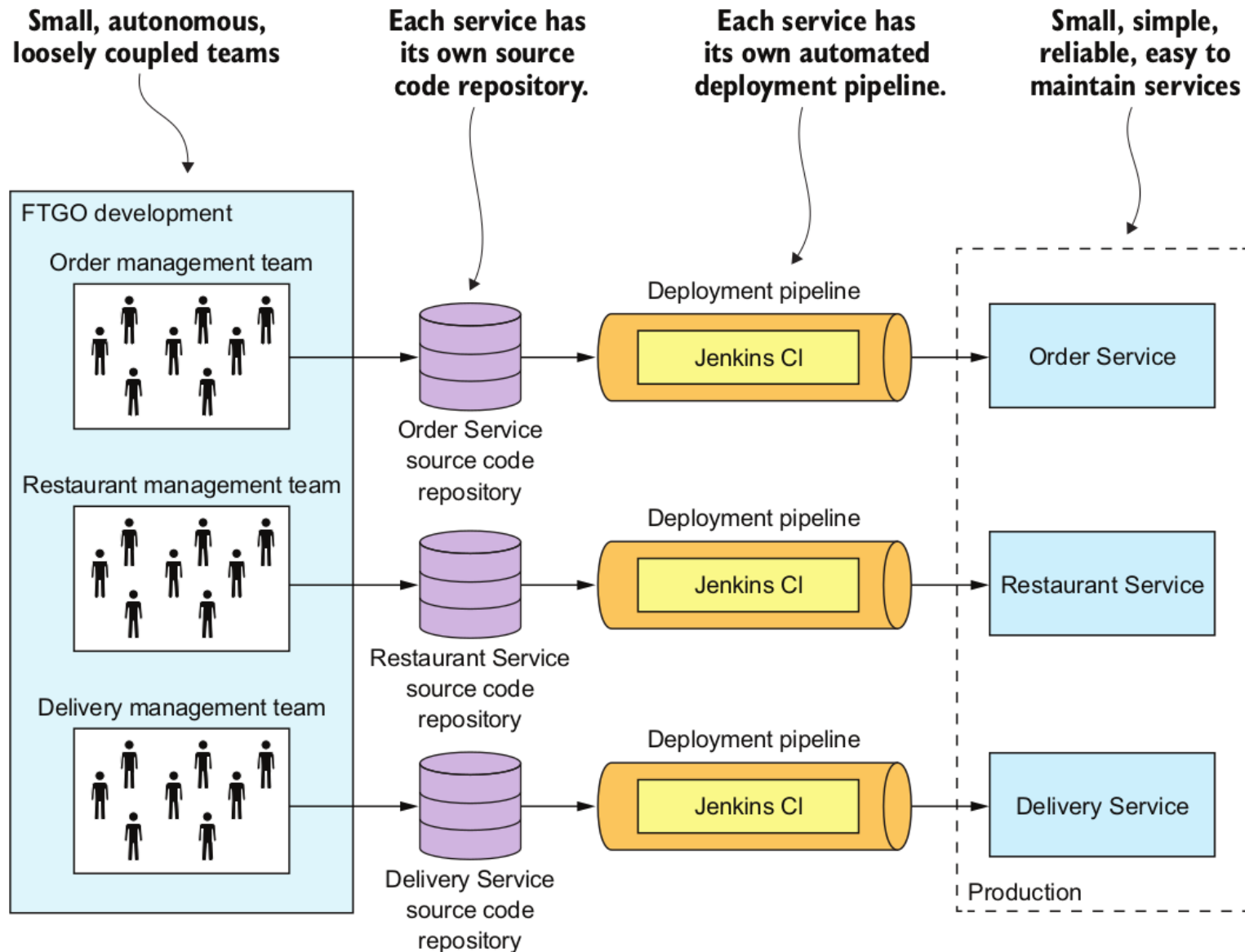
Une architecture micro-services implique la décomposition des applications en petits services

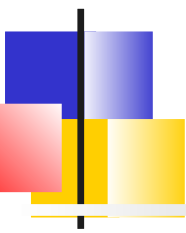
- faiblement couplés
- ayant une seule responsabilité métier
- Développés par des équipes full-stack indépendantes.

Une architecture micro-service



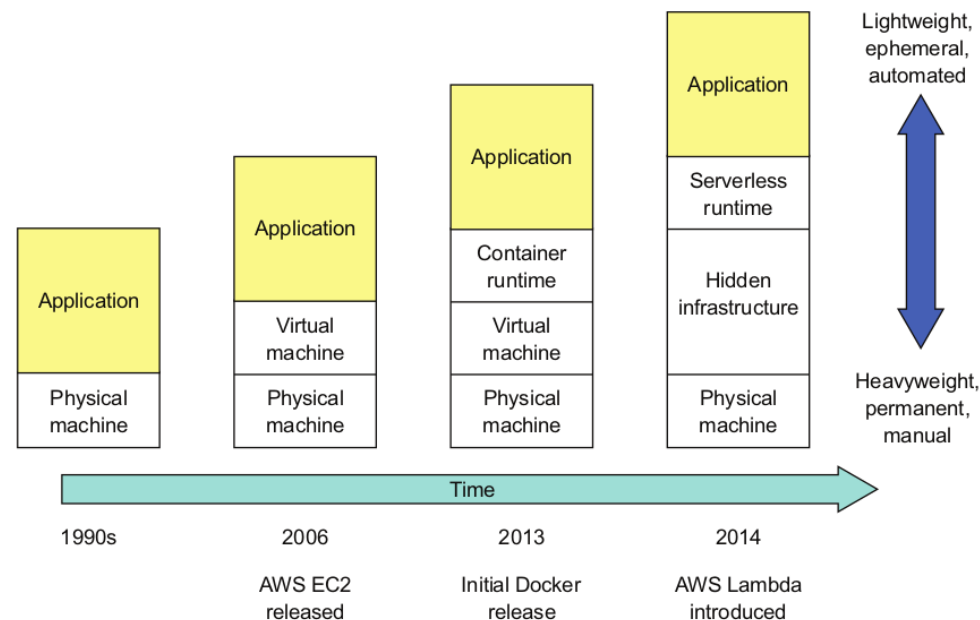
Organisation DevOps





Infrastructure de déploiement

Même si plusieurs alternatives peuvent être envisagées, l'utilisation des technologies de container et d'orchestrateur de container sont plus adaptées





Service comme Container

Deploy a service as a container

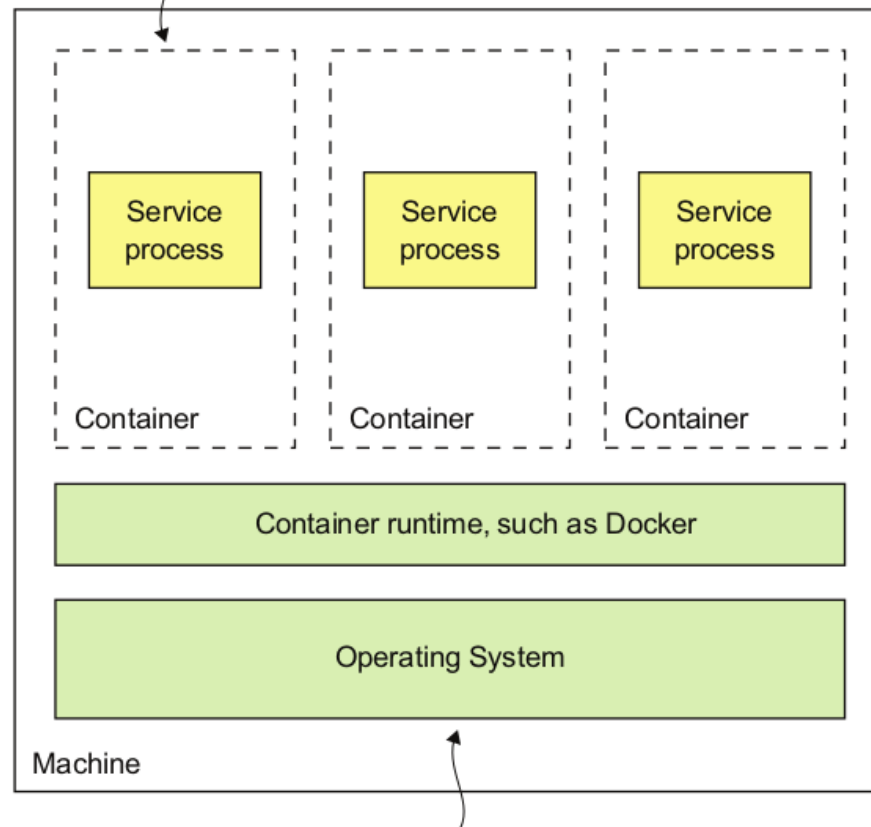
Pattern¹ : Déployé les services packagés comme des images de conteneur. Chaque service est un conteneur

Le packaging en image fait partie de la pipeline de déploiement

1. <http://microservices.io/patterns/deployment/service-per-container.html>

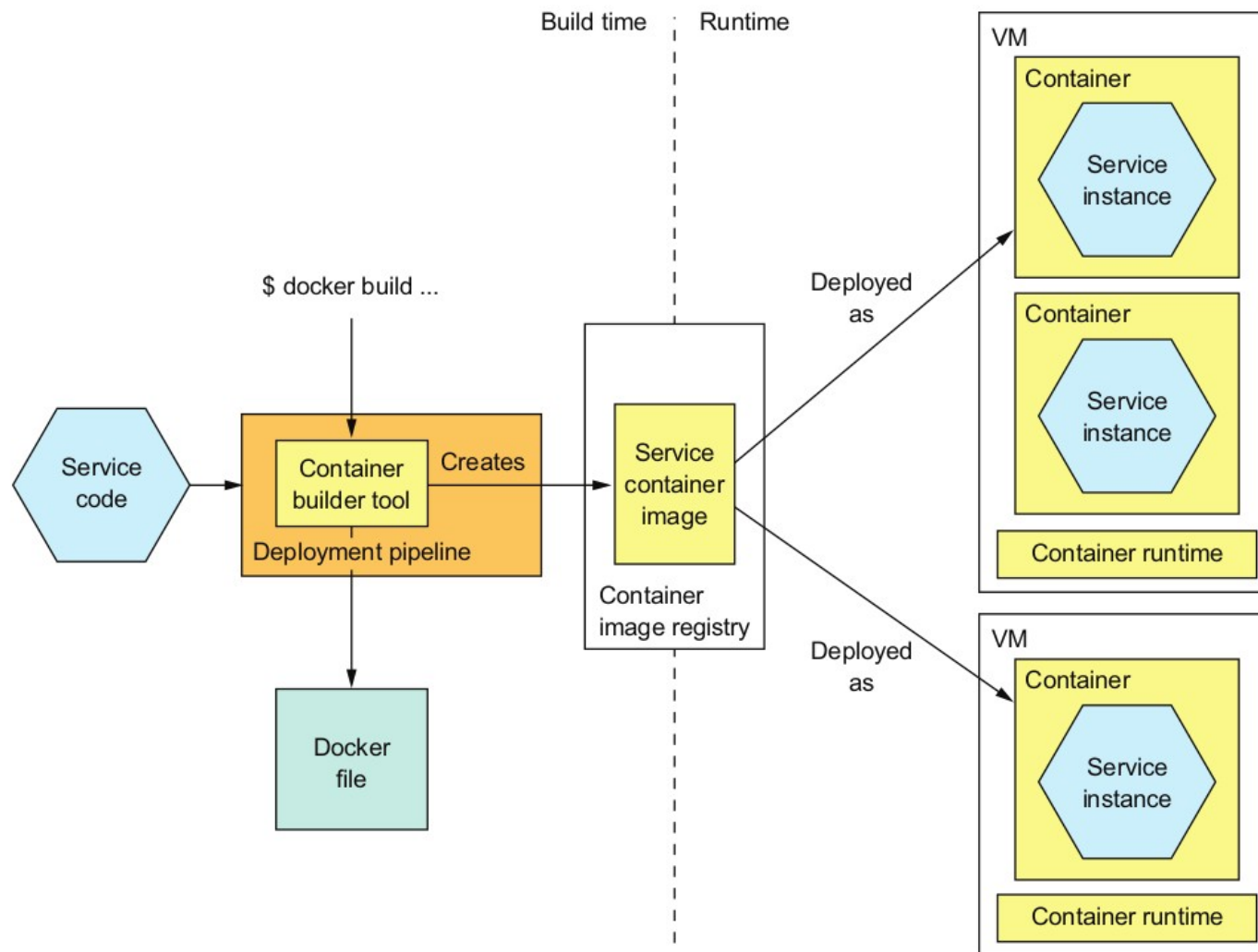
Exécution

Each container is a sandbox that isolates the processes.



Shared by all of the containers

Déploiement





Bénéfices / Inconvénients

Bénéfices

Encapsulation de la pile technologique. Déploiements immuables

Les instances de service sont isolées.

Les ressources des instances de service sont limitées.

Inconvénients

Équipe DevOps responsable de l'administration des images du conteneur. (Patches de l'OS par exemple)

Administrer l'infrastructure du conteneur-runtime et éventuellement des VMs associés



Apports des déploiement immuables

Facilite le provisionnement

Permet facilement des déploiements
blue-green sans interruption de service

Facilite le roll-back

Permet le canary deployment (plusieurs
versions déployées simultanément)

Multiplie les environnements (un
environnement par branche)



Introduction

Agilité et DevOps

Pipelines CI/CD

Architecture et Infrastructure

La plateforme Gitlab



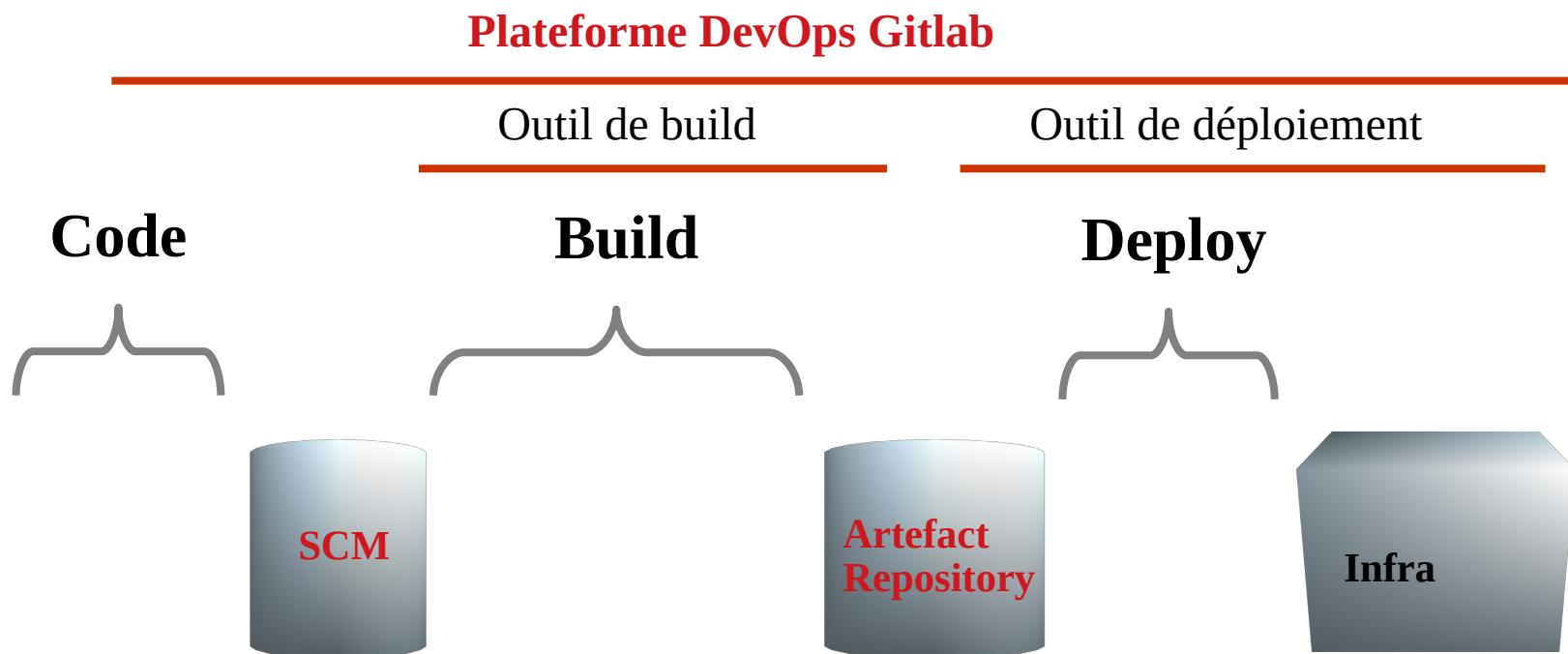
Introduction

Gitlab se définit comme une plateforme de DevOps complète qui inclut :

- La gestion des codes sources
- La gestion de projet agile
- Les pipelines de CI/CD
- La gestion des dépôts artefacts
- La gestion des environnements de déploiement
- La mise à disposition des bonnes pratiques DevOps



Gitlab et Build/Release/Run





Intégration GitLab

Gitlab peut également s'intégrer avec d'autres outils tierces qui pourront prendre une part du Devops

- Annuaire utilisateur et sécurité (LDAP, OpenID, ...)
- Pilotage de projet (Jira, ...)
- Communication projet (Slack, Gmail, ...)
- Dépôt d'artefact (Nexus, Artifactory, Docker registry, ...)
- Outils d'analyse (Sonarqube, ...)
- Exécution de pipeline (Jenkins, ...)
- Infrastructure (Terraform, Kubernetes, Cloud)
- ...



Installations

Gitlab s'installe sous Linux. Différentes façons :

- **Omnibus Gitlab** : Packages pour différentes distributions de Linux
- **GitLab Helm chart** : Version Cloud, installation sous Kubernetes
- Images **Docker**
- A partir des **sources**

Également disponible en ligne : *gitlab.com*



Community vs Enterprise

Le même cœur, l' *enterprise edition* ajoute du code propriétaire.

Le code propriétaire peut devenir gratuit au fur et à mesure des évolutions

Les versions payantes apportent généralement :

- Des fonctionnalités innovantes
- Des fonctionnalités avancées (Scanner de sécurité par exemple)
- Des fonctionnalités transverses au projet
- Des facilités d'intégration avec des outils
- Une installation en HA
- Du support 24h/24



Interface utilisateur

2 interfaces utilisateurs sont disponibles :

- Administrateur : Permet de configurer la plateforme, de gérer les utilisateurs, de configurer les runners disponibles et de configurer de façon transverse certains aspects des projets
- Utilisateur :
 - Permet de gérer son compte (infos, créden-tiels, notifications, préférences)
 - Permet d'accéder à ses projets



Pilotage de projet DevOps

Projets et Membres Gitlab

Issues, milestones et tableaux de bord



Projets

Un projet *Gitlab* a vocation à être associé à un dépôt de source *Git*

Par défaut, tous les utilisateurs *Gitlab* peuvent créer un projet

3 visibilité sont possibles pour un projet :

- **Public** : Le projet peut être cloné sans authentification. Tout utilisateur a la permission *Guest*
- **Interne** : Peut être cloné par tout utilisateur authentifié. Tout utilisateur a la permission *Guest*
- **Privé** : Ne peut être cloné et visible seulement par ses membres



Fonctionnalités

Un projet apporte plusieurs fonctionnalités :

- **Suivi d'issues** : Collaboration sur le travail planifié, milestones,
- **Gestion de dépôts** : Organisation des branches, Merge request, accès au source, Web IDE
- **Pipelines de CI/CD**
- **Autres** : Wiki, Tableaux de bords, Gestion de release et environnements, dépôts Maven ou NPM, registres docker,

Menus

Projects : Informations sur les commits, les branches, l'activité, les releases, tdb sur la productivité

Repository : Navigateur de fichiers, Commits, branches, tags, historique, comparaison, statistiques sur les fichiers du projet

Issues : Gestion des issues, tableau de bord Kanban

Merge requests : Travaux en cours

CI/CD : Historique d'exécution des pipelines

Operations : Gestion des environnements de déploiement

Packages : Accès au registre de conteneur

Wiki : Documentation annexe

Snippets : Bouts de code

Settings : Configuration projet, Visibilité, Merge Request, Membres, pipeline, intégration avec d'autres outils



Membres

Les utilisateurs peuvent être affectés à des projets, ils en deviennent **membres**

Un membre a un rôle qui lui donne des permissions sur le projet :

- **Guest** : Créer un ticket
- **Reporter** : Obtenir le code source
- **Developer** : Push/Merge/Delete sur les branches non protégée, Merge request sur les autres branches
- **Maintainer** : Administration de l'équipe, Gestion des branches protégés ou non, Tags, Ajouts de clés SSH
- **Owner** : Créateur du projet, a le droit de le supprimer



Groupes

Afin de faciliter la gestion des membres et de leurs permissions, il est possible de définir des **groupes de projets**.

- Pour ces groupes, il est possible de définir des membres
=> Les membres d'un groupe ont alors accès à tous les projets du groupe

Les groupes peuvent être hiérarchiques

Attention : Il est dangereux de déplacer un projet existant dans un autre groupe

Settings -> General -> Advanced -> Transfer project -> Select a new namespace



Configuration Utilisateur

Dans la partie *Settings* d'un utilisateur, en dehors des informations personnelles, on retrouve :

- La configuration des notifications par projet
- La gestion des clés SSH facilitant l'authentification
- La gestion des clés GPG permettant de signer des tags
- Les préférences (en particulier la langue)



Mise en place clés ssh

La mise en place des clés *ssh* permet de pouvoir interagir avec Gitlab sans avoir à fournir de mot de passe.

2 étapes :

- Créer une paire de clé privé/publique
- Fournir la clé publique à Gitlab via l'interface web



Mise en place

- Environnement Linux :

```
ssh-keygen -t ed25519 -C "email@example.com"
```

Ou

```
ssh-keygen -o -t rsa -b 4096 -C "email@example.com"
```

- Copier le contenu de la clé publique (*.pub) dans l'interface Gitlab
- Tester avec :

```
ssh -T git@gitlab.com
```




Outils de pilotage

Scopes des outils
Issues et milestones



Issues et Milestones

Les outils disponibles sont suffisamment configurable afin de s'adapter aux particularités de chaque méthodologie agile.

Le concept principal est généralement une **issue** qui peut représenter :

- Une user story
- Un demande d'évolution
- Une déclaration de bug
- Un idée d'amélioration

Les issues sont généralement affectés à des **milestones** qui peuvent représenter :

- Une release
- Un sprint
- Une date de livraison
-



Visualisation des issues

Les issues peuvent être visualisées via :

- Une **liste**. Elle affiche toutes les issues du projet ou de plusieurs projets. On peut les filtrer ou faire des actions par lots (bulk)
- Le **tableau de bord Kanban** qui affiche des colonnes en fonction des labels (par défaut statut de l'issue) ou des responsables. Les workflows sont customisable via les labels
- **Epic** : Vision transversale aux projets des issues partageant un thème, un milestone,



Labels

Les labels jouent un rôle très important dans Gitlab

Ils permettent :

- De catégoriser les issues à l'aide de couleurs et de titres descriptifs comme *bug, feature request, documents*.
- De filtrer les listes d'issues
- De créer des tableaux de bord



Usage des Labels

Gitlab propose des labels par défaut mais il est possible de configurer ses propres labels

Un label peut être défini au niveau

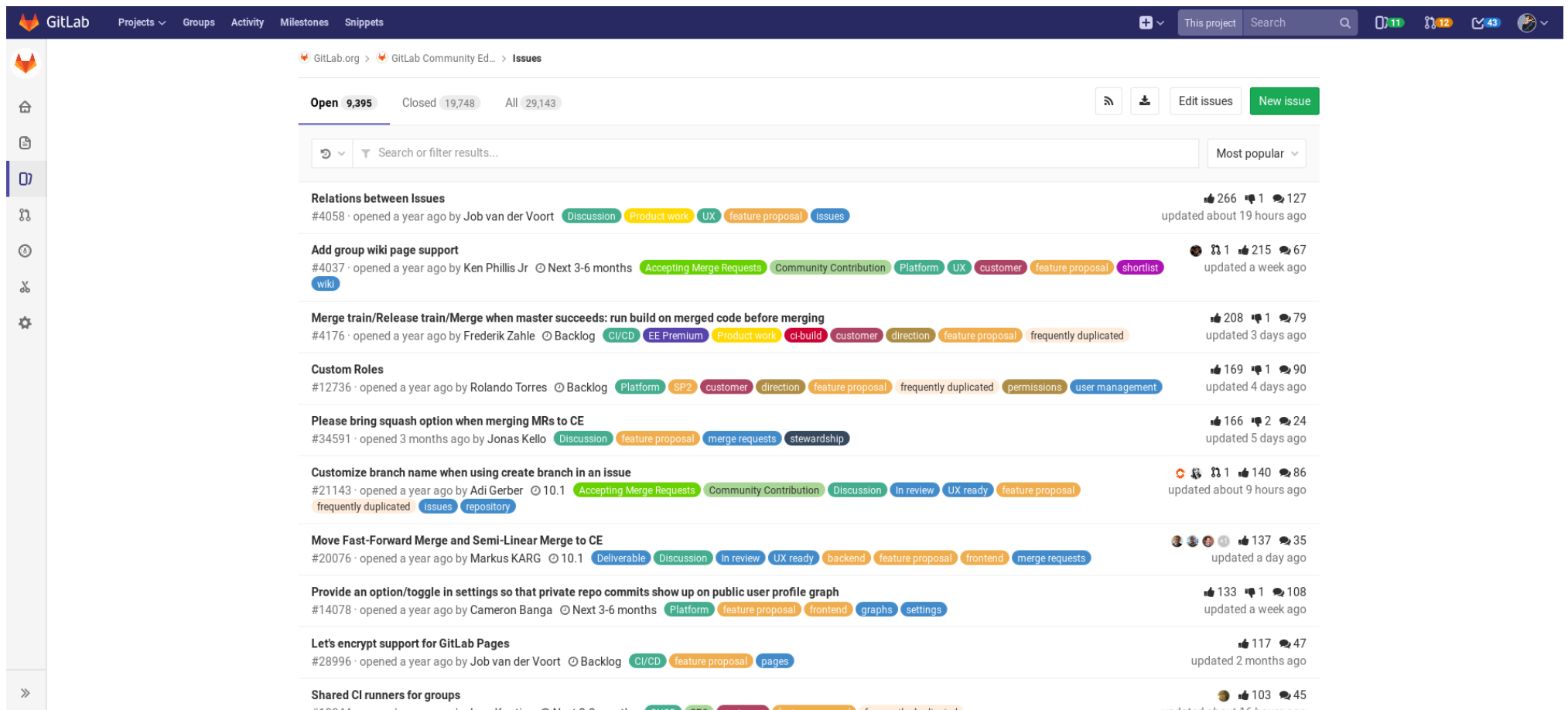
- Groupe : ***Group information > Labels.***
- ou projet : ***Project information > Labels.***

Plusieurs labels peuvent être associés à la même issue

On peut donc créer des labels permettant :

- De typer (Bug, Idée, RFC, User Story, ...)
- Indiquer le domaine concerné (Front-end, Back-end, CI/CD,...)
- Indiquer un statut (*Review, Duplicate, ...*)
- ...

Exemple : Liste d'issues avec labels



The screenshot displays the GitLab web interface for the 'GitLab Community Edition' project. The top navigation bar includes links for Projects, Groups, Activity, Milestones, and Snippets. The main header shows the project name and a search bar. The left sidebar contains navigation icons for Home, Issues, and other project features.

The 'Issues' section is active, showing a list of issues. The top of the list includes filters for 'Open' (9,395), 'Closed' (19,748), and 'All' (29,143). A search bar and a 'Most popular' dropdown are also present.

The list of issues includes the following entries:

- Relations between Issues** (#4058) - opened a year ago by Job van der Voort. Labels: Discussion, Product work, UX, feature proposal, issues. 266 likes, 1 comment, 127 replies. Updated about 19 hours ago.
- Add group wiki page support** (#4037) - opened a year ago by Ken Phillis Jr. Labels: Accepting Merge Requests, Community Contribution, Platform, UX, customer, feature proposal, shortlist, wiki. 1 like, 1 comment, 215 replies, 67 mentions. Updated a week ago.
- Merge train/Release train/Merge when master succeeds: run build on merged code before merging** (#4176) - opened a year ago by Frederik Zahle. Labels: Backlog, CI/CD, EE Premium, Product work, ci-build, customer, direction, feature proposal, frequently duplicated. 208 likes, 1 comment, 79 replies. Updated 3 days ago.
- Custom Roles** (#12736) - opened a year ago by Rolando Torres. Labels: Backlog, Platform, SP2, customer, direction, feature proposal, frequently duplicated, permissions, user management. 169 likes, 1 comment, 90 replies. Updated 4 days ago.
- Please bring squash option when merging MRs to CE** (#34591) - opened 3 months ago by Jonas Kello. Labels: Discussion, feature proposal, merge requests, stewardship. 166 likes, 2 comments, 24 replies. Updated 5 days ago.
- Customize branch name when using create branch in an issue** (#21143) - opened a year ago by Adi Gerber. Labels: Accepting Merge Requests, Community Contribution, Discussion, In review, UX ready, feature proposal, frequently duplicated, issues, repository. 1 like, 1 comment, 140 replies, 86 mentions. Updated about 9 hours ago.
- Move Fast-Forward Merge and Semi-Linear Merge to CE** (#20076) - opened a year ago by Markus KARG. Labels: Deliverable, Discussion, In review, UX ready, backend, feature proposal, frontend, merge requests. 137 likes, 1 comment, 35 replies. Updated a day ago.
- Provide an option/toggle in settings so that private repo commits show up on public user profile graph** (#14078) - opened a year ago by Cameron Banga. Labels: Next 3-6 months, Platform, feature proposal, frontend, graphs, settings. 133 likes, 1 comment, 108 replies. Updated a week ago.
- Let's encrypt support for GitLab Pages** (#28996) - opened a year ago by Job van der Voort. Labels: Backlog, CI/CD, feature proposal, pages. 117 likes, 47 replies. Updated 2 months ago.
- Shared CI runners for groups** (#10244) - opened a year ago by Jean Kertier. Labels: Next 3-6 months, CI/CD, SP2, customer, feature proposal, frequently duplicated. 103 likes, 45 replies. Updated about 16 hours ago.



Tableaux de bords

Gitlab propose des **boards** qui permettent de visualiser les issues à la façon des tableaux de bord agiles (Kanban, Scrum)

Ils utilisent les issues et les labels.

Les issues apparaissent sous forme de fiches dans des listes verticales, organisées selon les étiquettes, les jalons ou les responsables

Il est possible de définir plusieurs tableaux de bord dans un projet

56

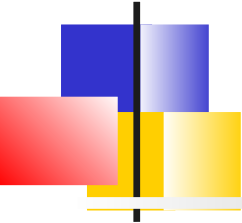


Scoped Labels

Les **scoped labels** ont un format *clé :: valeur*.

- A un instant t , une issue ne peut pas avoir plusieurs labels de la même clé.
- Cela peut permettre de définir des workflows et sont utilisés dans les tableaux de bord.

Exemple de scoped labels
workflow::development,
workflow::review
workflow::deployed



Champs d'une issue à la création

Une issue comporte de nombreux champs qui sont pour la plupart optionnels :

- Titre : On peut forcer des gabarits
- Types : Issue ou Incident
- Description : Rich text
- Assignee : Les personnes impliquées
- Due date
- Milestone
- Labels

Une issue peut être créée par tous les membres du projet et même par les utilisateurs si on active la fonctionnalité

ServiceDesk



Collaboration autour de l'issue

De nombreuses fonctionnalités de collaboration sont proposées autour de l'issue :

- Threads de discussion et notifications/alertes
- Workflows (Changement de labels/statut)
- Intégration DevOps :
 - Association aux modifications de code,
 - Aux pull/merge requests
 - A la revue de code
 - Aux pipelines, aux résultats des tests automatisés
 - Aux environnements de test



Données associées à une *issue*

Contenu :

- Titre
- Description et tâches
- Commentaires et activité

Membres

- Auteur
- Responsables

Etat

- Status (ouvert/fermé)
- Confidentialité
- Tâches (terminé ou en suspens)

Planning et suivi

- Milestone
- Date de livraison
- Poids
- Suivi du temps
- Labels
- Votes
- Reaction emoji
- Issues liées
- Epic (collection d'issues) affectée
- Identifiant et URL



Actions sur une issue

1. Création, Fermeture, Edition des champs de base
2. Ajouter à sa Todo List, la marquer comme terminé
3. Responsable(s) de l'issue, peut être changé à tout moment
4. Affecter une issue à un milestone
5. Temps estimé, temps passé
6. Date de livraison, peut être changée à tout moment
7. Labels.
8. Poids. Indicateur sur l'effort nécessaire associé à l'issue
9. Participants. Indiqués dans la description ou qui ont participé à la discussion
10. Notifications. Permet de s'abonner/désabonner
11. Référence. Permet de copier l'URL d'accès
12. Titre et description (markup)
13. Mentions. Met en surbrillance pour la repérer facilement
14. Merge requests associés
15. emoji
16. Thread. Commentaires organisés en threads



Autres fonctionnalités

Issues liées : Permet d'associer une issue à une autre (Travail préliminaire, contexte, dépendance, doublon)

Crosslinking : Liens vers des objets référençant l'issue. (Commit, Autre Issue ou Merge Request)

- Par exemple un commit
git commit -m "this is my commit message. Ref #xxx"

Fermeture automatique : Possibilité de fermer les issues automatiquement après un merge request

Gabarits : Créer des issues à partir de gabarits

Édition en mode bulk

Import/Export d'issues

API Issues



Epics

Lorsque des issues partagent un thème entre des projets, ils peuvent être gérés à l'aide d'epics.

Les cas d'usage des epics sont :

- Une fonctionnalité importante qui implique plusieurs discussions sur différents issues de différents projets d'un même groupe.
- Pour connaître l'état d'avancement d'un groupe d'issues
- Discuter et collaborer sur des idées de fonctionnalités et leur portée à un niveau élevé.



Gestion des sources et collaboration

L'unique source de vérité
Gtilabflow et MergeRequest
Déclinaisons



Introduction

Dans l'approche DevOps, le SCM est **l'unique source de vérité**

Tout ce qui est nécessaire au projet y est stocké et versionné :

- Code source mais aussi
- Fichiers et outils de build, Pipelines de CI/CD, Tests automatisés, scripts de provisionnement et de déploiements
- Docs

=> A partir d'un clone du dépôt, un nouveau participant au projet doit être capable de faire tout seul :

- Build
- Deploy



SCM

Un **SCM** (*Source Control Management*) est un système qui enregistre les changements faits sur une structure de fichiers afin de pouvoir revenir à une version antérieure

Le système permet :

- De restaurer le projet à un instant t
- Visualiser tous les changements effectués leurs auteurs et leurs commentaires associés
- Le développement concurrent (branche)



Patch

Chaque modification du code source enregistrée dans le SCM peut être visualisée sous forme de ***patches***

Un patch indique les blocs de lignes d'un fichier ayant été modifiés

```
@ -35,7 +35,7 @@ stage('Parallel Stage') {
  stage('Déploiement artefact') {
    steps {
      echo 'Deploying..'
-      sh './mvnw -Pprod clean deploy'
+      sh './mvnw --settings settings.xml -Pprod clean deploy'
      dir('target/') {
        stash includes: '*.jar', name: 'service'
      }
    }
  }
}
```



Git:

Commit, Branches et Tag

Git stocke l'historique complet des sources du projet

- Les **commits** (Ids) permettent d'isoler chaque modification apportée par les développeurs, les historiser et les documenter
- Les **branches** permettent d'effectuer des travaux en isolation, lorsque les travaux sont terminés, ils sont intégrés dans une branche plus stable.
- Les **tags** sont à priori immuables et permettent de fixer un instantané du projet. Ils correspondent en général à des releases déployable en production



Principales opérations Git

clone : Recopie intégrale du dépôt

checkout : Extraction d'une révision particulière

commit : Enregistrement de modifications de source

push/pull : Pousser/récupérer des modifications d'un dépôt distant

log : Accès à l'historique

merge, rebase : Intégration des modifications d'une branche dans une autre



Particularités *Gitlab*

On peut interagir avec les dépôts GitLab via **l'UI** ou en **ligne de commande**.

GitLab supporte des langages de **markup** pour les fichiers du dépôt.
Utilisé principalement pour la documentation

Lorsqu'un fichier **README** ou index est présent, son contenu est immédiatement rendu (sans ouverture du fichier)

L'UI donne la possibilité de **télécharger** le code source et les archives générées par les pipelines

Verrouillage de fichier : Empêcher qu'un autre fasse des modifications sur le fichier pour éviter des conflits.

Gitlab utilise des hooks qui peuvent afficher des messages d'assistance

Accès aux données via **API**. Exemple :

```
GET /projects/:id/repository/tree
```



Particularités du commit

- **Skip pipelines**: Si le mot-clé **[ci skip]** est présent dans le commit, la pipeline de GitLab ne s'exécute pas.
- **Cross-link issues/MR**: Si on mentionne une issue ou un MR dans un message de commit, ils seront affichés sur leur thread respectif.
- Il est possible via l'UI d'effectuer aisément un *cherry-pick* ou un *revert* d'un commit particulier
- Possibilité de signer les commits via GPG



Vues proposées

Settings → Contributors : Les contributeurs au code

Repository → Commits : Historique des commits

Repository → Branches/Tags : Gestion des branches et des tags

Repository → Graph : Vue graphique des commits et merge

Repository → Charts : Affiche les langages détectés par Gitlab et des statistiques sur des commits



Branche par défaut

A la création de projet, *GitLab* positionne **main** comme branche par défaut.

– Peut-être changé *Settings* → *Repository*.

C'est dans la branche par défaut que sont fusionnées les modifications relatives à une issue lors d'un *merge request*.

La branche par défaut est également une branche protégée : seul le mainteneur du projet peut la modifier



Création de branche

Plusieurs façons de **créer des branches** avec Gitlab :

- A partir d'une issue, la branche est donc documentée avec la collaboration sur l'issue
- A partir du menu **Repository → Branches**, de la même façon la branche sera fusionnée dans la branche par défaut



Branche protégée

Un branche peut être protégée

- Seul un membre avec au moins la permission *Maintainer* peut la créer
Project → Settings → Protected branches
- Elle définit des permissions :
 - Allow to Merge : Qui peut y fusionner une autre branche
 - Allow to Push : Qui peut y faire un push
- Seul le mainteneur peut supprimer une branche protégée

On peut utiliser des *wildcards* pour protéger plusieurs branches en même temps. *Ex :*

-stable, production/



Gestion des sources et collaboration

L'unique source de vérité
Gitlabflow et MergeRequest
Déclinaisons



Introduction

Les **Merge Request** sont la base de la collaboration sur Gitlab

Un MR est associée à une branche, elle est généralement associée à une issue

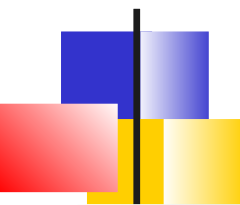
Une MR permet de :

- Empêcher une fusion trop précoce avec le statut *Draft*
- Comparer les changements entre 2 branches
- Revoir et discuter des modifications de code
- Visualiser les pipelines associées
- Accéder à l'appli. en fonctionnement (*Review Apps*)
- Faire un suivi du temps
- Effectuer la fusion avec une branche protégée




Cycle de vie d'un MR

- 1) Au démarrage d'un nouveau travail, le développeur crée une *merge request*.
La collaboration peut commencer et une feature branch est créée.
La *Merge Request* est préfixée par **Draft**
- 2) Lorsque la fonctionnalité est prête, le développeur le signale en enlevant le statut *Draft*
- 3) Le mainteneur peut alors faire de la revue de code, éventuellement accéder à la *review app*.
- 4) Le mainteneur a ensuite le choix entre :
 - effectuer la fusion dans la branche protégée, la MR obtient le statut **merged**
 - demander au développeur des améliorations,
 - abandonner la MR, elle obtient alors le statut **closed**



Vue projet

 GitLab Community Edition

Overview

Repository

Issues8,730



Merge Requests472

CI / CD

Wiki

Snippets


Settings

GitLab /  GitLab.org /  GitLab Community Edition

Merge Requests

Edit Merge RequestsNew merge request

Open472Merged11,188Closed1,969All13,629

 Search or filter results...

Last created

test MR

!13679 · opened 17 minutes ago by Mike Greiling

updated 14 minutes ago

Add docs for group issues page and group merge requests page

!13678 · opened 20 minutes ago by Victor Wu

updated 2 minutes ago

Docs update links guideline to inline links

!13677 · opened 36 minutes ago by Marcia Ramos 10.0 Documentation docs-update

updated 34 minutes ago

WIP: Clean up new dropdown styles 0 of 1 task completed

!13676 · opened 50 minutes ago by Winnie Hellmann 10.0 Deliverable UI polish frontend

updated 15 minutes ago

Greatly reduce test duration for git_access_spec

!13675 · opened 58 minutes ago by Robert Speicher 10.0 Edge backstage performance technical debt test

updated 20 minutes ago

Implement new system note icons 0 of 11 tasks completed

!13673 · opened about 3 hours ago by Bryce Johnson 10.0 Deliverable frontend

updated less than a minute ago

WIP: Prepare 9.5 RC6

!13672 · opened about 3 hours ago by Jose Ivan Vargas Lopez 9-5-stable Release

updated about an hour ago

Use Gitaly 0.33.0 0 of 11 tasks completed

!13671 · opened about 4 hours ago by Jacob Vosmaer (GitLab) 9.5 Gitaly Pick into Stable

updated about 4 hours ago

[WIP] Make the import take subgroups into account 0 of 9 tasks completed

!13670 · opened about 5 hours ago by Bob Van Landuyt 10.0 Platform

updated about 5 hours ago

79



Commentaires et discussions

Des **commentaires** peuvent être associés aux MR

- Soit au niveau général
- Soit au niveau d'un commit particulier

Un commentaire peut être transformé en **discussion/thread**. (via l'UI ou via un email)

Une thread groupe plusieurs commentaires et a un statut

- La discussion démarre avec un statut **unresolved**
- Une discussion non résolue peut empêcher la fusion
- Elle se termine avec le statut **resolved**



Commentaires/discussions sur un commit

Pour créer un commentaire/discussion sur un commit

- Afficher les commits liés au MR
- Sur un commit, accéder à l'onglet *Changes* et laisser un commentaire
- La discussion apparaît dans l'onglet discussions du MR et peut être résolue via le bouton « *Resolve Discussion* »

Il est possible de

- voir toutes les discussions non résolues
- De déplacer les discussions non résolues vers une issue



Revue de code

Lors d'une revue de code, le reviewer peut créer des commentaires qui ne sont visibles que par lui.

Lorsqu'il est prêt, il publie l'ensemble des commentaires en une fois.

- 1) Sélectionner l'onglet **Changes** de la MergeRequest
- 2) Sélectionner l'**icône de commentaire** en face du patch
- 3) Ecrire le 1^{er} commentaire et activer le bouton **Start Review**
- 4) Faire d'autres commentaires et activer le bouton **Add to review**
- 5) A la fin, activer le bouton **Submit the review**



Configuration des MR

Dans le menu ***Project → Settings → General***, le mainteneur peut configurer les merge request

- Méthode de fusion :
 - Commit de merge
 - Commit de merge avec Rebasing si conflit
 - Rebasing obligatoire
- Options de fusion : Résolution automatique des discussions, hooks, Suppression de la branche source cochée par défaut
- Squash des commits (perte de l'historique des commits de la branche source)
 - Autoriser, Favoriser ou empêcher
- Vérifications avant la fusion
 - La pipeline doit s'être exécutée avec succès
 - Tous les discussions doivent être résolues



Conflits

Lorsqu'une MR a des conflits, Gitlab peut proposer de les résoudre via l'UI

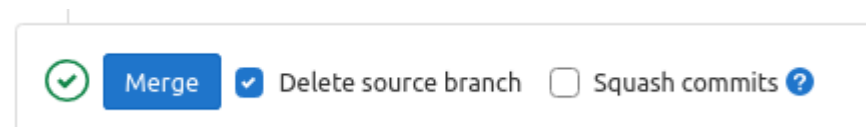
- *GitLab* résout les conflits en créant un commit de merge dans la branche source.
- Le commit peut alors être testé avant d'affecter la branche cible.



Merge

Lorsque le mainteneur active le bouton *Merge*, il a généralement le choix :

- Pour supprimer automatiquement la branche source
- Pour fusionner (squash) tous les commits en 1 seul



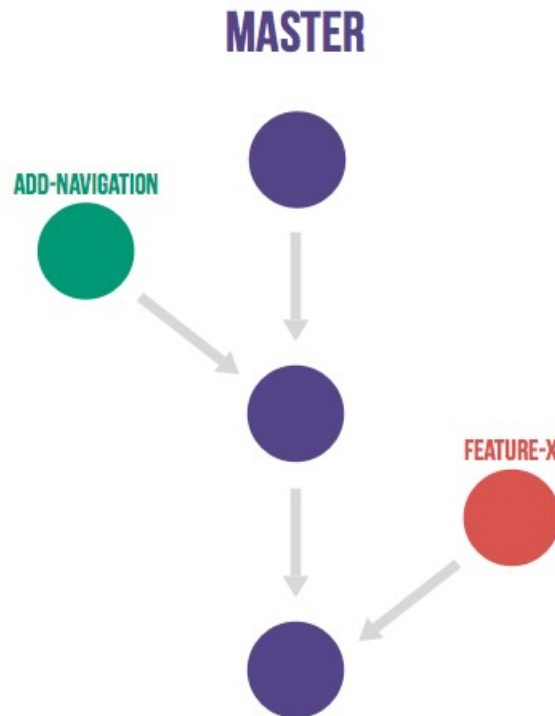


Gitlab Flow

Gitlab Flow est une stratégie simplifiée d'utilisation des branches pour un développement piloté par les features ou le suivi d'issues

- 1) Les corrections ou fonctionnalités sont développées dans une **feature branch**
- 2) Lorsque le développeur considère que le travail est terminé, il émet une demande de fusion dans la branche principale
- 3) Des tests sont effectués en isolation, une revue de code peut être effectué par le mainteneur.
Le responsable décide alors d'accepter ou de refuser la fusion
- 4) Si la fusion est effectuée, les modifications de code sont intégrées dans la branche principale et la branche source est supprimée
- 5) La branche principale reste toujours potentiellement livrables en production

Features et Master





Gestion des sources et collaboration

L'unique source de vérité
Gitlabflow et MergeRequest
Déclinaisons



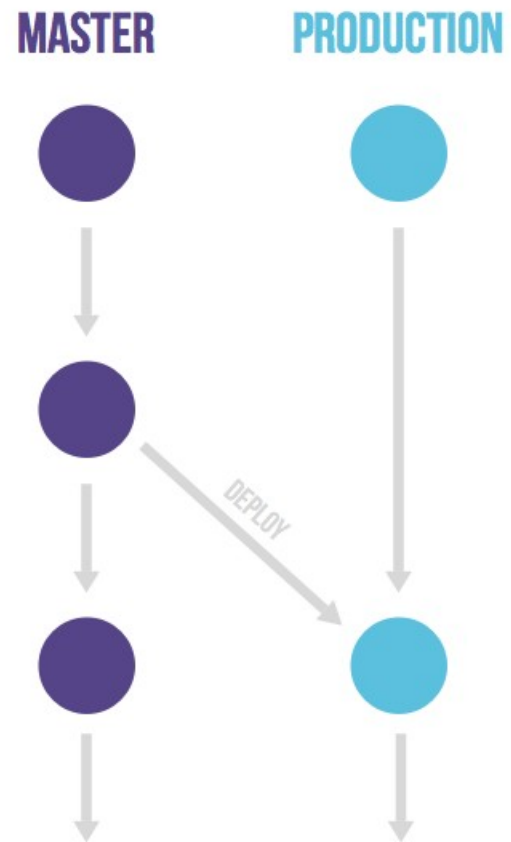
Branche de production

Si l'on veut maîtriser les déploiement vers la production, Il est possible d'utiliser une branche ***production*** qui reflète le code qui a réellement été déployé

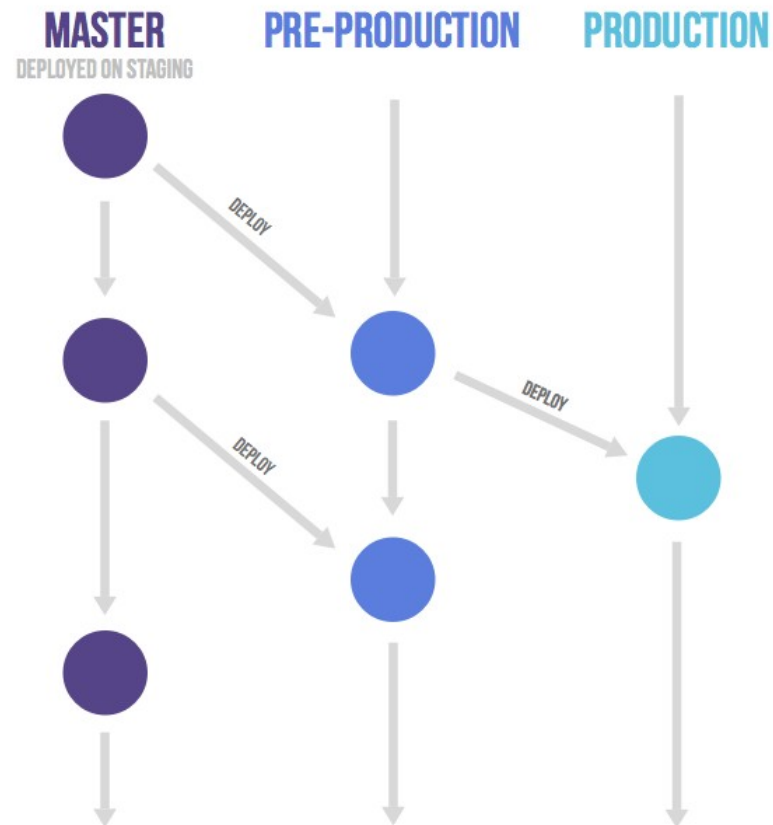
La mise en production consiste alors à fusionner *main* avec la branche de production puis déployer à partir de production.



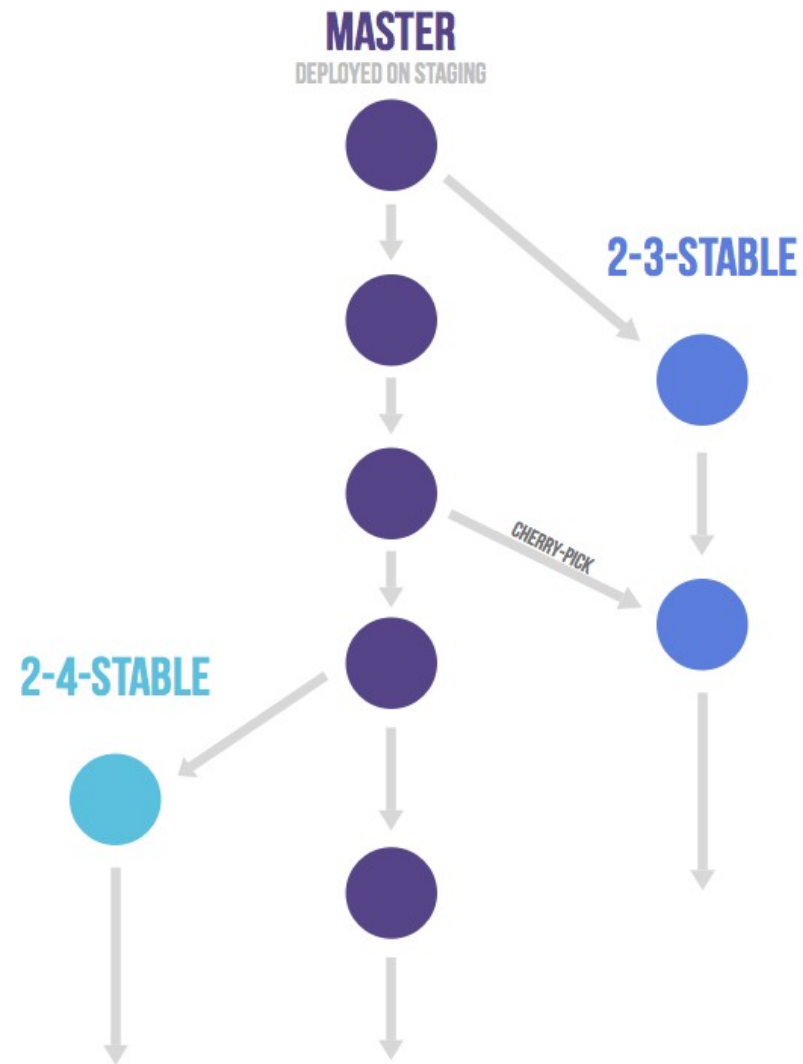
main et Production



Déclinaison avec staging



Branches de releases





Pipelines CI/CD

Introduction

Jobs et Runners

UI Pipelines

Définition de Pipelines

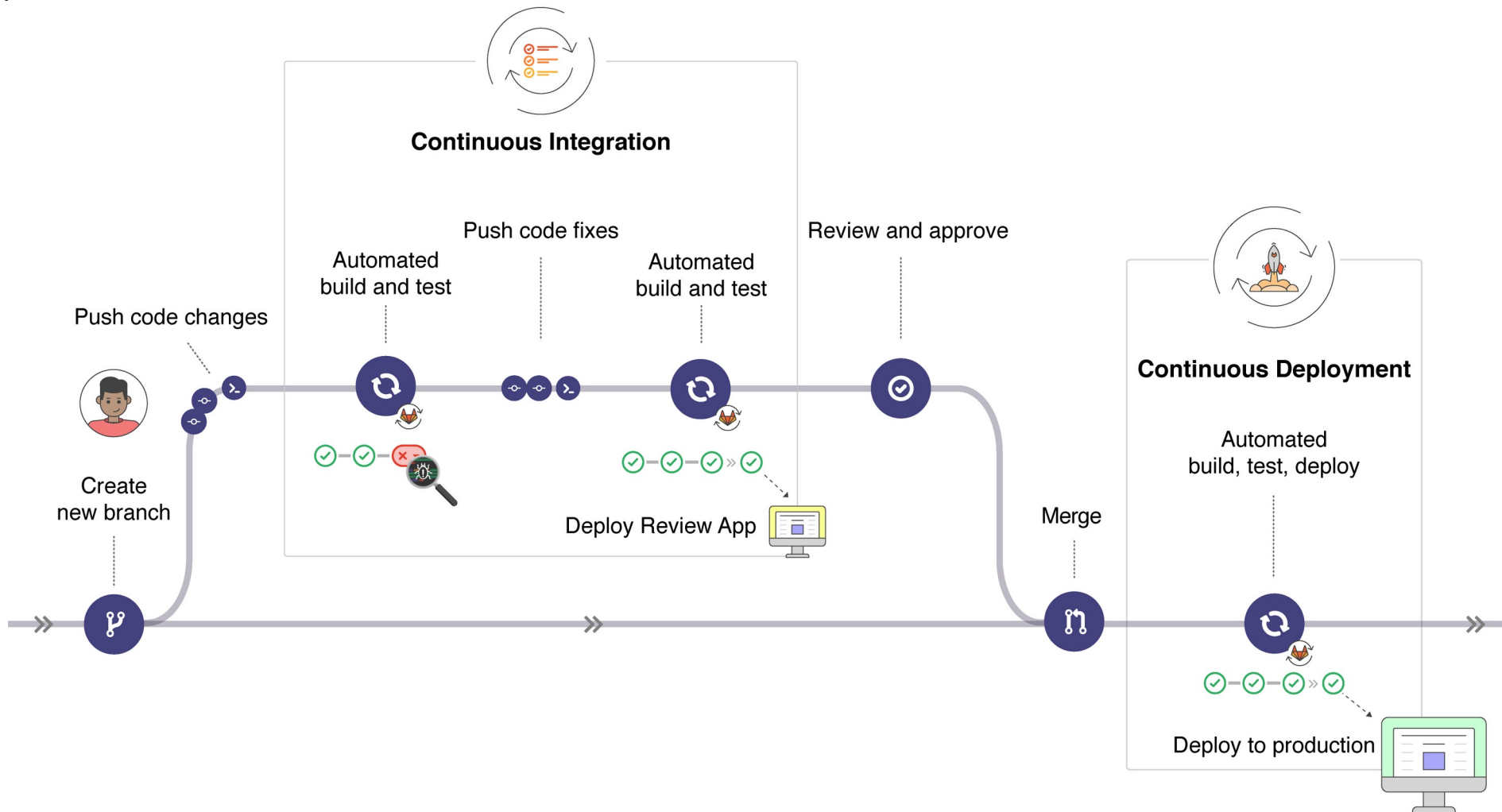


Introduction

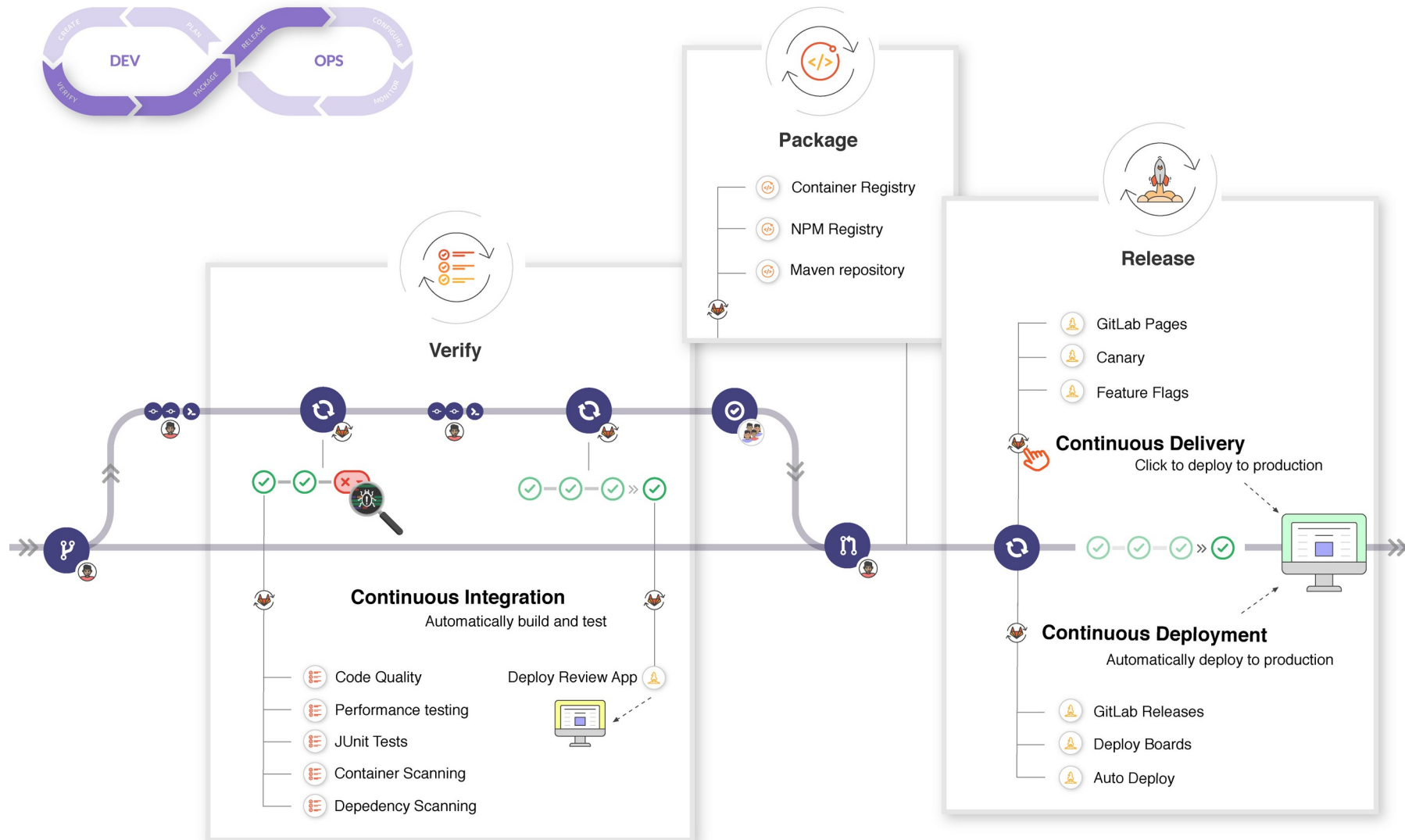
GitLab CI/CD est un outil permettant l'intégration, la livraison ou le déploiement continu

- **CI** : A chaque push, une pipeline de scripts pour construire, tester, analyser est exécutée avant de la fusionner dans la branche principale
- **CD** : A chaque push sur la branche principale, on effectue une release ou on déploie en prod
Les dernières étapes peuvent être manuelles

CI/CD



En plus détaillé





Typologie des tests

Test Unitaire :

*Est-ce qu'une simple classe/méthode fonctionne correctement ?
JUnit, PHPUnit, UnitTest, Karma, Mockito*

Test d'intégration :

*Est-ce que plusieurs classes/couches fonctionnent ensemble ?
BD/Serveurs embarqués*

Test fonctionnel :

*Pour l'utilisateur final, est-ce que mon application fonctionne ?
Selenium, HttpUnit, Protractor*

Test de performance :

*Est-ce que mon application fonctionne bien ?
JMeter, Gatling*

Test d'acceptation :

*Est-ce que mon client aime mon application ?
Cucumber*



Classification pipeline DevOps

Dans le cadre d'une pipeline DevOps, classification en fonction de :

- Le test nécessite t il un provisionnement d'infrastructure ?
- Durée d'exécution

=> Conditionne la position des tests dans la pipeline

Typiquement :

- Tests unitaires/intégration en premier,
- Test fonctionnel, d'acceptation de performance sur les infrastructures d'intégration, recette, pré-production et de production



Analyse statique

L'analyse statique est l'analyse du code sans l'exécuter.

Les objectifs sont :

- La mise en évidence d'éventuelles erreurs de codage
- La vérification du respect du formatage convenu.
- La vérification des licences utilisées
- Les détections de vulnérabilités



Pipelines CI/CD

Introduction

Jobs et Runners

UI Pipelines

Définition de Pipelines



Runner

Les jobs de builds sont exécutés via des **runners**

GitLab Runner est une application qui s'exécute sur des machines distinctes et qui communique avec Gitlab.

Un runner peut être dédié à un projet à un groupe de projet ou peut être partagé par tous les projets



Type de Runners

Un **Runner** peut être une machine virtuelle, une machine physique, un conteneur docker ou un pod dans un cluster *Kubernetes*.

GitLab et les Runners communiquent via une API
=> La machine du runner doit avoir un accès réseau au serveur Gitlab.

Pour disposer d'un runner :

- Il faut l'installer
- L'enregistrer comme runner partagé (administrateur seulement) ou comme dédié au projet



Installation GitlabRunner

L'installation s'effectue :

- Via des packages
Debian/Ubuntu/CentOS/RedHat
- Exécutable MacOS ou Windows
- Comme service Docker
- Auto-scaling avec Docker-machine
- Via Kubernetes



Enregistrement

Avant la procédure d'enregistrement du runner, il faut obtenir un token via l'UI de gitlab

La commande ***gitlab-runner register*** exécutée dans l'environnement du runner démarre un assistant posant les question suivantes :

- L'URL de *gitlab-ci*
- Le token
- Une description
- Une liste de tags
- L'exécuteur (shell, docker, ...)
- Si docker, l'image par défaut pour construire les builds



Exécuteurs

Les exécuteurs d'un runner ont une influence sur les jobs que le runner peut exécuter :

- **Shell** : Le plus facile à installer mais toutes les dépendances du projet doivent être pré-installées sur le runner (git, npm, jdk, ...)
- **Virtual Machine** : Nécessite Virtual Box ou Parallels. Les outils sont pré-installés sur la VM
- **Docker** : Permet d'exécuter des builds avec une image docker. D'autres services docker peuvent être démarrés pendant le build (une base de données par ex.)
- **Docker-machine** : Idem docker + auto-scaling. Les exécuteurs de build sont créés à la demande
- **Kubernetes** : Utilisation d'un cluster Kubernetes. Via l'API, le runner crée des pods (machine de build + services)
- **ssh** : Peu recommandé, exécute le build via ssh sur une machine distante



Configuration runner

La configuration d'un runner présente dans ***/etc/gitlab-runner/config.toml*** déterminent d'autres paramètres techniques du runner

```
concurrent = 5 // Nombre de jobs concurents  
check_interval = 0
```

```
[session_server]  
  session_timeout = 1800  
[[runners]]  
  name = "Another shell executeur"  
  url = "http://localhost"  
  token = "1NkCzKU1x_S6hz6VQ2Uu"  
  executor = "shell"  
[runners.custom_build_dir]  
[runners.cache]  
  [runners.cache.s3]  
  [runners.cache.gcs]
```



Affectation d'un runner et tags

Lorsqu'une pipeline doit être exécutée, Gitlab affecte un runner pour le job.

Il choisit de préférence un runner dédié au projet

Chaque runner peut également avoir une liste de tags et une pipeline peut définir également des tags

=> Gitlab recherche alors le runner ayant les mêmes tags que le job

Si il ne trouve pas, la pipeline ne démarre pas (état stuck)



Pipelines CI/CD

Introduction
Jobs et Runners
UI Pipelines
Définition de Pipelines



Editeur

Un éditeur en ligne de *.gitlab-ci.yml* est disponible

Il permet une validation

Repository → Files → .gitlab-ci.yml → Pipeline Editor



Exécution des pipelines

Les pipelines s'exécutent automatiquement à chaque push

Elles peuvent être également planifiées pour s'exécuter à des intervalles réguliers via l'UI ou l'API

Settings → CI/CD → Schedules

Enfin, elles peuvent être démarrées manuellement par l'UI

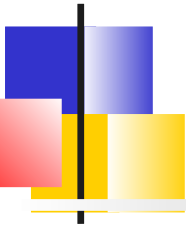


Tableau de bord d'exécution

Status	Pipeline	Triggerer	Stages	
<div>ⓧ blocked</div> <div>🕒 00:13:00</div>	<div>Adding probe</div> <div>#123967683 master 302815b0 </div> <div>latest Auto DevOps</div>		<div>🟢🟢🟡🟢⚙️🔴</div>	<div>▶️ ⌂ 🛑 ⬇️</div>
<div>✅ passed</div> <div>🕒 00:21:12 📅 3 years ago</div>	<div>Bonjour</div> <div>#123967122 1-changer-hello-en-bonjour d9111678 </div> <div>latest Auto DevOps</div>		<div>🟢🟢🟢🟢🟡🟢</div>	<div>▶️ ⌂ ⌛ ⬇️</div>

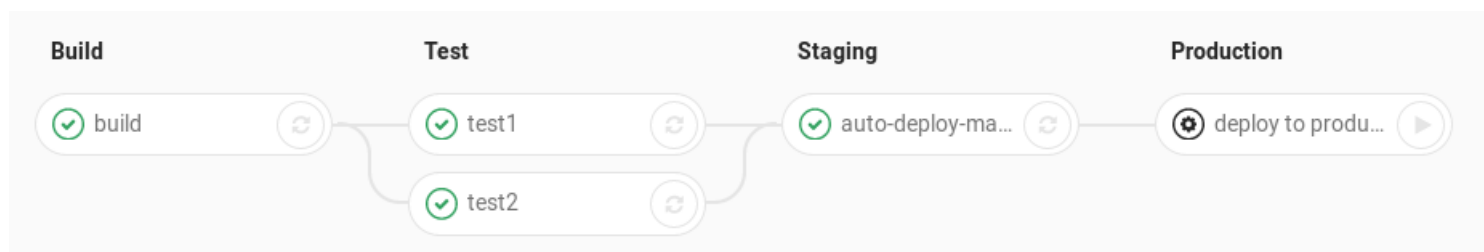
Un tableau de bord permet de voir les dernières exécutions de pipeline, leurs status, le commit, le déclenchement et l'exécution des tâches

Une barre de boutons permet de continuer ou redémarrer la pipeline et de télécharger les artefacts



Visualisation d'une pipeline

Le détail d'une pipeline est affichée graphiquement.



Il est possible de visualiser la sortie standard de chaque tâche en la sélectionnant

De déclencher une tâche manuelle



Pipelines CI/CD

Introduction
Jobs et Runners
UI Pipelines

Définition de Pipelines



Spécification de la pipeline

La spécification du job et de ses différentes phases peuvent être faits de différentes façons :

- **AutoDevOps** : Mode par défaut.
Gitlab choisit la pipeline en fonction du projet.
Nécessite des runners docker
- Fichier **.gitlab-ci.yml** à la racine du projet
Des gabarits selon les piles technologies sont proposés par Gitlab



AutoDevOps

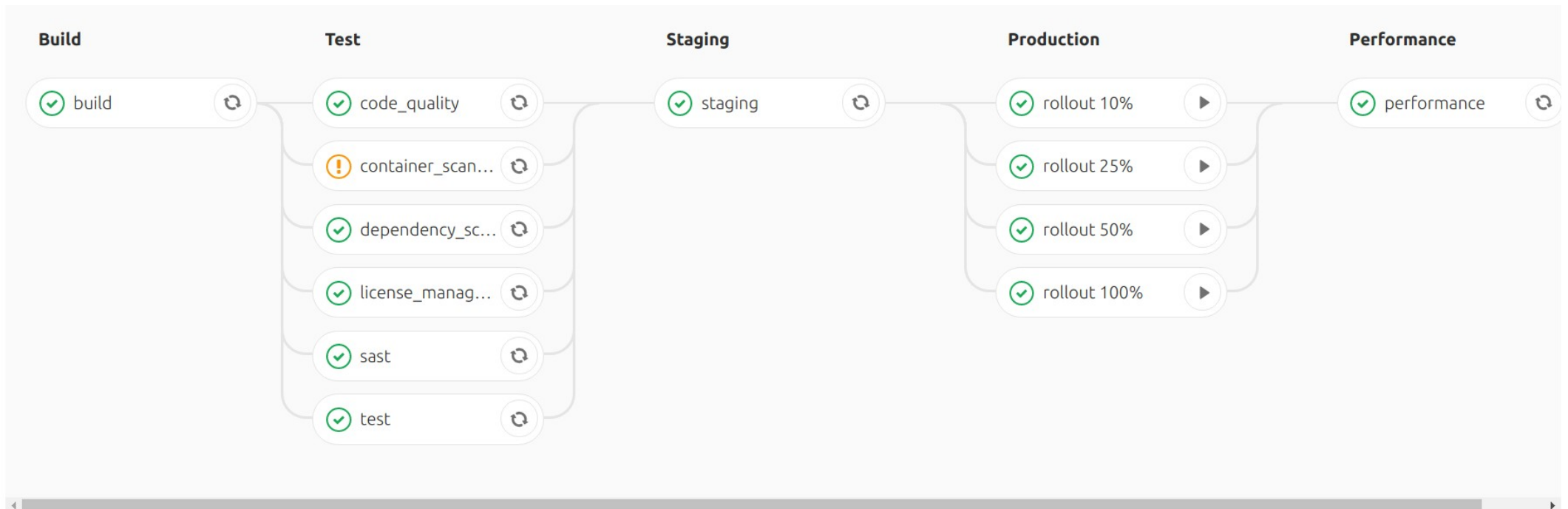
AutoDevOps est la pipeline qu'essaye d'appliquer GitLabCI sur un projet dans n'importe quelle technologie

- Docker pour builder, tester construire le conteneur
- Kubernetes (GKE ou Existant) : Pour les déploiements

Chaque branche du projet a alors un environnement de déploiement (URL d'accès) dédié

Les revues de code et de fonctionnalités sont alors faites en toute isolation

Pipeline AutoDevops de Gitlab CI





Jobs / phases / tâches

Le fichier *.gitlab-ci.yml* définit des **jobs**.

Les jobs sont constitués de **phases** exécutées séquentiellement.

- Par défaut, si une phase échoue, les phases suivantes ne sont pas exécutées.

Une phase est constituée d'une ou plusieurs **tâches** exécutées en parallèle.

- Chaque tâche est exécutée sur un runner

Les *runners* peuvent récupérer ou sauvegarder des résultats par le biais du serveur Gitlab



Directives de base

script : Seul mot-clé nécessaire, décrit les commandes de la tâche (shell Linux)

before-script, after-script : Les commandes exécutées avant/après chaque script. Permet d'initialiser le build, installer des outils



Syntaxe

stages:

- Build
- Test
- Staging
- Production

build:

stage: Build

script: make build dependencies

test1:

stage: Test

script: make build artifacts

test2:

stage: Test

script: make test

auto-deploy-ma:

stage: Staging

script: make deploy

deploy-to-production:

stage: Production

script: make deploy



Contexte des directives

En fonction de leur niveau (indentation *yml*), les directives s'appliquent à l'ensemble des tâches ou à une tâche particulière.

Les principales directives globales sont :

- **default** : Valeurs par défauts des tâches. Inclut entre autres :
 - **image** : L'image docker utilisée pour le build (Nécessite un runner docker)
 - **services** : Les services devant être démarrés avant le build
 - **tag** : Tags du jobs permettant de l'affecter au bon runner
 - **timeout, retry, cache**, ...
- **variables** : Variables d'environnement disponibles pour le job
- **stages** : Liste des phases



Variables

GitLab CI/CD fournit un ensemble prédéfini de variables d'environnement (Id d'issue, commit ID, branch ...)

Des variables peuvent être définies par l'UI au niveau transverse projet (administrateur), au niveau groupe ou projet.

Enfin des variables peuvent être définies dans *.gitlab-ci.yml* au niveau job ou tâches

Une variable peut être configurée comme étant masquée ou protégée¹

Le build peut alors utiliser ces variables avec la notation **\$**
{variable}

Ex :

```
docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"  
$CI_REGISTRY
```



Variable `GIT_STRATEGY`

La variable **`GIT_STRATEGY`** peut être positionnée dans la pipeline pour conditionner, l'interaction du runner avec le dépôt.

La variable peut prendre 3 valeurs :

- **`clone`** : Le dépôt est cloné par chaque job
- **`fetch`** : Réutilise le précédent workspace si il existe en se synchronisant ou effectue un clone (défaut)
- **`none`** : N'effectue pas d'opération *git*, utilisé généralement pour les tâches de déploiement qui utilisent des artefacts précédemment construits



Directive de tâches

Les directives de tâche relatives aux phases typiques d'une pipeline:

- **artifacts** : Liste des artefacts produits par la tâche et sauvegardés sur Gitlab
- **dependencies** : Liste des artefacts à récupérer de Gitlab
- **cache** : Liste des fichiers devant être cachés entre 2 exécution (performance de la pipeline)
- **coverage** : Expression régulière pour extraire la couverture des tests exécutés par la tâche
- **dast_configuration** : Configuration de *Dynamic Application Security Testing*
- **environment** : Le nom de l'environnement dans lequel le build déploie
- **release** : Indique au runner de générer une release
- **pages** : Publie le résultat du build comme Gitlab Page

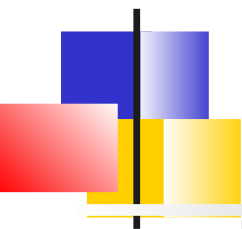


Artifacts et dependencies

Les fichiers définis par **artifacts** sont téléchargeable (tar.gz) via l'UI.
Ils sont conservés 1 semaine (par défaut)

La directive **dependencies** permet d'indiquer une dépendance entre 2 jobs.

- Elle a pour effet de récupérer les artefacts générés par la dépendance.
- Si, les dépendances ne sont pas disponibles lors de l'exécution du job, il échoue.



Réutilisation des artefacts

```
build:osx:
  stage: build
  script: make build:osx
  artifacts:
    paths:
      - binaries/

build:linux:
  stage: build
  script: make build:linux
  artifacts:
    paths:
      - binaries/

test:osx:
  stage: test
  script: make test:osx
  dependencies:
    - build:osx
```



Directives de conditions

rules : Liste d'expressions qui conditionne la création du job

when conditionne l'exécution d'un job. Les valeurs possibles sont :

- **on_success** : Tous les jobs des phases précédentes ont réussi (défaut).
- **on_failure** : Au moins un des jobs précédents a échoué
- **always** : Tout le temps
- **manual** : Exécution manuelle déclenchée par l'interface

only et **except** limitent l'exécution d'un job à une branche ou une tag. Il est possible d'utiliser des expressions régulières



Exemples

Exécute le job seulement si un fichier Dockerfile existe

rules:

- if: \$CI_COMMIT_BRANCH
exists:
 - Dockerfile

Toutes les refs git démarrant avec issue-

job:

only:

- /^issue-.*\$/

Seulement via l'API ou une planification

job:

only:

- triggers
- schedules

Seulement les branches en fonction d'une variable

deploy:

only:

refs:

- branches

variables:

- \$RELEASE == "staging"



Control Flow

allow_failure permet à une tâche d'échouer sans impacter le reste de la pipeline.

- La valeur par défaut est *false*, sauf pour les jobs manuels.

retry permet de configurer le nombre de tentatives avant que le job soit en échec.

tags : Liste de tags pour sélectionner un runner

parallel : Nombre d'instances du jobs exécutés en parallèle

trigger : Permet de déclencher une autre pipeline à la fin d'un job.



Organisation des fichiers pipelines

Gitlab fournit un support afin que les organisations puissent partager des pratiques entre projets.

Les fichiers peuvent inclure ou étendre d'autres fichiers *yaml* fournis par l'entreprise ou les gabarits fournis par Gitlab.



Phases de build

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Déploiement et release

Tests post-déploiements

Gestion de l'infrastructure



Construction et build

Les premières tâches d'une pipeline consiste généralement

- A exécuter les tests unitaires développeurs
- Si ceux-ci réussissent, packager le code source

En fonction des piles technologiques, le packaging du code source diffère

- Java : Compilation + création d'un jar
- Javascript : Minification et obfuscation du code + création d'un zip
- ...

Gitlab s'appuie alors sur des outils de la pile technologique qui sont soit préinstallés sur des runners ou dans des images docker



Outils de build

Maven (*Java*): Le plus répandu et le plus supporté.

Gradle : « *Build As Code* ».

S'applique à d'autres langages que Java (C, C+, Python, Php, ...)

npm, yarn, webpack, ... : Monde JavaScript

tslint : Linter typescript

Composer, PHPUnit : PHP

pip, unittest : Python



Support Gitlab pour les tests

Pour ces tests de début de pipeline, Gitlab permet :

- De publier les résultats des tests et de les attacher à une MR
- Si *Ruby*, de définir des *FailFast Test* permettant d'économiser des ressources runners
- De publier la couverture de test et de conditionner une MR à un certain seuil de couverture **Premium**



Publier les résultats de test

Gitlab ne supporte que le format JUnit

Il suffit de placer la directive

artifacts:reports:junit dans *.gitlab-ci.yml*

```
ruby:
  stage: test
  script:
    - bundle exec rspec --format progress --format RspecJUnitFormatter --
out rspec.xml
  artifacts:
    when: always
    paths:
      - rspec.xml
  reports:
    junit: rspec.xml
```

Publication résultat de test

Gitlab propose alors des rapports de tests qui permettent de facilement isoler les tests ayant échoués

Pipeline

Needs

Jobs 1

Failed Jobs 1

Tests 3

Summary

3 tests

2 failures

0 errors

33.33% success rate

16.00ms

Jobs

Job	Duration	Failed	Errors	Skipped	Passed	Total
jest	16.00ms	2	0	0	1	3

! Test summary contained 3 failed out of 3 total tests

View full report

Collapse

! rspec found 1 failed out of 1 total test

✖ Failed 1 time in master in the last 14 days

User#full_name returns first_name + last_name

! jest found 2 failed out of 2 total tests

✖ New Calculator #add returns the sum of the 2 given numbers

✖ Failed 1 time in master in the last 14 days

Calculator #subtract returns the difference between the 2 given numbers



Couverture des tests

Si l'on s'est équipé d'un outil calculant la couverture des tests, il est possible de configurer la pipeline Gitlab afin que ses métriques soient publiés.

Les résultats sont visibles :

- Dans les Merge request
- Dans les analytiques projets
- Dans les analytiques groupe
- Sous forme de badge au niveau d'un dépôt



Mise en place

La mise en place consiste à utiliser le mot-clé **coverage** dans *.gitlab-ci.yml* et d'indiquer une expression régulière permettant d'extraire l'information de la sortie standard.

Exemple *JacoCo (Java/Kotlin)*

```
/Total.*?([0-9]{1,3})%/ .
```

Publication

⚠ Pipeline #4637061 passed with warnings for 99f87a83. [View details](#)

Coverage 91.98%

Accept Merge Request

The source branch will be removed.

[✎ Modify commit message](#)

✅ Post Test

✅ passed

#5199993

coverage

🕒 01:20

📅 about 2 hours ago

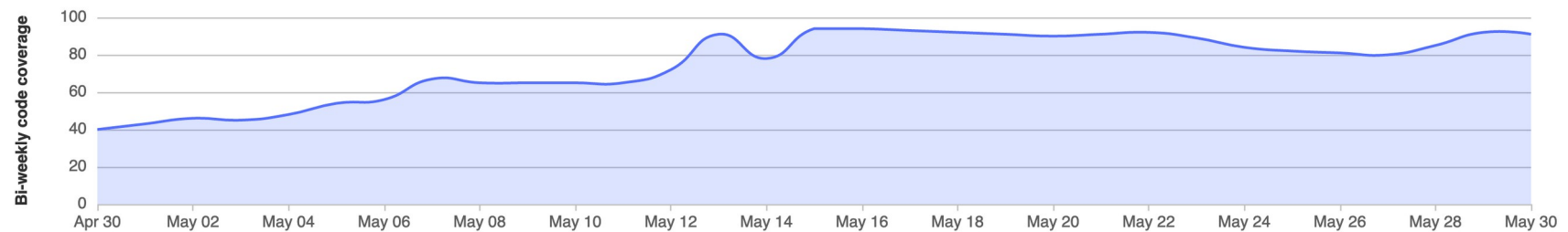
91.98%



Code coverage statistics for master Mar 12 - Jun 10

[Download raw data \(.csv\)](#)

rspec ▾



— rspec Avg: 74.9 · Max: 94

Conditionner une MR

×

Add approval rule

Rule name

Coverage-Check

Examples: QA, Security.

Target branch

Any branch


Apply this approval rule to any branch or a specific protected branch.


Approvals required

1

Add approvers

Search users or groups

 Administrator



Cancel

Add approval rule



Phases de build

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Déploiement et release

Tests post-déploiements

Gestion de l'infrastructure



Introduction

Gitlab fournit également du support pour les analyses statiques de code source

- Analyse qualité
- Utilisation des licences
- Détection de vulnérabilités

La mise en place s'effectue dans *.gitlab-ci.yml* et les résultats sont publiés dans le projet



Analyse qualité

Gitlab s'appuie sur l'outil **Code Climate**¹ et ses plugins pour analyser le code source.

Les résultats sont disponibles² :

- Dans une Merge request
- Dans le détail d'une pipeline
- Dans la vue qualité d'un projet

Ils peuvent également être téléchargés au format brut

1. Supporte : Ruby, Python, PHP, JavaScript, Java, TypeScript, GoLang, Swift, Scala, Kotlin, C#

2. Dépend fortement de la licence



Mise en place

La mise en place nécessite des pré-requis :



- Une phase nommée **test** dans *.gitlab-ci.yml*
- Suffisamment d'espace de stockage
- Contraintes sur les runners en fonction de si ils sont partagés ou pas

Pour autoriser l'analyse qualité :

- Utiliser *AutoDevOps*
- Inclure le gabarit de qualité de code dans *.gitlab-ci.yml*
include:
 - template: Code-Quality.gitlab-ci.yml



Affichage Merge Request

 Code quality degraded on 7746 points 

Collapse

- ◆ Critical - Consider simplifying this complex logical expression.
in [scripts/frontend/stylelint/stylelint-utils.js:15](#)
- ◆ Critical - Consider simplifying this complex logical expression.
in [app/assets/javascripts/projects/settings/access_dropdown.js:248](#)
- ◆ Critical - CSRF vulnerability in OmniAuth's request phase
in [Gemfile.lock:810](#)
- ▼ Major - Function `buildConfig` has 7 return statements (exceeds 4 allowed).
in [workhorse/main.go:67](#)
- ▼ Major - Function `run` has 8 return statements (exceeds 4 allowed).
in [workhorse/main.go:152](#)
- ▼ Major - Method `Resizer.Inject` has 5 return statements (exceeds 4 allowed).
in [workhorse/internal/imageresizer/image_resizer.go:164](#)
- ▼ Major - Method `Resizer.tryResizeImage` has 7 return statements (exceeds 4 allowed).
in [workhorse/internal/imageresizer/image_resizer.go:268](#)
- ▼ Major - Function `unpackFileFromZip` has 7 return statements (exceeds 4 allowed).
in [workhorse/internal/artifacts/entry.go:64](#)



Sécurité

GitLab analyse la sécurité d'une application, soit dans le cadre d'une pipeline, soit de façon planifiée.

Cela couvre :

- Le **code source**

Static Application Security Testing (SAST) + Analyze du dépôt pour la détection de secrets

- Les **dépendances** (Librairies, conteneurs)

Dependencies Scanning, Container Scanning

- Les **vulnérabilités** dans une application Web en cours d'exécution.

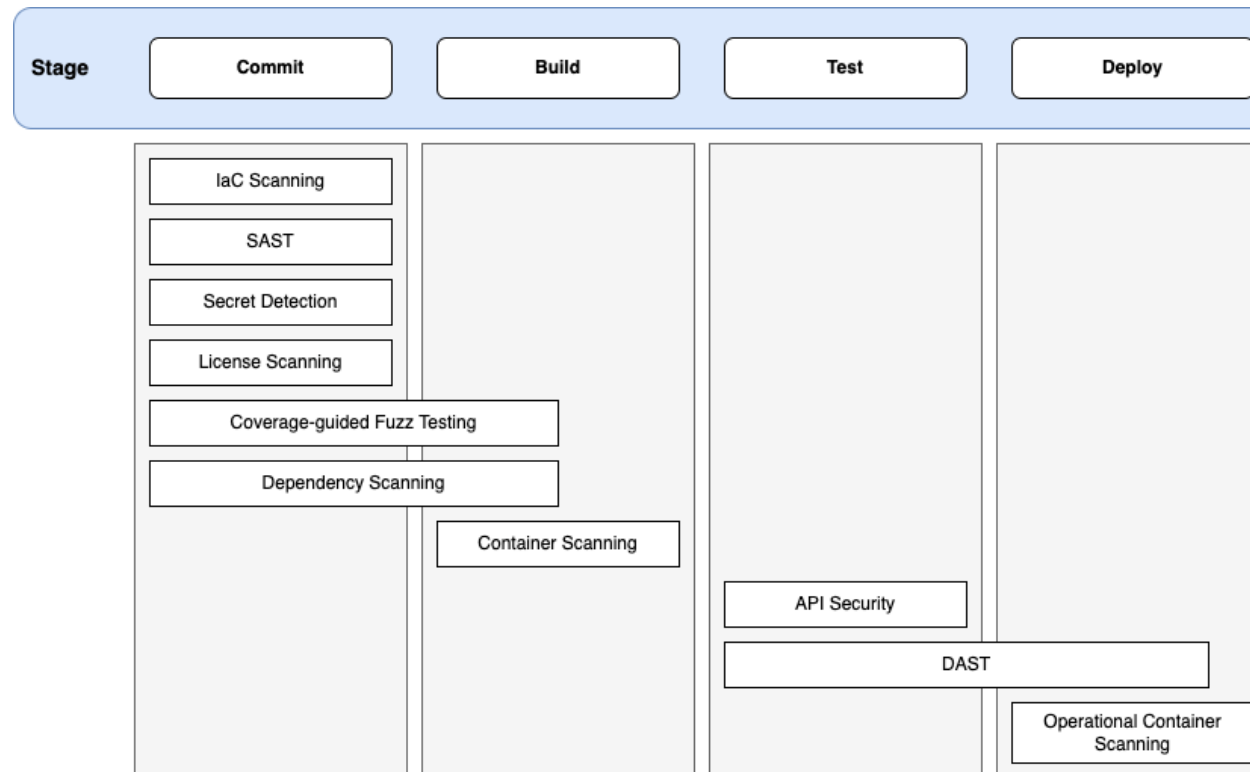
Dynamic Application Security Testing (DAST), Analyse des APIs pour détection de vulnérabilités connus ou inconnus (API fuzzing)

- **L'infrastructure** : Configuration de l'IAC (Infrastructure As Code)



Étapes vs Outils

Chaque outil intervient à différentes étapes de la pipeline





Mise en place

La mise en place de ces outils dépend fortement de la licence et certains outils ne sont disponibles qu'en version payante.

Les résultats des outils peuvent être publiés dans l'interface si une directive ***artifacts:reports <mot-clé>*** est présente dans *.gitlab-ci.yml*



Licenses

Gitlab permet également de fixer des politiques transverses quant à l'utilisation de produit tiers.

L'analyse de code permet de détecter les licences associées aux dépendances et d'afficher des rapports si le projet utilise des dépendances non-permises.



Outils Free

SAST : Static Application Security Testing

```
stages:  
- test  
sast:  
  stage: test  
include:  
- template: Security/SAST.gitlab-ci.yml
```

Secret Detection

```
include:  
- template: Security/Secret-Detection.gitlab-ci.yml
```

Infrastructure as Code (IaC) Scanning

```
include:  
- template: Security/Secret-Detection.gitlab-ci.yml
```



Phases de build

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Déploiement et release

Tests post-déploiements

Gestion de l'infrastructure



Gitlab

Gitlab offre plusieurs supports pour stocker et partager les artefacts construits :

- **GitLab Package Registry** est un registre privé ou public supportant les gestionnaires de packages courants. (Composer, Conan, Generic, Maven, npm, NuGet, PyPI, RubyGems)
- **GitLab Container Registry** est un registre privé pour les images Docker
- **GitLab Terraform Module Registry** supporte les modules Terraform

D'autre part, *Dependency Proxy* est un proxy local utilisé pour les images et paquets fréquemment utilisés.

Bien sûr, il est possible des solutions tierces (Nexus, ...)

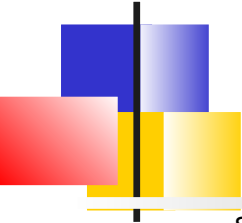


Gitlab Package Registry

GitLab Package Registry est en fait un projet Gitlab sans dépôt de source associé.

Les packages publiés héritent de la visibilité du projets

Pour publier, les autres projets doivent s'authentifier auprès du registre via des jetons (personnel ou job) et configurer leur outil de build afin de publier vers la bonne URL



Exemple Maven

settings.xml

```
<server>
  <id>gitlab-maven</id>
  <configuration>
    <httpHeaders>
      <property>
        <name>Job-Token</name>
        <value>${env.CI_JOB_TOKEN}</value>
      </property>
    </httpHeaders>
  </configuration>
</server>
```

pom.xml

```
<distributionManagement>
  <repository>
    <id>gitlab-maven</id>
    <url>https://gitlab.example.com/api/v4/projects/<project_id>/packages/maven</url>
  </repository>
  <snapshotRepository>
    <id>gitlab-maven</id>
    <url>https://gitlab.example.com/api/v4/projects/<project_id>/packages/maven</url>
  </snapshotRepository>
</distributionManagement>
```



GitLab Container Registry

Avec **GitLab Container Registry**, chaque projet peut avoir son propre espace pour stocker les images Docker.

La fonctionnalité doit être activée par l'administrateur. Elle n'est typiquement accessible qu'en **https**

Les images doivent suivre une convention de nommage :

<registry URL>/<namespace>/<project>/<image>

Des permissions fines peuvent être associés au registre



Exemple

```
stage: build
before_script:
  - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" $CI_REGISTRY
# Default branch leaves tag empty (= latest tag)
# Other branches are tagged with the escaped branch name (commit ref slug)
script:
  - |
    if [[ "$CI_COMMIT_BRANCH" == "$CI_DEFAULT_BRANCH" ]]; then
      tag=""
      echo "Running on default branch '$CI_DEFAULT_BRANCH': tag = 'latest'"
    else
      tag=":$CI_COMMIT_REF_SLUG"
      echo "Running on branch '$CI_COMMIT_BRANCH': tag = $tag"
    fi
  - docker build --pull -t "$CI_REGISTRY_IMAGE${tag}" .
  - docker push "$CI_REGISTRY_IMAGE${tag}"
```



Terraform Module Registry

Avec ***Terraform Module Registry***, les projets GitLab peuvent devenir des registres privés pour les modules *terraform*¹.

Les modules peuvent être publiés par des jobs CI/CD puis consommés par d'autre projet

Les autres projets s'authentifient auprès du registre par des jetons



Publication de modules

Plusieurs façons pour publier un module :

- Utiliser directement l'API

```
PUT /projects/:id/packages/terraform/modules/:module-name/:module-system/:module-version/file
```

- Dans une pipeline CI/CD utiliser le gabarit

Terraform-Module.gitlab-ci.yml

- Dans une pipeline CI/CD, effectuer un appel d'API



Publication via gabarit

Le gabarit *Terraform-Module.gitlab-ci.yml* contient les 3 jobs :

- ***fmt*** : Valide le module Terraform.
- ***kics-iac-sast*** : Test le module du point de vue de la sécurité.
- ***deploy*** : Pour les pipelines tag. Déploie le module vers *Terraform Module Registry*.



Phases de build

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Déploiement et release

Tests post-déploiements

Gestion de l'infrastructure



Introduction

Les pipelines CI/CD déploient généralement sur différents environnements

Les **environnements** sont comme des tags décrivant où le code est déployé

Les **déploiements** sont créés lorsque les job déploient des versions de code vers des environnement

=> ainsi chaque environnement peut avoir plusieurs déploiements

GitLab:

- Fournit un historique complet des déploiements pour chaque environnement
- Garde une trace des déploiements => On sait ce qui est déployé sur les serveurs
- Si on dispose d'un service de déploiement comme Kubernetes, cela offre d'autres possibilités



Types d'environnement

Un environnement peut être :

- **Statique** : A un nom fixe comme « staging », « production ».
Réutilisé par des déploiements successifs.
Créé manuellement ou via une pipeline
- **Dynamique** : Créé par une pipeline.
Utilisé par un unique déploiement puis arrêté et supprimé.
A un nom dynamique basé sur une variable de CI/CD.
Permet de valider une fonctionnalité en live (Review apps).



Définition des environnements

Les environnements sont définis dans *.gitlab-ci.yml*

Le mot-clé ***environment*** indique à *GitLab* que ce job est un job de déploiement. Il peut être associée à une URL

=> Chaque fois que le job réussit, un déploiement est enregistré, stockant le SHA Git et le nom de l'environnement.

Operations → Environments

Le nom de l'environnement est accessible via le job par la variable `$CI_ENVIRONMENT_NAME`



Example

```
deploy_staging:
  stage: deploy
  script:
    - echo "Deploy to staging server"
environment:
  name: staging
  url: https://staging.example.com
only:
  - master
```



Déploiement manuel

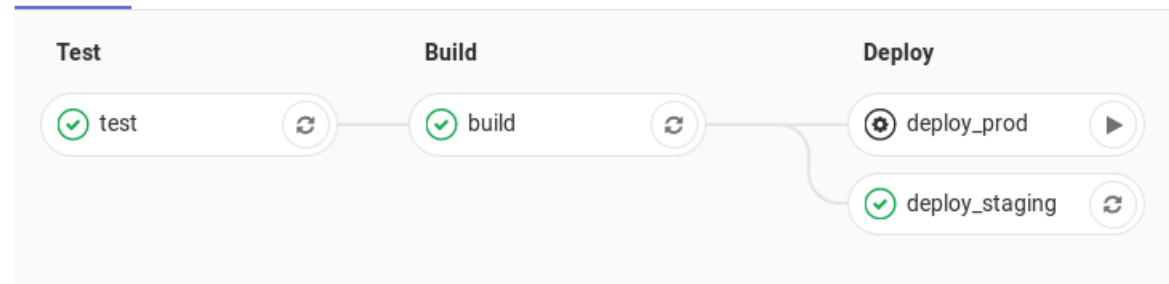
L'ajout de *when: manual* convertit le job en un job manuel et expose un bouton Play dans l'UI

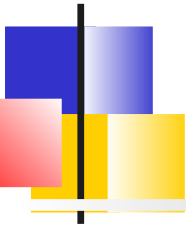
Use busybox

🕒 4 jobs from [master](#) in 5 minutes 25 seconds (queued for 1 minute 45 seconds)

🔑 [ec75f5bf](#) ... 📄

Pipeline Jobs 4





Environnements dynamiques

Dans le cas d'**environnements dynamiques**, les clés *name* et *url* peuvent utiliser :

- Les variables d'environnement prédéfinies
- Les variables de projets ou de groupes
- Les variables de *.gitlab-ci.yml*

Ils ne peuvent pas utiliser :

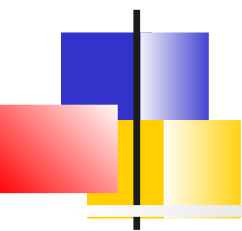
- Les variables définies dans script
- Du côté du runner

=> Il est possible de créer un environnement/déploiement pour chaque issue ou MR :
Les Review Apps

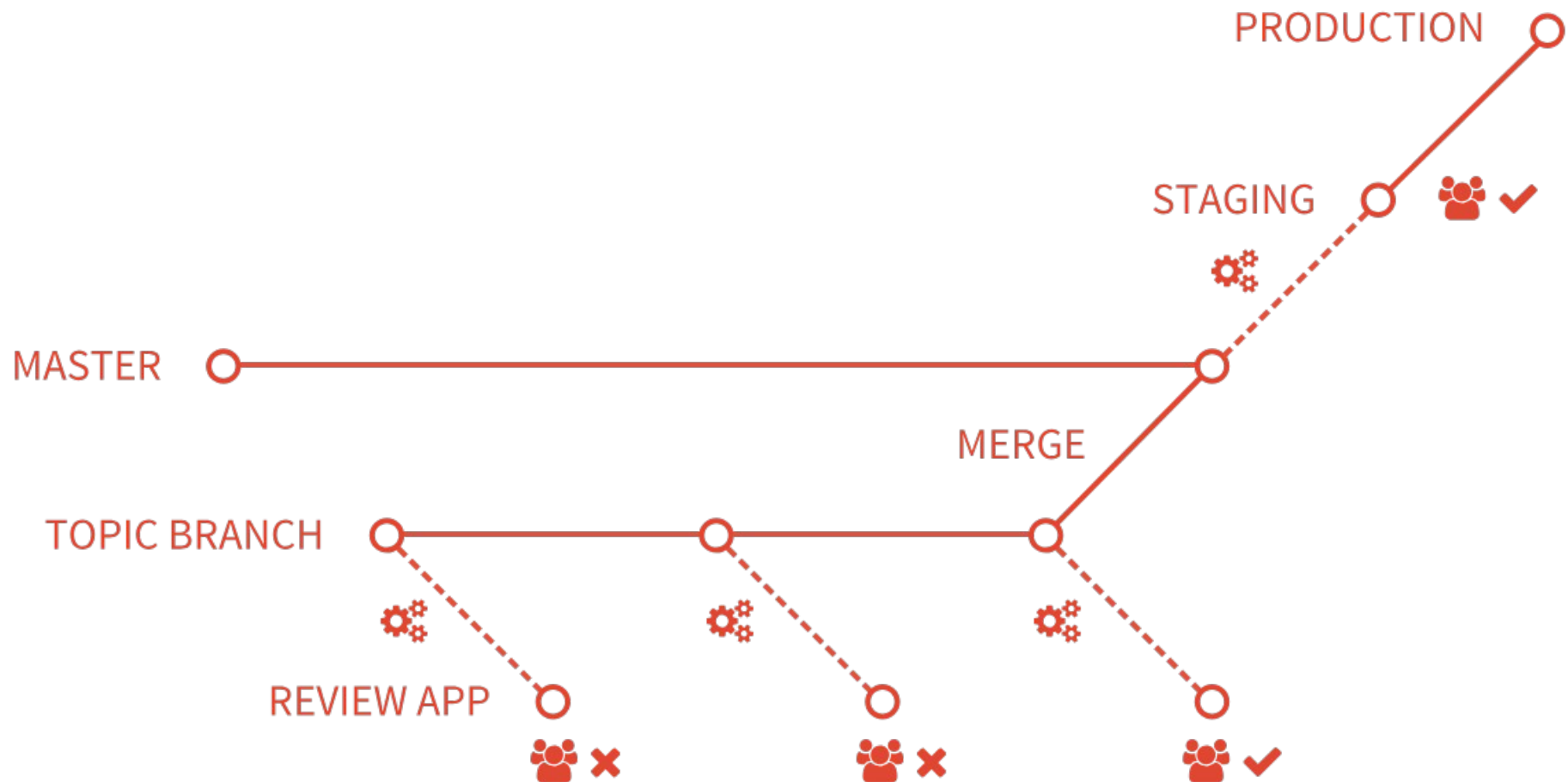


Example

```
deploy_review:
  stage: deploy
  script:
    - echo "Deploy a review app"
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
  only:
    - branches
  except:
    - master
```



Review App dans le workflow





Exemple complet

```
stages:
  ..
  - deploy
  ...

deploy_review:
  stage: deploy
  script: echo "Deploy a review app"
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
  only:
    - branches
  except:
    - master

deploy_staging:
  stage: deploy
  script: echo "Deploy to staging server"
  environment:
    name: staging
    url: https://staging.example.com
  only:
    - master

deploy_prod:
  stage: deploy
  script: echo "Deploy to production server"
  environment:
    name: production
    url: https://example.com
  when: manual
  only:
    - master
```




Arrêter un environnement

Arrêter un environnement consiste à appeler l'action ***on_stop*** si elle est définie.

- Cela peut se faire par l'UI ou automatiquement dans la pipeline.
- Lors du workflow Review App, l'action *on_stop* est automatiquement appelée à la suppression de la branche de feature.



Exemple

```
deploy_review:
  stage: deploy
  script:
    - echo "Deploy a review app"
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com
    on_stop: stop_review
  only:
    - branches
  except:
    - master

stop_review:
  stage: deploy
  variables:
    GIT_STRATEGY: none
  script:
    - echo "Remove review app"
  when: manual
  environment:
    name: review/$CI_COMMIT_REF_NAME
    action: stop
```



Release

Dans GitLab, une **Release** permet de créer un instantané du projet incluant les packages et les notes de version.

- La release peut être créée sur n'importe quelle branche.
- La création d'une Release crée un tag. Si le tag est supprimé, la release également.



Contenu et création d'une release

Une release peut contenir :

- Un instantané du code source
- Des packages créés à partir des artefacts des jobs
- Des méta-données de version
- Des releases notes



Création : Edition

Une *release* peut être créée

- Via un Job CI/CD
- Manuellement via l'UI Releases page
- Via l'API

Après avoir créé une release, on peut :

- Ajouter des release notes
- Ajouter un message pour le tag associé
- Associer des milestones avec
- Joindre des ressources (packages ou autres)



Création de release via un job CI/CD

Dans `.gitlab-ci.yml`, on crée des release en utilisant le mot-clé ***release***

3 méthodes typiques :

- Créer une release lorsqu'un tag est créé
- Créer une release quand un commit est fusionné dans la branche par défaut.
- Créer les méta-données de release dans un script personnalisé.



Exemple

Création lors fusion dans la branche par défaut

```
release_job:
  stage: release
  image: registry.gitlab.com/gitlab-org/release-cli:latest
  rules:
    - if: $CI_COMMIT_TAG
      when: never          # Ne pas exécuter si création manuelle de tag
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH # Branche par défaut
  script:
    - echo "running release_job for $TAG"
  release:
    tag_name: 'v0.$CI_PIPELINE_IID'          # Version incrémentée par pipeline
    description: 'v0.$CI_PIPELINE_IID'
    ref: '$CI_COMMIT_SHA'                    # tag créé à partir du SHA1.
```



Feature flags

Les **indicateurs de fonctionnalité** s'appuient sur des déploiements par petits lots.

Les fonctionnalités peuvent alors être activées ou désactivées pour des sous-ensembles d'utilisateurs.

Cette fonctionnalité s'appuie sur le projet OpenSource <https://github.com/Unleash/unleash>

Cela nécessite que l'application intègre l'API *unleash*



Phases de build

Construction et tests développeur

Analyses statiques

Dépôts d'artefacts

Déploiement et release

Tests post-déploiements

Gestion de l'infrastructure



Introduction

Après un déploiement, des tests de post-déploiement peuvent être inclus dans la pipeline.

Ces tests sont métiers et peuvent être faits avec tout type d'outil.

De son côté, Gitlab propose d'intégrer des outils de surveillance dans son interface.



Monitoring gitlab

De nombreuses fonctionnalités Gitlab concernant le monitoring des applications sont en cours de dépréciation.

Reste :

- La gestion des alertes et des incidents
- La traque des erreurs



Gestion des incidents

La gestion des incidents nécessite :

- L'intégration dans Gitlab des outils de monitoring.
- La gestion des astreintes dans le système de notifications des alertes.
- La classification des Alertes et Incidents.
- Informer les utilisateurs avec une page de Status.



Intégration

Alertes : Gitlab peut recevoir des alertes via des webhooks

Le menu **Settings > Monitor** permet d'activer les endpoints et de configurer les champs de l'alerte

Le menu **Monitor > Alerts** permet de les visualiser

Incidents : Les incidents sont créés manuellement dans l'interface

Gestion des astreintes **Premium**

Page de statut **Ultimate +** Compte AWS

Status page



[Contact support](#)

Incidents

April 1, 2020

Testing a new incident

Opened

[Full report](#)

March 27, 2020

2020-03-24: Last WALE backup was seen 20m 10s ago

Opened

[Full report](#)

March 27, 2020

2020-03-23: error burn-rate exceeding SLO across several services in CNY

Opened

[Full report](#)

March 27, 2020

2020-03-19: GitLab Appears to be throwing many 500s

Closed

[Full report](#)



Phases de build

Construction et tests développeur
Analyses statiques
Dépôts d'artefacts
Déploiement et release
Tests post-déploiements
Gestion de l'infrastructure



Introduction

La gestion de l'infrastructure avec Gitlab s'appuie sur ***Terraform***.

On peut alors définir des ressources versionnées, réutilisées et partagées :

- Composants de bas niveau : CPU/mémoire, stockage, réseau
- Haut-niveau : entrées DNS, fonctionnalités SaaS
- Adopter les déploiements ***GitOps***
- Utilisez GitLab comme stockage d'état Terraform.
- Stockez les modules Terraform



Intégration Terraform

L'intégration s'effectue via Gitlab CI

Un gabarit est fourni qui :

- Utilise ***GitLab-managed Terraform state*** pour stocker les états Terraform
- Déclenche 5 phases de pipeline : ***init, test, validate, build, deploy***
- Exécute les commandes Terraform ***test, validate, plan***, et ***plan-json*** ainsi que ***apply*** dans la branche par défaut
- Effectue un ***laC scanning*** pour vérifier la sécurité



Exemple *.gitlab-ci.yml*

include:

- # To fetch the latest template, use:
 - template: Terraform.latest.gitlab-ci.yml
- # To fetch the advanced latest template, use:
 - template: Terraform/Base.latest.gitlab-ci.yml
- # To fetch the stable template, use:
 - template: Terraform.gitlab-ci.yml
- # To fetch the advanced stable template, use:
 - template: Terraform/Base.gitlab-ci.yml

variables:

- TF_STATE_NAME: default
- TF_CACHE_KEY: default
- # If your terraform files are in a subdirectory, set TF_ROOT accordingly. For example:
 - # TF_ROOT: terraform/production



Kubernetes

GitLab peut se connecter à un cluster Kubernetes pour déployer, gérer et surveiller les applications

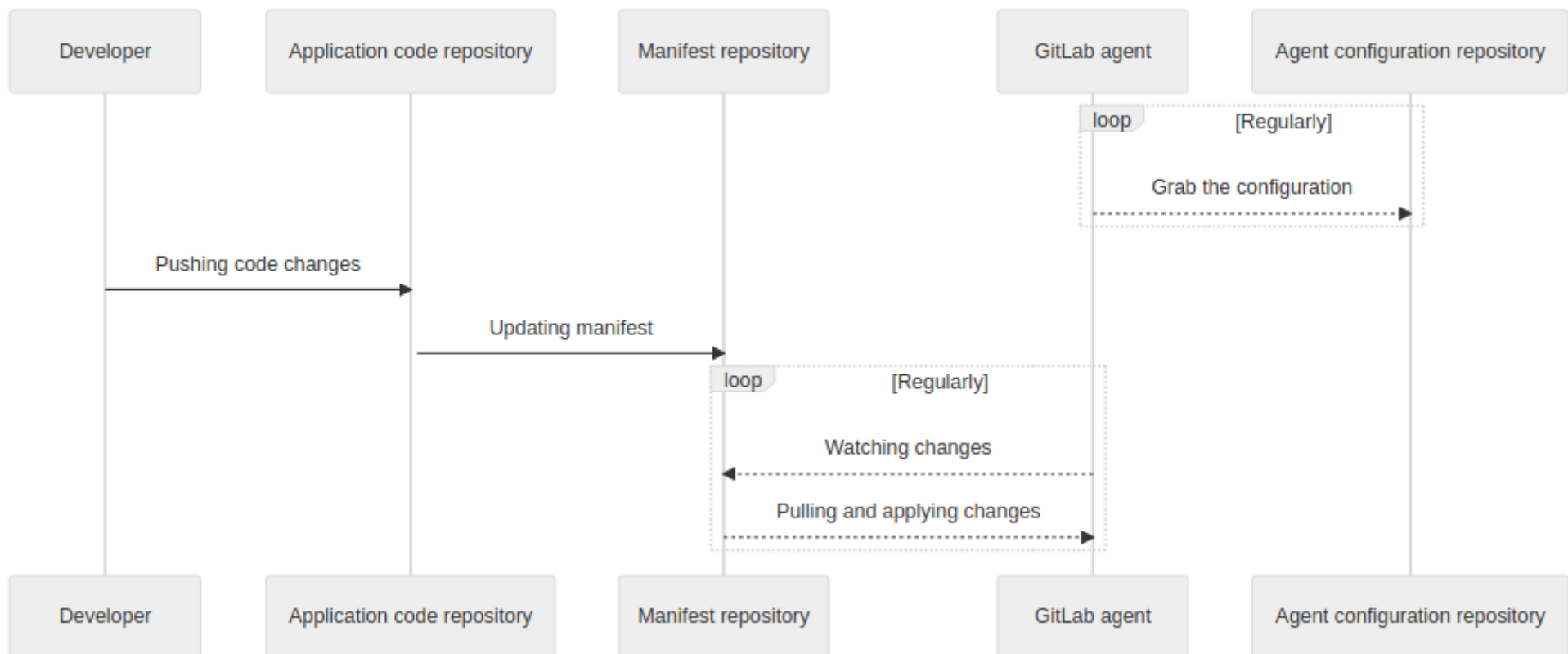
L'interface permet de créer des clusters vers 3 fournisseurs : Google GKE, Amazon EKS, Civo

La connexion s'effectue en installant un agent dans le cluster

2 workflows sont alors proposés :

- GitOps
- Gitlab CI/CD

GitOps





Gitlab CI/CD

Dans ce cas, les commandes de déploiement Kubernetes sont dans la pipeline.

La mise en place consiste à

- Enregistrer l'agent dans le projet
- Utiliser des commandes *kubectl* dans *.gitlab-ci.yml*



Example

deploy:

image:

name: bitnami/kubectl:latest

entrypoint: ['']

script:

- **kubectl config get-contexts**
- **kubectl config use-context path/to/agent/repository:agent-name**
- **kubectl apply -f myManifest.yml**



Runbooks

Les ***runbooks*** sont un ensemble de procédures documentées qui expliquent comment exécuter un processus particulier, (démarrer, arrêter, déboguer).

À l'aide de *Jupyter Notebooks* et de la bibliothèque *Rubix*, on peut écrire des runbooks exécutables.