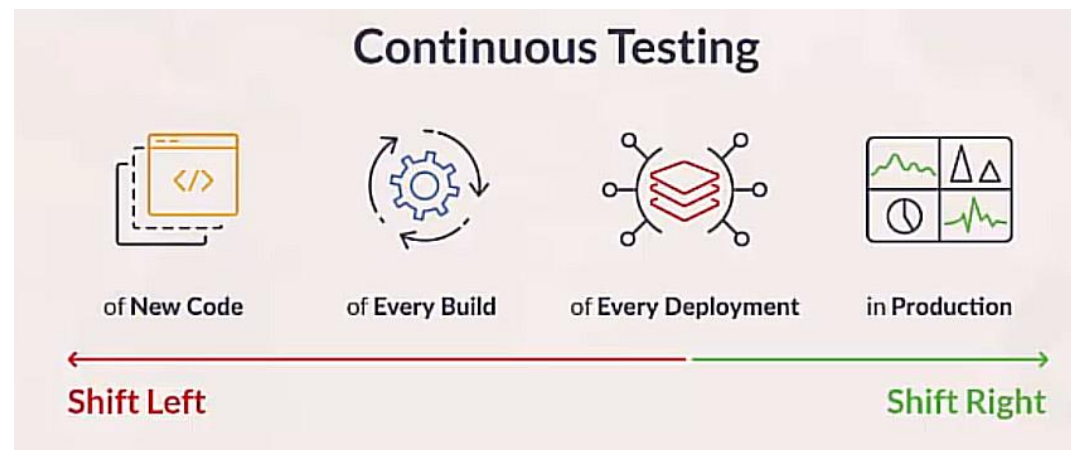

CI/CD



CLEF DE LA PRATIQUE DU DEVOPS

CONTINUOUS TESTING

- **Continuous testing** : process d'exécution automatique des test intégré à une pipeline de déploiement. Permet d'obtenir immédiatement des feedback sur les risques encouru par l'entreprise avec une release
- Intègre le concept de « **shift left** » de la deuxième méthode
- Permet de détecter les problèmes et de les régler en amont de la production
- Les tests automatisés sont critiques au DevOps. Pour les créer, il est nécessaire de préparer des plans de tests en amont
- Les tests continus ont été **proposés à l'origine comme un moyen de réduire le temps d'attente** pour les commentaires des développeurs en introduisant des tests déclenchés par l'environnement de développement ainsi que des tests déclenchés par les développeurs/testeurs plus traditionnels.



CLEF DE LA PRATIQUE DU DEVOPS

CONTINUOUS TESTING

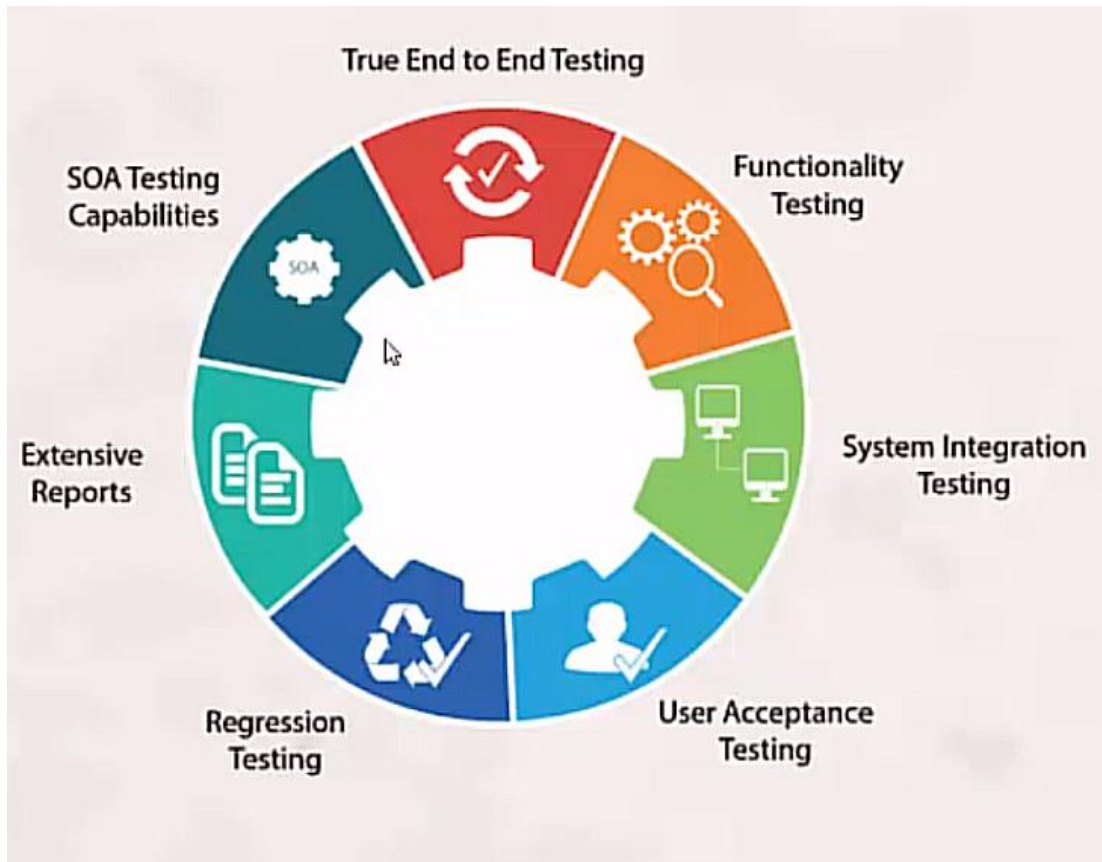
Concepts clés

- La séquence des tests et la nécessité de tests automatisés et continus sont essentielles pour DevOps ; les plans de test et l'automatisation sont indispensables.
- Passez un peu de temps à discuter des exigences fonctionnelles et non fonctionnelles
- Bien que le terme non fonctionnel puisse sembler moins important, il s'agit des exigences qui sous-tendent que les produits doivent être testés au cours du développement et au-delà
- Prêtez une attention particulière au concept croissant de "**shift left**", où les tests, la sécurité, la conformité et les autres exigences fonctionnelles et non fonctionnelles sont testés au cours des processus de développement

CLEF DE LA PRATIQUE DU DEVOPS

CONTINUOUS TESTING

■ Tests fonctionnels

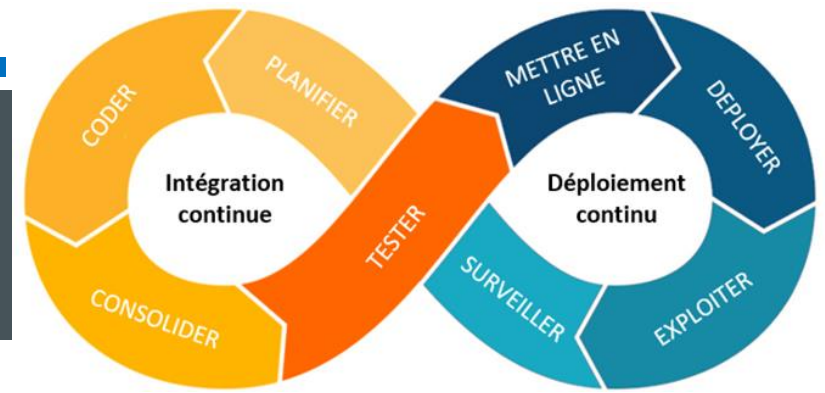


■ Tests non fonctionnels



PRATIQUE DU DEVOPS

INTÉGRATION CONTINUE



- **Continuous integration (CI)** : Pratique de développement qui demande aux développeurs de commit leur code dans un référentiel (repository) partagé, au minimum tous les jours
- Chacun des ajout à ce référentiel doit être validé par
 - Un build automatique
 - Des tests unitaire, d'intégration et d'acceptation automatique
- Nécessite des référentiels de contrôle de version (version control repositories) et des serveurs de CI pour collecter, construire et tester ensemble le code commité.

Avec l'intégration continue, les **développeurs fusionnent aussi souvent que possible** leurs changements de code vers la branche principale.

- Dès qu'une modification est validée, des **processus automatisés de build et de test sont exécutés pour valider l'exactitude de la modification.**
- Les **défauts sont détectés le plus tôt possible** dans le cycle de développement – là où leur **impact et leur coût sont les plus faibles** : un « **shift left** » des défauts.
- **L'automatisation des tests est la clé de l'intégration continue** pour s'assurer que les nouveaux commits ne corrompent pas l'application lorsqu'ils sont intégrés dans la branche principale.

PRATIQUE DU DEVOPS

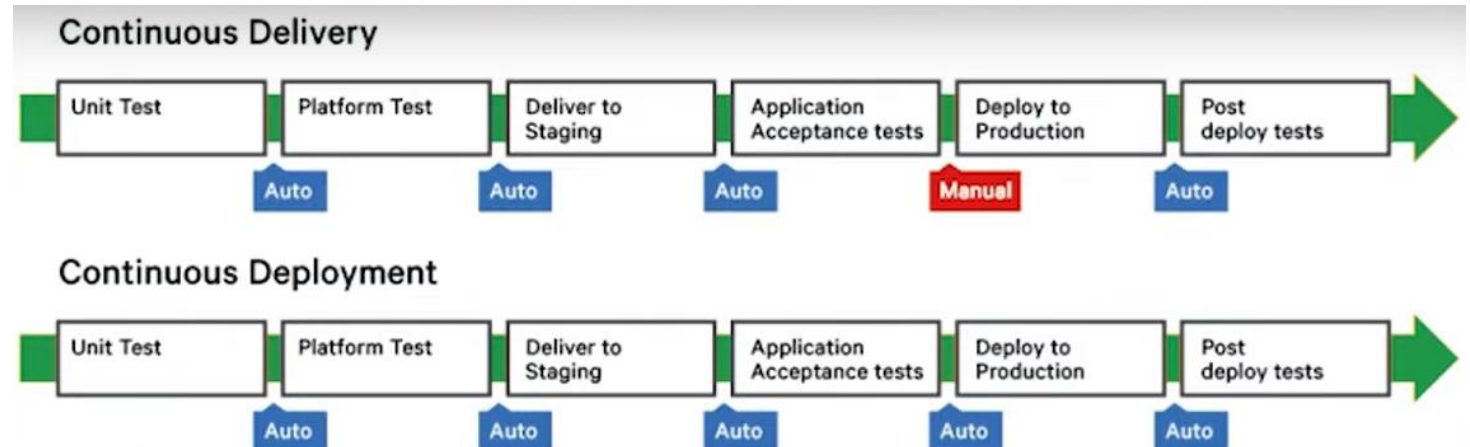
LIVRAISON CONTINUE

- **La livraison continue (CD)** : est une extension de CI, où le logiciel est conçu, configuré et paramétré de manière à ce qu'il puisse être mis en production automatiquement et à tout moment.
- C'est une méthodologie qui consiste à **s'assurer que les logiciels sont toujours dans un état de release** contrôlé à travers son cycle de vie
 - Suite aux tests positifs de l'intégration continue, les releases sont **automatiquement déposés** dans un référentiel logiciel (repository)
 - S'appuie sur une pipeline de déploiement qui permet de **déployer à la demande** en production via un « **push-button** »
 - Réduit le coût, le temps et les risques de livrer des modifications incrémental,

PRATIQUE DU DEVOPS

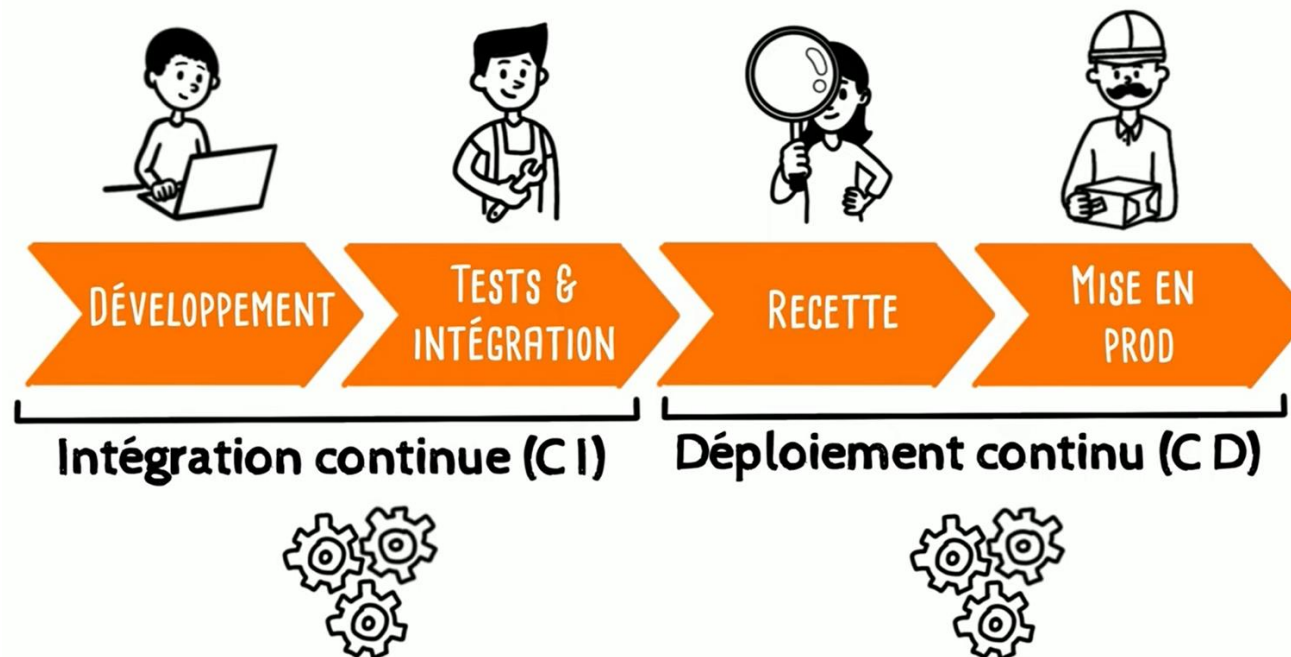
DÉPLOIEMENT CONTINU

- **Continuous deployment (CD)** : Le déploiement continu est la méthodologie de création d'un cadre de test robuste et de sortie automatique de votre code lorsque les tests réussissent. Il va encore plus loin que la livraison continue, **en orchestrant automatiquement le déploiement des applications au client à chaque changement.**
- N'oubliez pas que les **grands changements nécessitent de petits changements.**
- Un déploiement continu correct est difficile car il est difficile de savoir si vous avez **une couverture de test exhaustive.** Les équipes ont généralement besoin d'une certaine marge de manœuvre entre le développement et les déploiements, en particulier pour les modifications sensibles telles que les migrations de bases de données.
- Le déploiement continu **encourage également d'autres types d'actions continues**, comme les **analyses**, les **rapports d'erreurs** et les **tests.**
- En **accélérant la boucle de retours clients**, les gains peuvent être énormes – en termes de **qualité logicielle**, de **délais de projet**, de **résultats** et de **coûts de développement.**

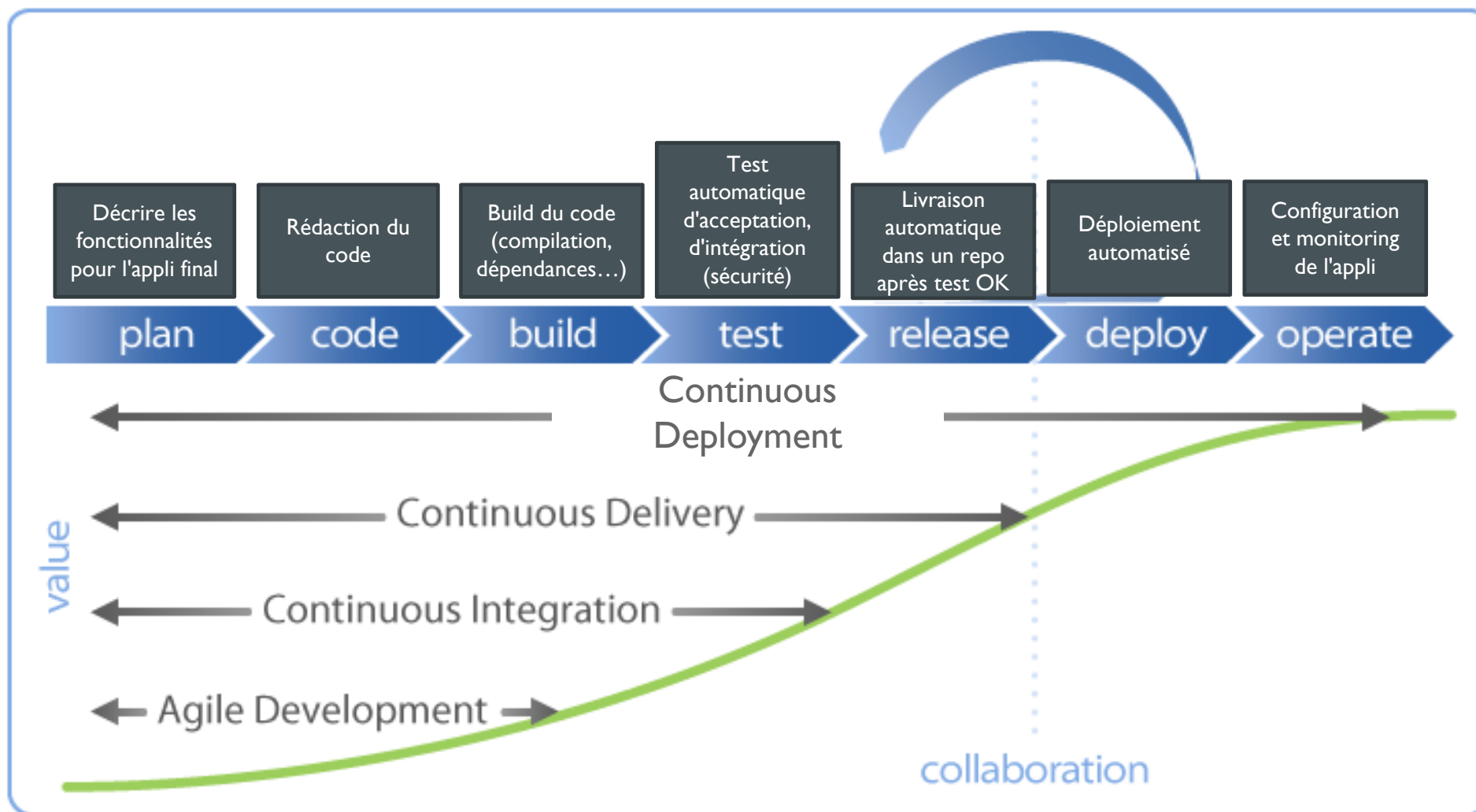


PROCESSUS D'AUTOMATISATION CI/CD

- CI/CD, ou **Intégration Continue/Livraison et Déploiement Continu**, met l'accent sur la **mise en production rapide de petits changements incrémentiels** et l'utilisation de **l'automatisation tout au long** du processus du développement. Au cœur de DevOps, l'outillage CI/CD est la clé de son succès.



PROCESSUS D'AUTOMATISATION CI/CD



PRATIQUE DU DEVOPS

SITE RELIABILITY ENGINEERING (SRE)

- En tant que **discipline, le SRE** se concentre sur l'amélioration de la fiabilité des systèmes logiciels dans des catégories clés telles que la disponibilité, les performances, la latence, l'efficacité, la capacité et la réponse aux incidents.
- Les pratiques SRE sont essentielles pour respecter les accords de niveau de service en temps réel et pour répondre aux exigences d'intégration continue/de livraison continue (CI/CD) des équipes DevOps et DevSecOps.
- Le DevOps est un aspect plutôt culturel, attribué à une équipe, automatise et le processus de mise en production. Tandis que le SRE améliore les opérations **une fois que le code est déployé en production.**
- **Se concentre sur opérations et le maintien de services hautement disponibles.**
- Le professionnel qui assume ce rôle doit être un ingénieur logiciel et **utiliser ces compétences pour automatiser son chemin vers la haute disponibilité.**
- **En bref, DevOps amène le code en production, tandis que SRE veille à ce qu'il fonctionne correctement une fois en production.**

PRATIQUE DU DEVOPS

SITE RELIABILITY ENGINEERING (SRE)

- Les SRE aident à résoudre les principaux problèmes des responsables informatiques dans une culture DevSecOps.
- Dans ce bras de fer entre les rôles, la tâche principale du SRE est de s'assurer que les sites et les services de l'entreprise :
 - Offrent des performances, un temps de fonctionnement et une disponibilité constants.
 - Assurent la sécurité et la redondance du site.
 - Développent des moyens de détection précoce des problèmes.
 - Utiliser un cadre de mesure pour suivre la fiabilité.
 - Utiliser l'automatisation pour réduire la gestion manuelle.
 - Compréhension globale des besoins actuels et futurs.

À la fois administrateur de systèmes et développeur, le SRE doit comprendre les deux côtés de l'équation et les conflits qui surviennent pendant le développement et l'exploitation. La capacité à dépanner les processus dus à des méthodologies différentes fait partie intégrante du travail du SRE, qui utilise son point de vue pour créer un système opérationnel équilibré.

PRATIQUE DU DEVOPS

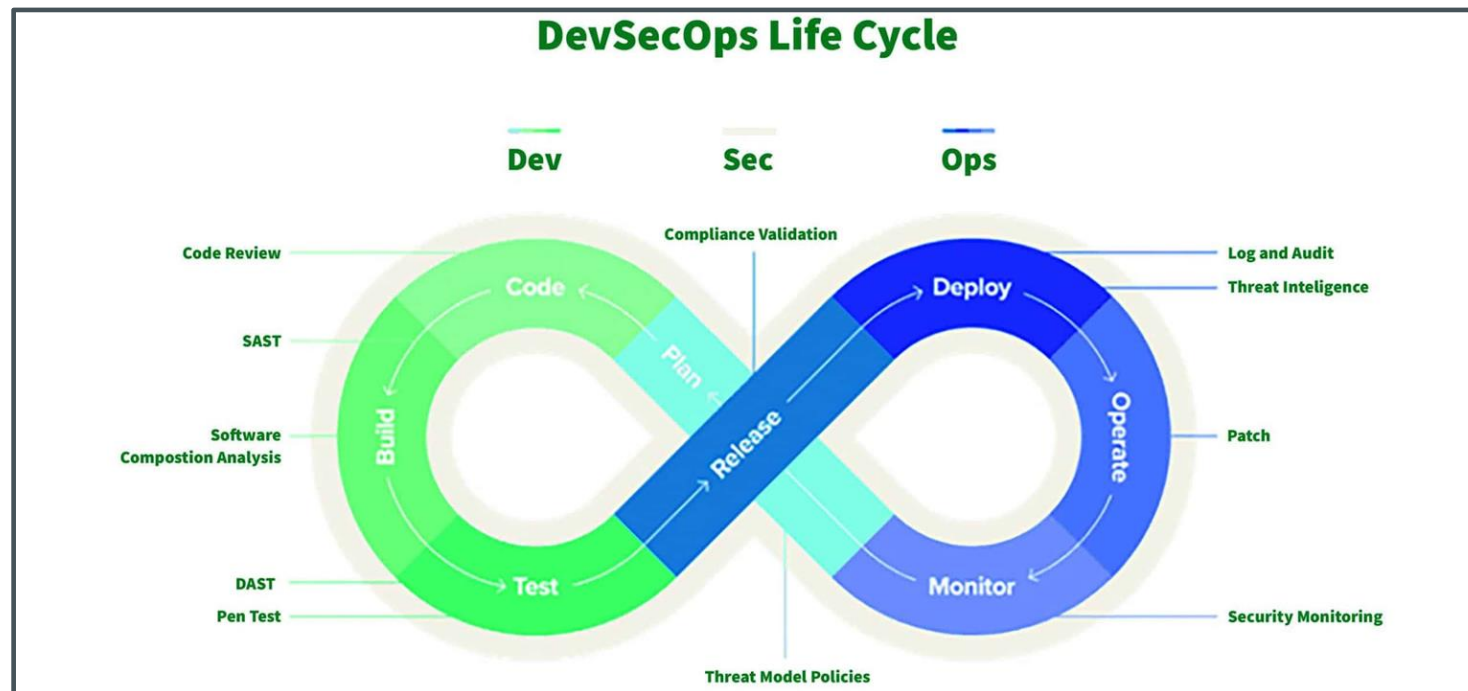
SITE RELIABILITY ENGINEERING (SRE) - LE BUDGET D'ERREUR

- Un objectif de niveau de service fixe la référence pour la fiabilité d'un système pour les utilisateurs finaux. **Le budget d'erreur est établi sur la base de l'objectif de niveau de service.**
- Les développeurs peuvent **construire et mettre à jour leurs produits et les déployer en respectant ce budget d'erreurs**. Tant que le produit fonctionne avec peu d'erreurs négligeables, ils sont libres **d'ajouter de nouvelles fonctionnalités à un rythme qui convient à l'entreprise**.
- À l'inverse, **lorsque le budget d'erreurs est dépassé**, les autres mises à jour ou lancements sont **gelés jusqu'à ce que le nombre d'erreurs soit réduit**, et tous les efforts des développeurs sont **concentrés sur les corrections** nécessaires.
- Les développeurs sont ainsi **incités à réduire les erreurs et à améliorer la fiabilité** à toutes les étapes du cycle de vie d'un produit.

PRATIQUE DU DEVOPS

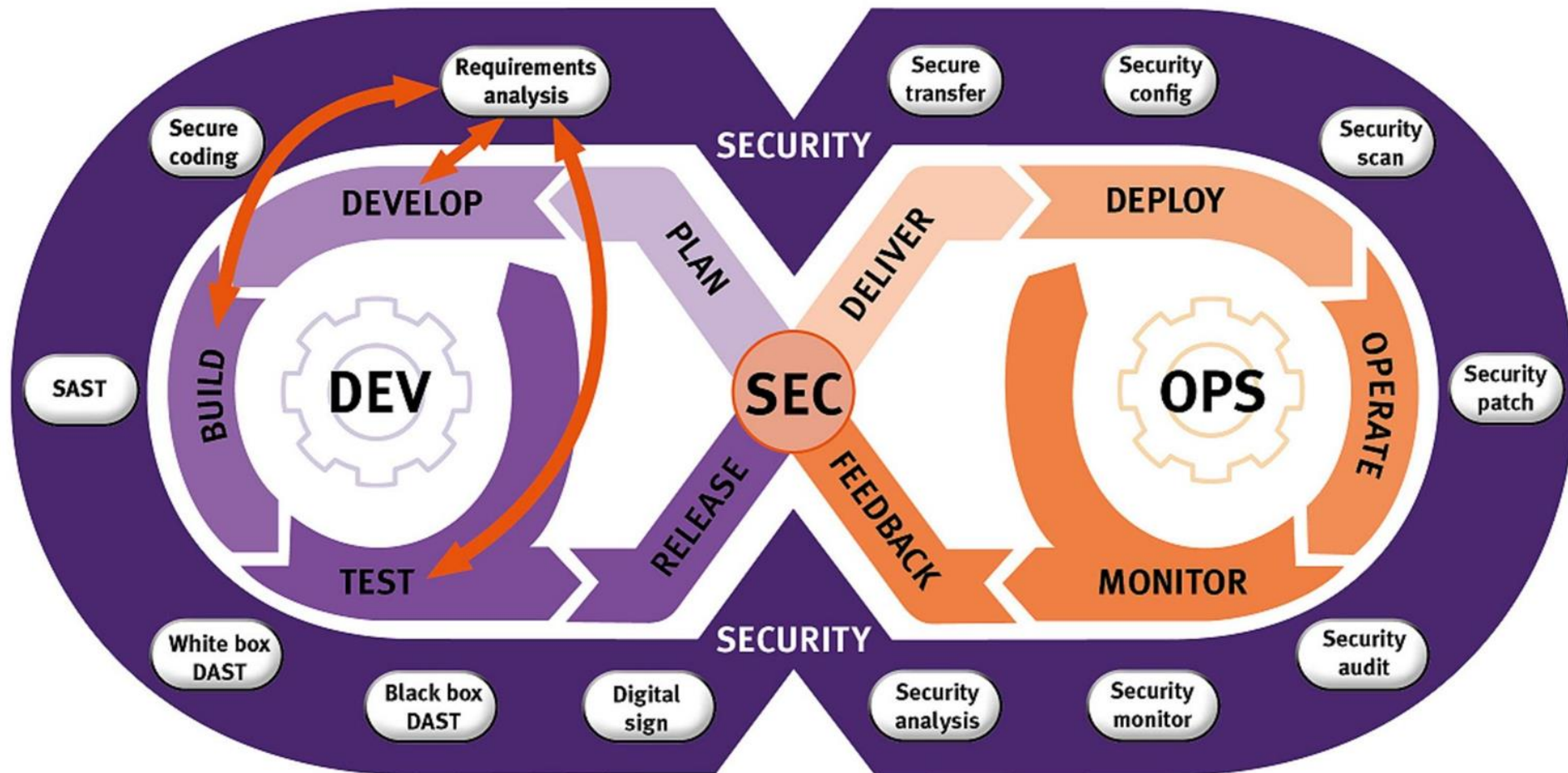
DEVSECOPS

- Tout le monde est responsable de la sécurité
- Intégrer de la sécurité dans chacune des étapes du cycle de vie des applications
- Ajouter des checks de sécurité automatique dans son workflow
- Briser les contraintes de sécurité en les intégrant dès le développement



PRATIQUE DU DEVOPS

DEVSECOPS



PRATIQUE DU DEVOPS

CHATOPS










- ChatOps: Approche de communication permettant aux équipes de d'échanger et collaborer en temps réel
- Ces outils peuvent aider à :
 - Echanger pour résoudre des problèmes en direct et potentiellement réduire de MTTR
 - Créer un historique de communication partagé et indexé
 - Être aidé pour du code
 - Recevoir des réponses rapides sur des sujets d'infra
 -



PRATIQUE DU DEVOPS

KANBAN

- ▢ Kanban: Méthode de travail qui permet de faire avancer le workflow au travers d'un process à un rythme gérable
- ▢ Mot d'origine japonaise, il se compose de « kan » qui signifie « visuel » et « ban » qui veut dire « tableau »

À FAIRE	EN COURS	EN ATTENTE	TERMINÉ
  	 	 	 

PRATIQUE DU DEVOPS

KANBAN

- Il permet de visualiser et manager les workflows
- Permet aux équipes de tirer les tâches quand ils sont prêts pour les faire
- Rend plus facile le travail collaboratif
- Il est possible de mesurer la vitesse de travail d'une équipe
- Réduire les temps mort dans les process
- Rend le travail visible
- Pose les besoin explicitement
- Limite la quantité total de travail à la capacité de l'équipe

CULTURE, COMPORTEMENT ET MODÈLE OPÉRATIONNEL

DÉFINITION DE LA CULTURE

- La culture est un terme vaste. Il n'existe pas de définition unique,
- On peut voir la culture comme les valeurs et les comportements qui contribuent à l'environnement social et psychologique unique de l'entreprise.
- On ne peut pas directement changer la Culture. Mais on peut changer les comportements, et les comportements changent la Culture.

CULTURE, COMPORTEMENT ET MODÈLE OPÉRATIONNEL

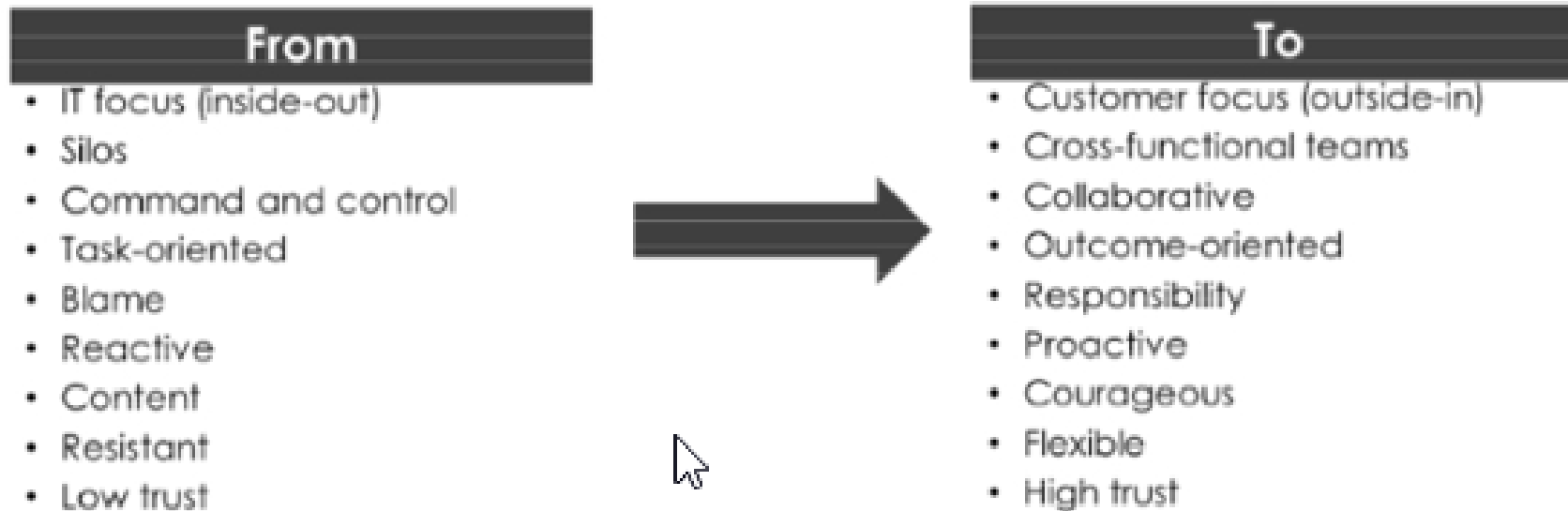
DÉFINITION DE LA CULTURE

Caractéristiques de la culture DevOps

- ▣ Vision et objectifs partagés
- ▣ Communication ouverte, honnête et bidirectionnelle
- ▣ Collaboration
- ▣ Fierté du travail accompli
- ▣ Le respect
- ▣ La confiance
- ▣ La transparence
- ▣ L'amélioration continue
- ▣ Expérimentation
- ▣ Prise de risque intelligente
- ▣ Apprendre et s'entraîner
- ▣ Axé sur les données
- ▣ Sécurité
- ▣ Réflexion
- ▣ Reconnaissance

CULTURE, COMPORTEMENT ET MODÈLE OPÉRATIONNEL

MODÈLES DE COMPORTEMENT



- Le changement de Culture est un processus qui prend du temps. Il doit se faire progressivement et réalisé à un rythme réaliste.
- En général, ce changement prend plus de temps qu'on ne prévoit.
- Vous ne pouvez pas changer les gens. Seul eux le peuvent.
- L'implication de tous est important.

CULTURE, COMPORTEMENT ET MODÈLE OPÉRATIONNEL

MODÈLES OPÉRATIONNEL

Comment favoriser de nouveaux comportements ?

- Améliorer les pratiques de communication et de collaboration et les outils partagés.
- Créer un vocabulaire commun
- Shadowing
- Expériences d'immersion et simulations
- Communautés de pratique
- Journées DevOps
- Hackathons
- Partage d'idées, d'histoires, de résolution de problèmes dans le style des réseaux sociaux

CULTURE, COMPORTEMENT ET MODÈLE OPÉRATIONNEL

LA DETTE CULTURELLE

- Le DevOps aide à surpasser la dette culturel
- La dette culturel est présente quand la considération de la culture est mise au second plan, en faveur à la croissance et l'innovation,
- On ressent la dette quand :
 - Des silos sont présents et impénétrable
 - Ce qui mène à des informations cachés
 - Les sociétés recrutent les mauvais profils
 - Les employés ne sont pas responsabilisés
 - Les employés n'ont pas le sentiments que leur contribution compte ou n'est pas reconnue
 - On ne laisse pas le temps ou les ressources nécessaire pour s'améliorer et expérimenter
 - Il n'existe pas de boucle de feedback ou elle sont non constructive

CULTURE, COMPORTEMENT ET MODÈLE OPÉRATIONNEL

LA DETTE CULTURELLE

High Trust vs. Low Trust



- Les organisations où règne la confiance encouragent :
 - la bonne circulation des informations
 - La collaboration inter-équipes
 - Le partage des responsabilités
 - L'apprentissage à partir des échecs nouvelles idées

AUTOMATISATION & ARCHITECTURE DES DEVOPS TOOLCHAIN

CI/CD

- Le DevOps s'appuie et se construit sur les pratiques de « l'infrastructure as code »
- L'infrastructure as code permet de reconstruire l'ensemble de son infrastructure à partir d'un dépôt de code source, des sauvegardes et des machines bare metal
- Créer des environnements de travail identiques,

AUTOMATISATION & ARCHITECTURE DES DEVOPS TOOLCHAIN

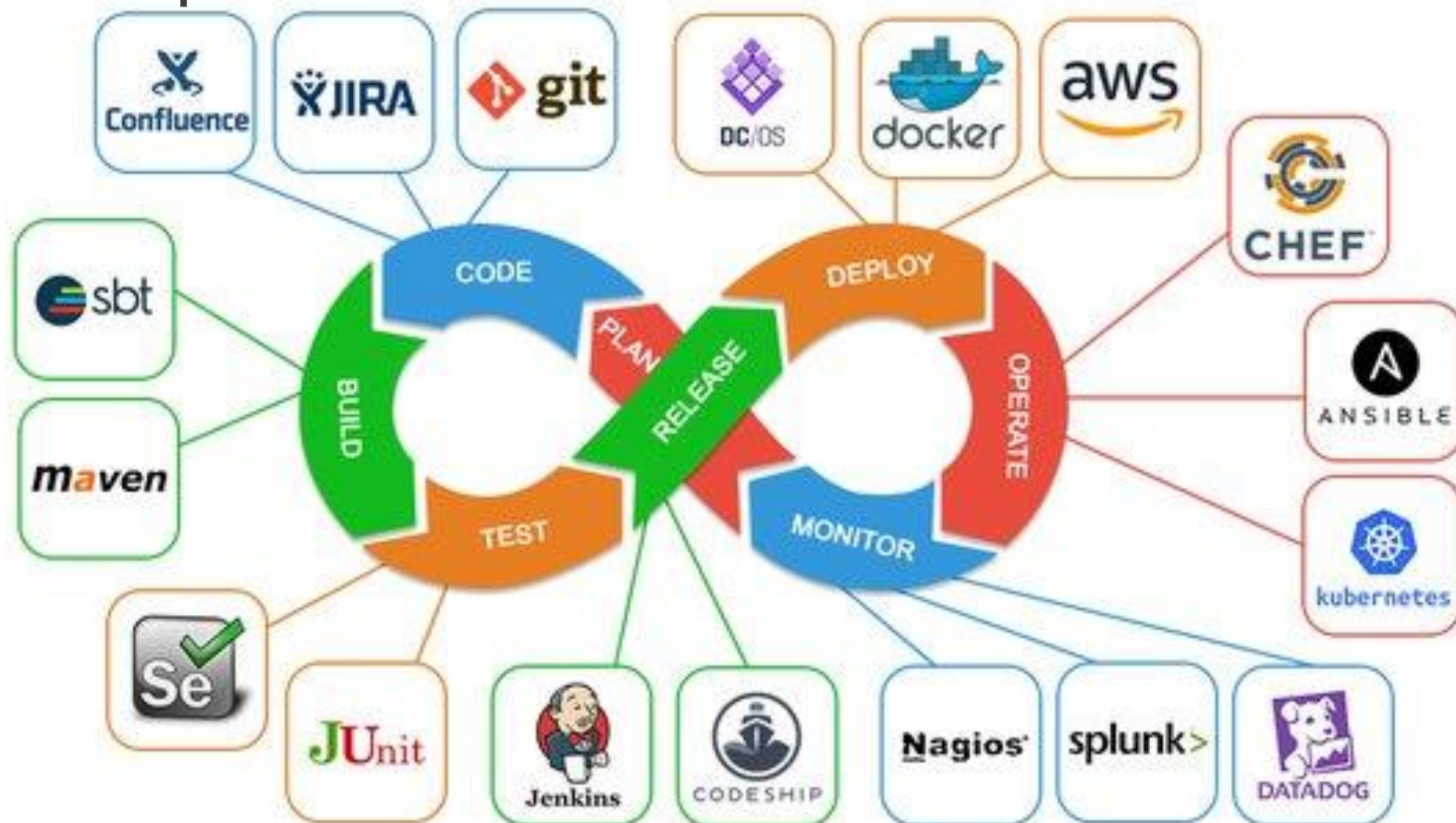
CLOUD, CONTAINER ET MICROSERVICES

- ❑ Le Cloud Computing est une pratique consistant à utiliser des serveurs qui sont hébergés sur Internet pour y exécuter des applications/charges de travail
- ❑ Amazon Web Services (AWS), Microsoft Azure et Google Cloud Platform (GCP) sont les fournisseurs Cloud les plus répandues
- ❑ Docker est un outil permettant de concevoir, tester et déployer des application rapidement en utilisant des conteneurs.
- ❑ Kubernetes est un outil OpenSource d'orchestration des conteneurs qui automatise les processus de déploiement, gestion et mise à l'échelle des applications conteneurisées
- ❑ Une architecture de micro-services se différencie d'une approche monolithique classique par le fait qu'elle décompose une application pour en isoler les fonctions clés
- ❑ Les micro-services peuvent communiquer entre elles à l'aide d'API indépendantes de tout langage

AUTOMATISATION & ARCHITECTURE DES DEVOPS TOOLCHAIN

DEVOPS TOOLCHAIN

Tool chain & exemple d'outils



AUTOMATISATION & ARCHITECTURE DES DEVOPS TOOLCHAIN

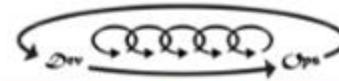
DEVOPS TOOLCHAIN

- Les DevOps toolchain sont composés des outils qui sont nécessaires pour supporter de l'intégration continue, la livraison continu et le déploiement continu
- Chacun des éléments de la toolchain sert un besoin spécifique. Pour que ces éléments soient interopérables, il est nécessaire de concevoir une architecture en premier
- Les applications au sein de la toolchain doivent être connectés ensemble. De préférence en API
- Il n'est pas nécessaire que tous les outils viennent du même éditeur.
- Une pipeline de déploiement est un processus automatisé qui gère les changements. De l'enregistrement du besoin à la livraison
- Ces pipelines s'assurent que le code présente dans les contrôleurs de sources est automatiquement conçu et tester dans des environnement similaire à la production

MESURES ET MÉTRIQUES

L'IMPORTANCE DES MESURES

The Importance of Measurement

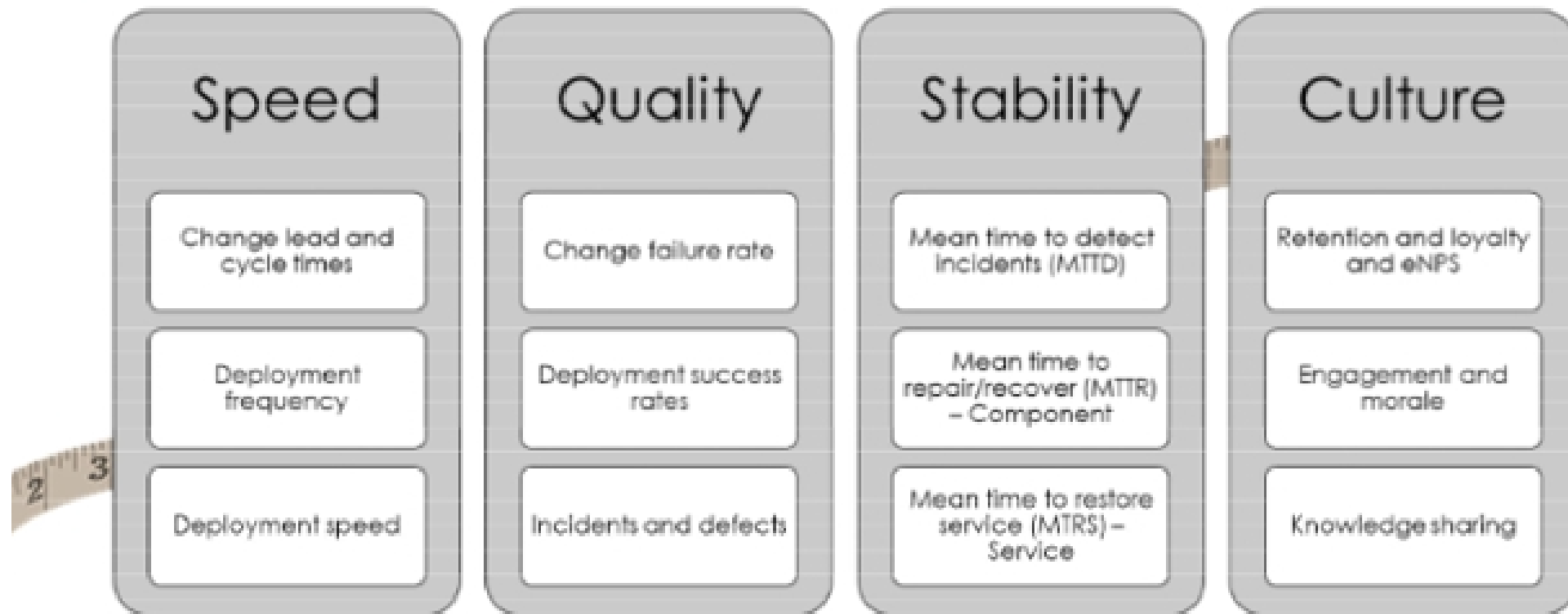


The First Way	The Second Way	The Third Way
Flow	Feedback	Continuous Experimentation & Learning
<ul style="list-style-type: none">• Change lead time• Change cycle time• Time to value• Value realisation	<ul style="list-style-type: none">• Build/test results• Change fail rate• Monitoring• % rework / complete & accurate	<ul style="list-style-type: none">• Hypothesis log• Time allocated• Time spent• Mastery achieved and reported
Measurements allow us to find constraints and justify their removal and monitor improvement	Evidence builds trust and earns the right to do more – placing the bets in experimentation	Hypotheses need quantifiable outcomes to determine next experiment

MESURES ET MÉTRIQUES

LES MÉTRIQUES DEVOPS

Measuring Success



MESURES ET MÉTRIQUES

LES MÉTRIQUES DEVOPS

- Les 4 métriques thématique du DevOps sont : la rapidité, la qualité, la stabilité et la culture
- Rapidité
 - Représente globalement la vitesse des process
 - Le change Lead and cycle time est l'une des métriques les plus importante car elle représente ce que voit le client
- Qualité
 - Se concentre sur la mesure des réussites et échecs
 - Le change failure rate est important pour mesurer la fiabilité et la stabilité
- Stabilité
 - Se focalise sur les capacités à revenir à un état fonctionnel
 - Attention à ne pas confondre le MTTR (composant/fonctionnalité) avec le MTRS (le service complet)
- Culture
 - Les mesures concernent particulièrement le partage de connaissance et l'engagement de l'équipe dans les pratiques DevOps

MESURES ET MÉTRIQUES

LEAD/CYCLE TIME

Change Lead/Cycle Time

Lead Time	Cycle Time
The total elapsed time from the point when a user story enters the backlog, until the time it is completed – including the time spent waiting in a backlog.	The time it takes for a story to go from being “In progress” to Done.

Lead Time minus Cycle Time is Wait Time

PARTAGER, OBSERVER & ÉVOLUER

UN APPRENTISSAGE IMMERSIF ET EXPÉRIMENTAL

- Certaines entreprises propose des journées dédiés à l'échange sur le DevOps (DevOps Days).
 - Ce sont des journées conférences autour du DevOps
 - Généralement composé de 30min de présentation, 5 minutes que Q/R avec le conférencier suivi d'une pause en Open Space pour échanger avec tout le monde
 - Idéal pour apprendre, partager, discuter et apporter de nouveaux point de vue
- En plus petit comité, l'apprentissage interactif contribue à encourager le partage des outils, des connaissances, des découvertes et des leçons apprises. On peut y retrouver :
 - des jeux,
 - des hackatons,
 - des simulations

PARTAGER, OBSERVER & ÉVOLUER

UNE ÉQUIPE DEVOPS

- Se développer au travers des concepts d'une équipe Agile ou Scrum
- Eclater les silos en intégrant les compétences de développement et d'exploitation dans un seul groupe holistique.
- Peut être lié à un produit spécifique temporairement ou y être dédiées
- Peut évoluer pour fournir des services partagés / transversale.
- Ont des responsabilités partagées.
- Devrait adhérer aux normes définies pour le développement, l'automatisation, le risque et la conformité qui s'appliquent à toutes les équipes DevOps.
- Il n'existe pas de structure 'parfaite' pour une équipe DevOps

PARTAGER, OBSERVER & ÉVOLUER

UN CHANGEMENT ÉVOLUTIF

- il s'agit d'un parcours, pas d'une solution miracle, et les dirigeants doivent éviter de se laisser entraîner par la paralysie de l'analyse. Commencez à effectuer les changements, obtenez les victoires et laissez l'organisation évoluer.
- Les facteurs de réussite sont des actions sur le long terme:
 - Engagement de la direction en faveur d'un changement de culture.
 - Création d'une culture de collaboration et d'apprentissage.
 - Formation et amélioration continue des compétences.
 - Valeurs et vocabulaire communs.
 - Une ingénierie systèmes qui englobe le dev et les ops.
 - Des métriques significatives.
 - Un équilibre entre l'automatisation et l'interaction humaine.
 - L'application de méthodes agiles et allégées.
 - Une communication ouverte et fréquente.

LES OUTILS

1OsGI
GitLab

3FmGh
GitHub

4EnDt
Datical

11OsSv
Subversion

12EnDb
DBMaestro

19EnCw
ISPW

20EnDp
Delphix

21OsJn
Jenkins

22FmCs
Codeship

23OsFn
FitNesse

24FrJu
JUnit

25FrKa
Karma

26FmSu
SoapUI

27EnCh
Chef

28FrTf
Terraform

29EnXld
XebiaLabs
XL Deploy

30EnUd
UrbanCode
Deploy

31OsKu
Kubernetes

32FmCc
CA CD
Director

33EnPr
Plutora
Release

34PdAl
Alibaba
Cloud

35OsOs
OpenStack

36OsPs
Prometheus

37OsAt
Artifactory

38EnRg
Redgate

39PdBa
Bamboo

40FmVs
VSTS

41FrSe
Selenium

42FrJm
JMeter

43OsJa
Jasmine

44PdSl
Sauce Labs

45OsAn
Ansible

46OsRu
Rudder

47EnOc
Octopus
Deploy

48OsGo
GoCD

49OsMs
Mesos

50PdGke
GKE

51FmOm
OpenMake

52PdCp
AWS
CodePipeline

53OsCy
Cloud
Foundry

54EnIt
ITRS

55OsNx
Nexus

56OsFw
Flyway

57OsTr
Travis CI

58FmTc
TeamCity

59OsGa
Gatling

60FrTn
TestNG

61FmTt
Tricentis
Tosca

62PdPe
Perfecto

63EnPu
Puppet

64OsPa
Packer

65FmCd
AWS
CodeDeploy

66EnEc
ElectricCloud

67OsRa
Rancher

68PdAks
AKS

69OsRk
Rkt

70OsSp
Spinnaker

71PdIr
Iron.io

72PdMg
Moogsoft

73FmBb
BitBucket

74EnPf
Perforce
HelixCore

75FmCr
Circle CI

76PdCb
AWS
CodeBuild

77FrCu
Cucumber

78OsMc
Mocha

79OsLo
Locust.io

80EnMf
Micro Focus
UFT

81OsSl
Salt

82OsCe
CFEngine

83EnEb
ElasticBox

84EnCa
CA Automic

85EnDe
Docker
Enterprise

86PdAe
AWS ECS

87FmCf
Codefresh

88OsHm
Helm

89OsAw
Apache
OpenWhisk

90OsLs
Logstash

2EnSp
Splunk

5EnXlr
XebiaLabs
XL Release

6FmAws
AWS

7PdAz
Azure

8EnGc
Google
Cloud

9EnOp
OpenShift

10FmSi
Sumo Logic

13OsDk
Docker

14EnUr
UrbanCode
Release

15PdAf
Azure
Functions

16PdLd
Lambda

17EnIc
IBM Cloud

18OsFd
Fluentd

EMBED

OsOpen Source

FrFree

FmFreemium

PdPaid

EnEnterprise

Source Control Mgmt.

Database Automation

Continuous Integration

Testing

Configuration

Deployment

Containers

Release Orchestration

Cloud

AI/Ops

Analytics

Monitoring

Security

Collaboration

 **XebiaLabs**
Enterprise DevOps

 Follow @xebialabs

Publication Guidelines

91EnXliXebiaLabs XL Impact	92OsKiKibana	93FmNrNew Relic	94EnDtDynatrace	95EnDdDatadog	96FmAdAppDynamic	97OsEiElasticSearch	98OsNiNagios	99OsZbZabbix	100EnZnZenoss	101EnCxCheckmarx SAST	102EnSgSignal Sciences	103EnBdBlackDuck	104OsSrSonarQube	105OsHvHashiCorp Vault
106EnSwServiceNow	107PdJrJira	108FmTlTrello	109FmSlSlack	110FmStStride	111EnCnCollabNet VersionOne	112EnRyRemedy	113EnAcAgile Central	114PdOgOpsGenie	115PdPdPagerduty	116OsSnShort	117OsTwTripwire	118EnCkCyberArk Conjur	119EnVcVeracode	120EnFfFortify SCA

Don't see your tool listed?

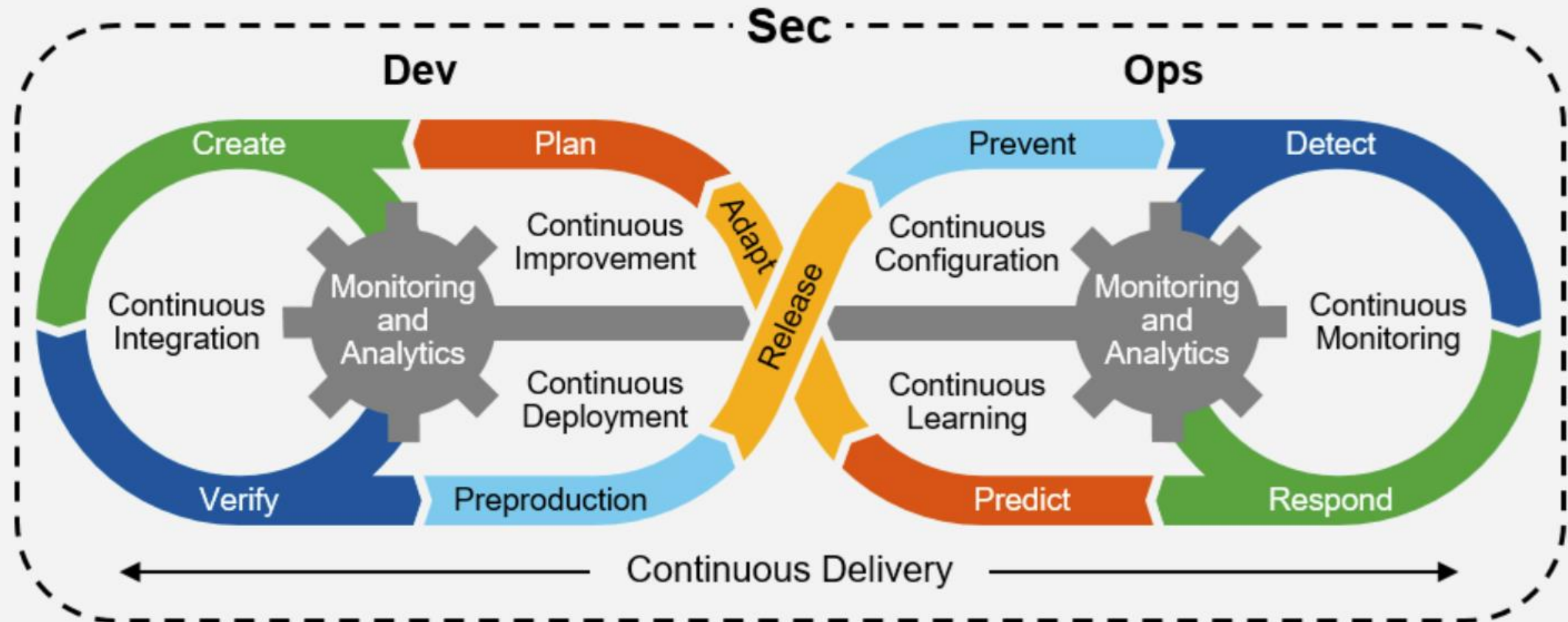
SUBMIT A TOOL

[DOWNLOAD THE PERIODIC TABLE](#)

PRATIQUE DU DEVOPS

DEVSECOPS

DevSecOps: Seamlessly Integrating Security Throughout DevOps



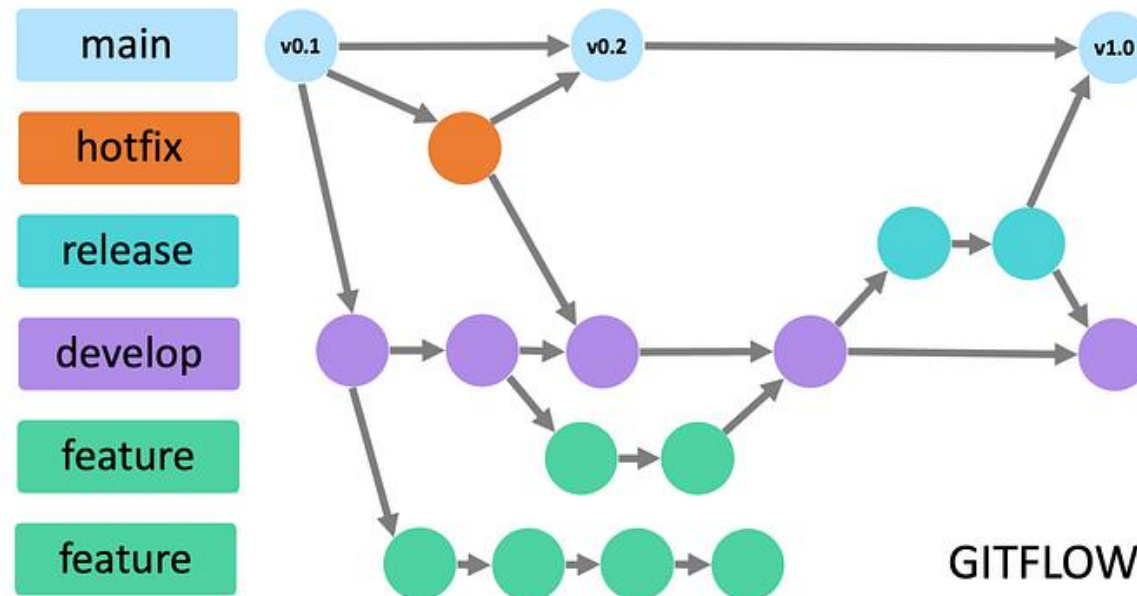
GITLAB

- `git init` : initialise un nouveau dépôt Git dans le répertoire courant.
- `git clone` : clone un dépôt distant dans un nouveau répertoire.
- `git add` : ajoute des fichiers à l'index (étape intermédiaire avant de les valider).
- `git commit` : valide les modifications ajoutées à l'index et les enregistre dans l'historique du dépôt.
- `git push` : envoie les commits vers un dépôt distant.
- `git pull` : récupère les commits d'un dépôt distant et les fusionne dans le dépôt local.
- `git fetch` : récupère les commits d'un dépôt distant sans les fusionner.
- `git merge` : fusionne les commits d'une autre branche dans la branche courante.
- `git branch` : gère les branches dans le dépôt.
- `git checkout` : permet de naviguer entre les branches et de restaurer des fichiers.
- `git stash` : enregistre temporairement des modifications non validées.
- `git log` : affiche l'historique des commits d'un dépôt.
- `git show` : affiche les détails d'un commit spécifique.
- `git diff` : affiche les différences entre les commits, les fichiers ou les branches.

Il existe également de nombreuses options et arguments qui peuvent être utilisés avec ces commandes pour personnaliser leur comportement.

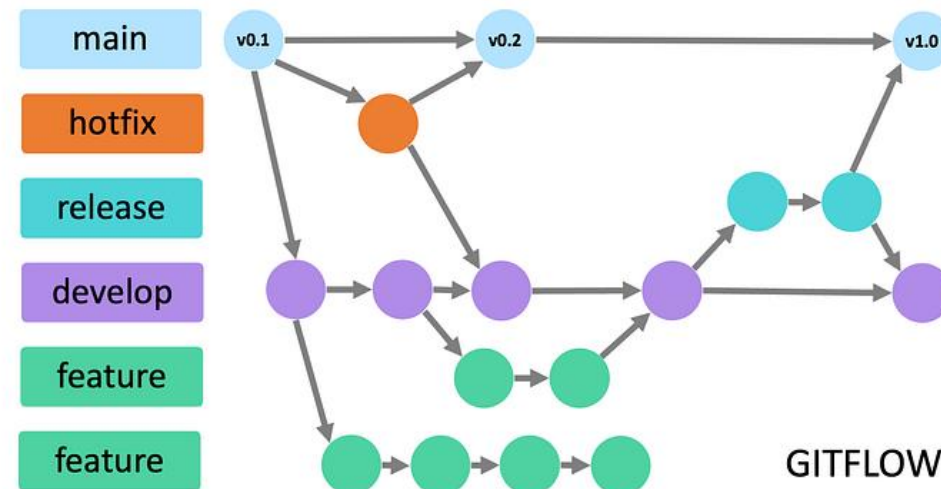
GIT FLOW

- Git Flow est un modèle de branchement qui fournit une approche structurée de la gestion de grands projets avec des versions planifiées.
- Il introduit deux branches à longue durée de vie – develop et main - ainsi que plusieurs branches de soutien.



GIT FLOW

- **Branche de développement (develop)**: branche d'intégration pour les fonctionnalités, contenant le code en cours.
- **Branches de fonctionnalités (features)** : à partir de la branche de développement pour travailler sur de nouvelles fonctionnalités ou des corrections de bogues.
- **Branches de mise en production (release)** : est créée à partir de la branche de développement pour les tests finaux et les corrections de bogues.
- **Branches de correctifs (hotfix)** : résoudre rapidement les problèmes critiques dans la base de code de production. Ils sont créés à partir de la branche master.



AVANTAGES

- **Développement structuré** : Git Flow fournit une structure claire pour la gestion du développement des fonctionnalités, des versions et des correctifs.
- **Flux de travail parallèles** : Différentes équipes peuvent travailler simultanément sur des branches de fonctionnalités distinctes, favorisant ainsi le développement parallèle.
- **Stable Master** : La branche master reste stable et sert de base fiable pour les versions de production.

EXEMPLE

- Une fois que vous avez terminé de développer une fonctionnalité ou de préparer une version, vous devez fusionner (merge) votre branche dans la branche appropriée :
- Fusionner une **branche de fonctionnalité dans develop** : Utilisez une "Pull Request" ou une "Merge Request" pour demander une revue de code avant de fusionner.
- Fusionner une **branche de release dans main et develop** : Assurez-vous que la version est prête à être déployée en production avant de fusionner.
- Fusionner une **branche de hotfix dans main et develop** : Corrigez le bug rapidement et fusionnez la branche dès que possible.

EXEMPLE DE WORKFLOW

- Débuter sur develop
- Créer une branche feature/**
- Développer sur la branche feature/**
- Pousser (push) la branche feature/**
- Créer une Merge/Pull Request

EXEMPLE DE WORKFLOW

- **Revue de code et fusion** : Votre équipe examinera votre code, apportera des commentaires et, si tout est OK, acceptera la Merge/Pull Request, **fusionnant ainsi votre branche feature/** dans develop**.
- **Supprimer la branche feature/****
- **Préparer une release** : créez une **branche release/** à partir de develop**.
- **Finaliser la release sur la branche release/**** : Effectuez les derniers ajustements, tests et corrections de bugs sur cette branche.
- **Fusionner la branche release/** avec main et develop**
- **Créer un tag**
- **Supprimer la branche release/****
- **Cas exceptionnel : Hotfix** : Si un bug critique est découvert en production, **créez une branche hotfix/** à partir de main**, corrigez le bug, puis **fusionnez-la dans main ET dans develop**.

EXEMPLE DE WORKFLOW

```
cd my-go-app/  
git fetch  
git pull  
git checkout -b develop  
git push -u origin develop  
git checkout develop  
git checkout -b feature/new-feature  
# Modification de code ou autre  
git add .  
git commit -m "Ajout de la nouvelle fonctionnalité"  
git push -u origin feature/new-feature  
git checkout develop  
git pull  
git branch -d feature/new-feature  
git checkout develop  
git checkout -b release/1.0.0  
git checkout main  
git merge release/1.0.0  
git checkout develop  
git merge release/1.0.0  
git tag -a v1.0.0 -m "Version 1.0.0"  
git branch -d release/1.0.0  
git push origin v1.0.0
```

1. Débuter sur develop
2. Créer une branche feature/**
3. Développer sur la branche feature/**
4. Pousser (push) la branche feature/**
5. Créer une Merge/Pull Request
6. **Revue de code et fusion** : Votre équipe examinera votre code, apportera des commentaires et, si tout est OK, acceptera la Merge/Pull Request, **fusionnant ainsi votre branche feature/** dans develop.**
7. Supprimer la branche feature/**
8. Préparer une release : créez une branche release/** à partir de develop.
9. **Finaliser la release sur la branche release/**** : Effectuez les derniers ajustements, tests et corrections de bugs sur cette branche.
10. Fusionner la branche release/** avec main et develop
11. Créer un tag
12. Supprimer la branche release/**
13. **Cas exceptionnel : Hotfix** : Si un bug critique est découvert en production, **créez une branche hotfix/** à partir de main**, corrigez le bug, puis **fusionnez-la dans main ET dans develop.**

TP

- Créer un repository avec les recommandations du gitflow précédentes.

QU'EST-CE QU'UN PIPELINE CI/CD?

- Un pipeline est un processus automatisé qui exécute des jobs.
- Il assure l'intégration continue et le déploiement continu (CI/CD).
- Défini via un fichier YAML dans votre dépôt GitLab.
- Les pipelines sont déclenchés par des événements (pushes, merge requests, etc.).

EXEMPLE DE FICHIER .GITLAB-CI.YML

```
# Spécifier l'image Docker de base
```

```
image: alpine:latest
```

```
# Définir les étapes du pipeline
```

```
stages:
```

- build
- test
- deploy

```
# Définir un job pour l'étape de build
```

```
build_job:
```

```
  stage: build
```

```
  script:
```

- echo "Construction de l'application..."
- # Commandes de build

```
# Définir un job pour l'étape de test
```

```
test_job:
```

```
  stage: test
```

```
  script:
```

- echo "Exécution des tests..."
- # Commandes de test

```
# Définir un job pour l'étape de déploiement
```

```
deploy_job:
```

```
  stage: deploy
```

```
  script:
```

- echo "Déploiement de l'application..."
- # Commandes de déploiement

LES STAGES (ÉTAPES)

- Les stages regroupent logiquement les jobs.
- Exécution des jobs dans un ordre prédéterminé.
- Les jobs d'un stage sont exécutés en parallèle par défaut.
- Exemples : build, test, deploy.

EXEMPLE D'UTILISATION DES STAGES - PIPELINE

Un Pipeline est l'ensemble des jobs organisés selon les stages. Il représente le flux complet de l'intégration continue. Visualisation : GitLab offre une interface pour visualiser les pipelines, permettant de suivre l'état de chaque job.

stages:

- build
- test
- deploy

build:

```
stage: build
script:
  - echo "Construction du projet"
```

test:

```
stage: test
script:
  - echo "Test des fonctionnalités"
```

deploy:

```
stage: deploy
script:
  - echo "Déploiement en production"
```

[Pipeline]

```
|
|-- [ Stage: Build ]
|      |-- Job: build_app
|
|-- [ Stage: Test ]
|      |-- Job: unit_tests
|
|-- [ Stage: Deploy ]
|      |-- Job: deploy_app
```

TEMPLATE DE PIPELINE

stages: *# List of stages for jobs, and their order of execution*

- build
- test
- deploy

build-job: *# This job runs in the build stage, which runs first.*

stage: build

script:

- echo "Compiling the code..."
- echo "Compile complete."

unit-test-job: *# This job runs in the test stage.*

stage: test *# It only starts when the job in the build stage completes successfully.*

script:

- echo "Running unit tests... This will take about 60 seconds."
- sleep 60
- echo "Code coverage is 90%"

lint-test-job: *# This job also runs in the test stage.*

stage: test *# It can run at the same time as unit-test-job (in parallel).*

script:

- echo "Linting code... This will take about 10 seconds."
- sleep 10
- echo "No lint issues found."

deploy-job: *# This job runs in the deploy stage.*

stage: deploy *# It only runs when *both* jobs in the test stage complete successfully.*

environment: production

script:

- echo "Deploying application..."
- echo "Application successfully deployed."

LES BALISES - RÉSUMÉ

- image
 - Une image de docker qui est adapté à GitLab CI.
- cache
 - Pour éviter d'attendre entre chaque test, on peut mettre en cache le composer et Phpunit afin d'accélérer le processus.
- services
 - L'ensemble des services qui vous permettra de simuler un environnement, une base de données, un elasticsearch, un rabbitmq etc. Par défaut, l'image docker n'a aucun service d'installer pour améliorer les performances.
- stages
 - C'est les étapes qu'on veut effectuer dans notre CI, par exemple on veut faire un build, fixture, test. On peut créer autant d'étapes qu'on veut.
- variables
 - Les variables de configuration qu'on veut mettre en place dans Gitlab CI.
- build / test
 - Stage : L'étape que le script doit être utilisé.
- Script :
 - C'est les scripts qui vont permettre de mettre en place l'environnement de tests, lancer les tests unitaires. Pour mon environnement de tests, j'ai décidé d'installer le prochain avec les bundles dont j'ai besoin, la base de données et les fixtures de notre dernier article.

LES JOBS (TÂCHES)

- Les jobs sont les éléments de base du pipeline.
- Chaque job est associé à un runner (hébergé par GitLab ou auto-hébergé).
- Le script contient les commandes à exécuter pour chaque job.
- Les jobs peuvent s'exécuter en parallèle ou en séquence via des stages.

EXEMPLE DE JOB

job1:

script:

- echo "Hello World"
- ls -l
- cat README.md

LE SCRIPT

- Le script contient les commandes à exécuter par le job.
- Les scripts peuvent être écrits en shell, bash, etc.
- Les commandes sont exécutées de manière séquentielle.
- Exemples : compilation, tests, déploiement.

EXÉCUTION DE SCRIPTS SHELL DANS UN JOB

```
execute_script_job:
```

```
  script:
```

```
    - ./mon_script.sh
```

BEFORE_SCRIPT ET AFTER_SCRIPT

- Before_script : s'exécute avant le script principal.
- Utilisé pour installer des dépendances ou préparer l'environnement.
- After_script : s'exécute après le job principal.
- Utilisé pour nettoyer les ressources.

EXEMPLE BEFORE_SCRIPT ET AFTER_SCRIPT

before_script:

- apt-get update
- apt-get install -y cowsay

script:

- cowsay "Hello from CI/CD"

after_script:

- echo "Job terminé"

BONNES PRATIQUES - LES VARIABLES

- Variables globales : Définir des variables communes en haut du fichier.
- Variables prédéfinies par job
- Réutilisation : Utiliser des ancres YAML pour éviter la duplication.
- Les ancres YAML permettent de réutiliser des blocs de code ou des configurations plusieurs fois dans un fichier YAML, sans avoir à les dupliquer manuellement.
- [Predefined CI/CD variables reference | GitLab](#)
- [Where variables can be used | GitLab](#)

variables:

DATABASE_URL: "postgres://staging_user:pass@staging-db:5432/dbname"

variables:

DATABASE_URL: "postgres://prod_user:pass@prod-db:5432/dbname"

VARIABLES D'ENVIRONNEMENTS

- Les variables d'environnement sont utilisées pour passer des informations aux jobs, comme des clés API, des configurations, etc.
- Variables prédéfinies dans GitLab
- Variables protégées : Configurées via l'interface GitLab (Settings > CI/CD > Variables). Elles peuvent être masquées et protégées.

image: docker:27.2.0

stages:

- Build
- Push
- Deploy

variables:

APP: \$CI_REGISTRY_IMAGE:\$IMAGE_TAG

DOCKER_HOST: tcp://127.0.0.1:2375

IMAGE_TAG: "\${CI_COMMIT_REF_SLUG}-\${CI_COMMIT_SHORT_SHA}"

IMAGE_LATEST_TAG: "latest"

DATABASE_URL: "postgres://user:pass@localhost:5432/dbname"

script:

```
- echo "Connexion à la base de données :  
$DATABASE_URL"  
- docker build -t $GO_C1_APP  
- docker tag $GO_C1_APP  
$CI_REGISTRY_IMAGE:$IMAGE_LATEST_TAG
```

BONNES PRATIQUES - LES VARIABLES

```
variables:  
  DATABASE_URL: "postgres://staging_user:pass@staging-db:5432/dbname"
```

```
variables:  
  DATABASE_URL: "postgres://prod_user:pass@prod-db:5432/dbname"
```

LE CACHE

- Le cache stocke des fichiers entre les jobs et les pipelines pour accélérer les exécutions.
- key : Définit une clé de cache unique par branche.

- Configuration du cache :

cache:

paths:

- node_modules/

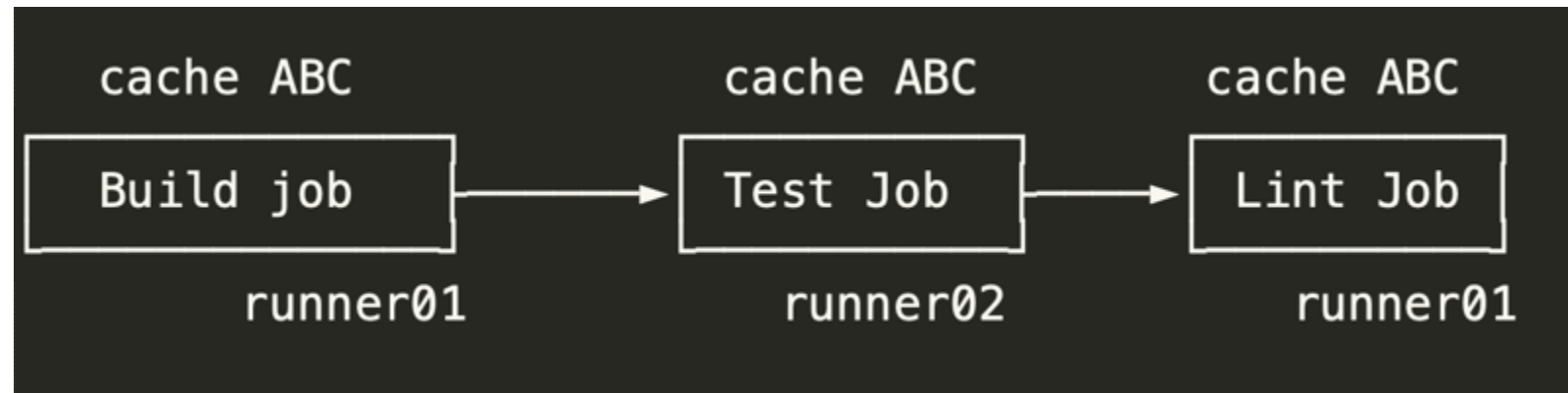
- Utilisation avancée avec clés :

cache:

key: "\$CI_COMMIT_REF_SLUG"

paths:

- node_modules/



CACHE VALIDÉ ET INVALIDÉ

- Un **cache validé** est un cache qui correspond à la clef donnée, et donc il **pourra être réutilisé**.
- Un **cache invalidé** est donc le contraire, si la clef change, le cache sera invalidé, et sera donc **recréé ou remplacé**.

LE CACHE

- Concepts key, prefix, files et cache
- key : La clé du cache est basée sur des fichiers spécifiques (package-lock.json) et un préfixe. Cela permet de s'assurer que le cache est réutilisé uniquement si ces éléments n'ont pas changé.
- prefix : Préfixe ajouté pour mieux organiser les caches dans GitLab.
- files : Fichiers qui déterminent le comportement du cache. Un changement dans ces fichiers forcera le cache à se régénérer.
- cache : Mécanisme qui permet de stocker les dépendances et fichiers fréquemment utilisés entre les jobs, accélérant ainsi le pipeline.

CACHE : PATHS ET KEY

- `paths` : Le chemin du dossier qui sera mis en cache.
- `key` : Détermine l'identifiant unique du cache. En modifiant la clé, le cache est invalidé ou différencié, comme avec `$CI_COMMIT_REF_SLUG` pour générer une clé basée sur la branche.

`cache:`

`key: "$CI_COMMIT_REF_SLUG"`

`paths:`

- `node_modules/`

- Donc ici par exemple, la variable `CI_COMMIT_REF_SLUG` correspond à la branche, le cache sera donc partagé entre les pipelines de la même branches. Donc chaque branche aura son cache distinct si on met ça sur chaque branche.
- Si par exemple j'utilise `$CI_COMMIT_SHA`, ce sera un cache par commit, plus lent.

CACHE : PATHS ET FILES

- `paths` : Le chemin du dossier qui sera mis en cache.
- `key` : Fichiers qui déterminent le comportement du cache. Un changement dans ces fichiers forcera le cache à se régénérer. Si le fichier ne change pas, le cache ne sera pas invalidé.

```
cache:
  key:
    files:
      - packages-lock.json
  paths:
    - node_modules/
```

- Donc ici par exemple, la clef est le fichier `packages-lock.json`, **si ce fichier est modifié le cache sera invalidé.**
- la clef sera donc considéré comme changé et le **cache sera recréé.**
- Le chemin du cache `node_modules/` sera réinstallé et stocké dans ce nouveau cache **tant que packages-lock.json sera inchangé.**

CACHE : PATHS, FILES ET PREFIX

- prefix : Il permet de combiné une clef ou un files pour éventuellement séparer les caches.
- On peut donc créer des caches distincts pour chaque job (par exemple, pour chaque job, on peut avoir un cache différent même si les fichiers d'invalidation sont les mêmes).
- Donc la le cache sera invalidé si le package-lock.json est modifié, et ce cache pourra être réutilisé si on met le prefix dans la réutilisation du cache. Si il est différent, il ne sera pas réutilisé même si la clef est la même.

```
build_job:  
  cache:  
    key:  
      files:  
        - package-lock.json  
      prefix: node-modules  
  paths:  
    - node_modules
```

```
test_job:  
  cache:  
    key:  
      files:  
        - package-lock.json  
      prefix: tests-modules  
  paths:  
    - node_modules
```

CACHE : PATHS, FILES ET PREFIX

- Intérêt de prefix :
- **Isolation des Caches** : Même avec les mêmes fichiers pour générer les clés, les caches seront isolés par job grâce à prefix.
- **Contrôle et Organisation** : mieux organiser et contrôler les caches pour des parties spécifiques du pipeline.
- **N:B** : Ici le cache global définit une configuration de cache par défaut si non défini dans un job.

```
stages:  
- build  
- test  
  
cache:  
  key:  
  files:  
  - package-lock.json  
  prefix: $CI_JOB_NAME  
  paths:  
  - node_modules/  
  policy: pull-push  
  
build_job:  
  stage: build  
  script:
```

```
  - npm install  
  cache:  
  key:  
  files:  
  - package-lock.json  
  prefix: $CI_JOB_NAME  
  
test_job:  
  stage: test  
  script:  
  - npm test  
  cache:  
  key:  
  files:  
  - package-lock.json  
  prefix: $CI_JOB_NAME
```

LE CACHE

```
unit_testing:
  stage: test
  image: node:17-alpine3.14
  before_script:
    - npm install
  script:
    - npm test
  artifacts:
    name: Mocha-Test-Result
    when: on_success
    paths:
      - test-results.xml
    expire_in: 3 days
  cache:
    policy: pull-push
    key:
      files:
        - package-lock.json
      prefix: kk-lab-node-modules
    paths:
      - node_modules
```

■ artifacts :

- Sauvegarde le fichier de résultats des tests test-results.xml lorsque le job réussit (when: on_success).
- Cet artifact expirera au bout de 3 jours.

■ cache :

- Définit une clé de cache pour gérer les dépendances Node.js (dans node_modules).
- key : Le cache sera créé avec une clé basée sur le fichier package-lock.json et aura un préfixe kk-lab-node-modules. Ce préfixe permet de mieux identifier et organiser les caches liés à ce projet.
- policy : pull-push permet d'utiliser le cache existant s'il est disponible, et de le mettre à jour après le job.

LE CACHE

```
code_coverage:
  stage: test
  image: node:17-alpine3.14
  before_script:
    - npm install
  script: |
    npm run coverage
  cache:
    policy: pull
    key:
      files:
        - package-lock.json
      prefix: kk-lab-node-modules
    paths:
      - node_modules
  artifacts:
    name: Lab3-Code-Coverage-Result
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage/cobertura-coverage.xml
  allow_failure: true
```

- cache :
 - Similaire au job précédent, mais ici la politique est pull, ce qui signifie que seul le cache existant est utilisé sans le mettre à jour.
- artifacts :
 - Génère un rapport de couverture de code sous le format cobertura.
 - Ce rapport est utile pour visualiser la couverture de test dans GitLab.
 - allow_failure : Autorise ce job à échouer sans affecter l'état global du pipeline.

LE CACHE

```
workflow:
  name: Solar System NodeJS Pipeline
  rules:
    - if: $CI_COMMIT_BRANCH == 'main' || $CI_COMMIT_BRANCH =~
/^feature/
      when: always
    - if: $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME =~ /^feature/
&& $CI_PIPELINE_SOURCE == 'merge_request_event'
      when: always

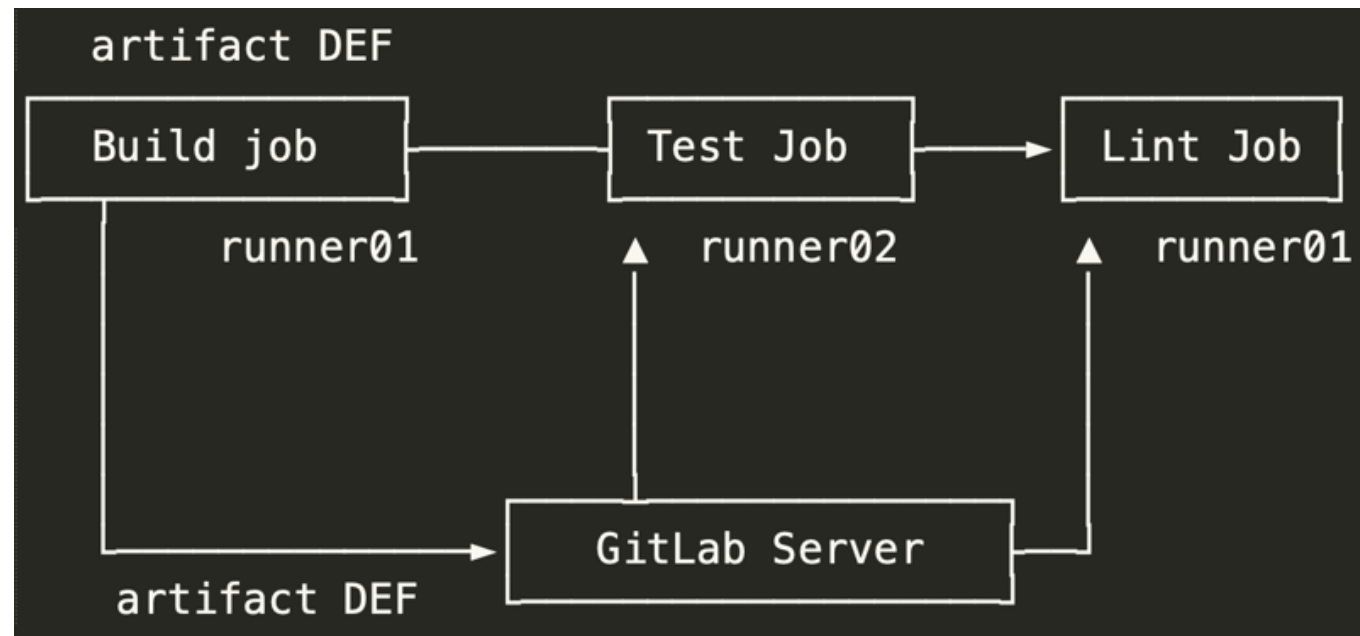
  stages:
    - test

  variables:
    MONGO_URI: mongodb://k8:30017/
    MONGO_DB_NAME: superData
    MONGO_USERNAME: root
    MONGO_PASSWORD: $M_DB_PASSWORD

  unit_testing:
    stage: test
    image: node:17-alpine3.14
    before_script:
      - npm install
      script:
        - npm test
      artifacts:
        name: Mocha-Test-Result
        when: on_success
        paths:
          - test-results.xml
        expire_in: 3 days
      cache:
        policy: pull-push
        key:
          files:
            - package-lock.json
          prefix: kk-lab-node-modules
        paths:
          - node_modules
      code_coverage:
        stage: test
        image: node:17-alpine3.14
      before_script:
        - npm install
      script: |
        npm run coverage
      cache:
        policy: pull
        key:
          files:
            - package-lock.json
          prefix: kk-lab-node-modules
        paths:
          - node_modules
      artifacts:
        name: Lab3-Code-Coverage-Result
        reports:
          coverage_report:
            coverage_format: cobertura
            path: coverage/cobertura-coverage.xml
        allow_failure: true
```

UTILISATION DES ARTEFACTS

- Les artefacts permettent de partager des fichiers entre jobs.
- Définis dans un job et utilisés par les jobs suivants.
- Exemples : fichiers de construction, résultats de tests.



LES TYPES D'ARTIFACTS

- paths
 - Spécifie les fichiers ou répertoires à sauvegarder comme artifacts.
- reports
 - Description : Permet de générer différents types de rapports (par exemple, tests, couverture de code, SAST) qui sont affichés dans l'interface GitLab.
 - Sous-types
 - junit : Pour les rapports de tests unitaires.
 - codequality : Pour les rapports de qualité de code.
 - sast : Pour les rapports d'analyse de sécurité statique.
 - dependency_scanning : Pour les rapports de scan de dépendances.
 - coverage : Pour les rapports de couverture de code.

LES TYPES D'ARTIFACTS

- **expose_as** : Donne un nom spécifique aux artifacts, ici « Build Results », pour une meilleure visibilité.
- **expire_in** : Définit la durée de conservation des artifacts, ici une semaine.
- **when** : Spécifie quand créer les artifacts. Dans cet exemple, seulement en cas de succès.
- **public** : Rend les artifacts disponibles publiquement via un lien.
- **exclude** : Exclut certains sous-dossiers ou fichiers du répertoire, comme temp/ dans test-results/.

EXEMPLE D'UTILISATION D'ARTEFACTS

artifacts:

paths:

- build/ # 1. paths : sauvegarde le répertoire build

expose_as: "Build Results" # 3. expose_as : nomme l'artifact dans l'interface GitLab

expire_in: 1 week # 4. expire_in : garde l'artifact pendant une semaine

when: on_success # 5. when : crée l'artifact seulement si le job réussit

artifacts:

reports:

junit: report.xml # 2. reports : enregistre un rapport JUnit pour les tests

codequality: gl-code-quality-report.json # enregistre un rapport de qualité de code

public: true # 6. public : rend les artifacts accessibles publiquement

paths:

- test-results/

exclude:# 7. exclude : exclut certains fichiers du répertoire

- test-results/temp/

BONNES PRATIQUES

Sécurité

- Informations sensibles : Ne jamais inclure de mots de passe ou de clés API en clair.
- Variables protégées : Utiliser les variables d'environnement protégées dans GitLab pour les secrets.

BONNES PRATIQUES

Optimisation des pipelines

- Parallélisation : Exploiter la parallélisation des jobs pour réduire le temps total.
- Caches et artefacts : Utiliser judicieusement pour accélérer les builds.
- Limitation des ressources : Spécifier les limites de ressources si nécessaire.

CONFIGURATION DES JOBS

```
build_app:
  stage: build
  script:
    - npm install
    - npm run build
  artifacts:
    paths:
      - dist/

# Tests unitaires
unit_tests:
  stage: test
  script:
    - npm test
  dependencies:
    - build_app

# Déploiement en staging
deploy_staging:
  stage: deploy_staging
  script:
    - scp -r dist/ user@staging-server:/var/www/html/
  environment:
    name: staging
    url: http://staging.example.com
  only:
```

```
    - develop

# Étape d'approbation manuelle
manual_approval:
  stage: manual_approval
  when: manual
  script:
    - echo "Approbation nécessaire pour le déploiement en production"
  only:
    - main

# Déploiement en production
deploy_production:
  stage: deploy_production
  script:
    - scp -r dist/ user@production-server:/var/www/html/
  environment:
    name: production
    url: http://www.example.com
  only:
    - main
  when: manual
  dependencies:
    - manual_approval
```

CONFIGURATION DES JOBS

- `when: manual` : Le job nécessite une intervention manuelle pour être déclenché.
- `manual_approval` : Étape pour obtenir une approbation avant de déployer en production.
- `Dependencies` : Assure que le déploiement en production ne se fait qu'après l'approbation.

CONFIGURATION DES JOBS

Gestion des clés SSH pour le déploiement

- Clé privée : Ne jamais la committer dans le dépôt. Utiliser une variable d'environnement protégée.
- Clé publique : Ajoutée au fichier `authorized_keys` du serveur cible.

before_script:

```
- 'which ssh-agent || ( apt-get update -y && apt-get install openssh-  
client -y )'  
- eval $(ssh-agent -s)  
- echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -  
- mkdir -p ~/.ssh  
- chmod 700 ~/.ssh  
- ssh-keyscan -H $SERVER_IP >> ~/.ssh/known_hosts
```

CONFIGURATION DES JOBS

- Nous pouvons considérer des jobs comme suite pour une pipeline

stages:

- build
- test
- deploy_staging
- manual_approval
- deploy_production

GITLAB PAGES POUR DES SITES STATIQUES

- Service intégré : Permet d'héberger des sites statiques directement depuis votre dépôt.
- Cas d'utilisation : Documentation, blogs, portfolios, sites d'entreprise.

PIPELINE POUR SITE STATIQUE

```
image: node:15.12-alpine3.13
```

```
stages:
```

- test
- deploy

```
test:
```

```
  stage: test
```

```
  script:
```

- cd website
- yarn install
- yarn build

```
  rules:
```

- if: \$CI_COMMIT_REF_NAME !=
\$CI_DEFAULT_BRANCH

```
pages:
```

```
  stage: deploy
```

```
  script:
```

- cd website
- yarn install
- yarn build
- mv ./build ../public

```
  artifacts:
```

```
    paths:
```

- public

```
  rules:
```

- if: \$CI_COMMIT_REF_NAME ==
\$CI_DEFAULT_BRANCH

ANCRES (&, *, ET <<)

- Les ancres YAML permettent de réutiliser des blocs de code ou des configurations plusieurs fois dans un **fichier YAML**, sans avoir à les dupliquer manuellement. On les utilise pour éviter les répétitions.
- Pour réutiliser une ancre ailleurs dans le fichier YAML, on utilise l'opérateur de fusion **<<: *** suivi du nom de l'ancre (ici, **default_variables**).

- Exemple avec ancres YAML :

```
.default_variables: &default_variables
```

```
  PYTHON_VERSION: "3.8"
```

```
  PIP_CACHE_DIR: ".cache/pip"
```

```
  VENV_DIR: "venv"
```

```
variables:
```

```
  <<: *default_variables
```

```
.cache_settings: &cache_settings
```

```
  cache:
```

```
    key:
```

```
      files:
```

```
        - package-lock.json
```

```
      paths:
```

```
        - node_modules/
```

```
      policy: pull-push
```

```
build_job:
```

```
  stage: build
```

```
  <<: *cache_settings
```

```
  script:
```

```
    - npm install
```

```
test_job:
```

```
  stage: test
```

```
  <<: *cache_settings
```

```
  script:
```

```
    - npm test
```

UTILISATION DES ANCRES

1. Une ancre est défini avec le symbole "&" avec son nom au début du fichier.
2. On appelle cette ancre avec "*" et son nom dans le fichier.
3. La clé "<<" est utiliser que toutes les propriétés de l'ancre doivent être utilisées.
4. Une ancre peut être appelé en variable et script également.

EXEMPLE D'ANCRES

```
.default_variables: &default_variables
```

```
  PYTHON_VERSION: "3.8"
```

```
  PIP_CACHE_DIR: ".cache/pip"
```

```
  VENV_DIR: "venv"
```

```
.template: &template
```

```
  image: "python:3.8"
```

```
  services:
```

- python

```
variables:
```

```
  <<: *default_variables
```

```
stages:
```

- install
- test

```
install_dependencies:
```

```
  stage: install
```

```
  <<: *template
```

```
  script:
```

- python -m venv \$VENV_DIR
- source \$VENV_DIR/bin/activate
- pip install -r requirements.txt

```
artifacts:
```

```
  paths:
```

- \$VENV_DIR/

```
cache:
```

```
  paths:
```

- \$PIP_CACHE_DIR

```
unit_tests:
```

```
  stage: test
```

```
  script:
```

- source \$VENV_DIR/bin/activate
- pytest tests/

```
dependencies:
```

- install_dependencies

```
variables:
```

```
  <<: *default_variables
```

EXEMPLE D'ANCRES

```
.default_variables: &default_variables
```

```
  PYTHON_VERSION: "3.8"
```

```
  PIP_CACHE_DIR: ".cache/pip"
```

```
  VENV_DIR: "venv"
```

```
.template_python:
```

```
  services: &python_config
```

```
    - python
```

```
.template_mysql:
```

```
  services: &mysql_config
```

```
    - mysql
```

```
variables:
```

```
  <<: *default_variables
```

```
stages:
```

```
  - install
```

```
  - mysql
```

```
install_python:
```

```
  stage: install
```

```
<<: *default_variables
```

```
services: *python_config
```

```
script:
```

```
  - python -m venv $VENV_DIR
```

```
  - source $VENV_DIR/bin/activate
```

```
  - pip install -r requirements.txt
```

```
artifacts:
```

```
  paths:
```

```
    - $VENV_DIR/
```

```
cache:
```

```
  paths:
```

```
    - $PIP_CACHE_DIR
```

```
install_mysql:
```

```
  stage: mysql
```

```
  services: *mysql_config
```

```
tags:
```

```
  - mysql
```

LE EXTENDS

- Les extends permettent à un job d'hériter d'un autre jobs.
- Si j'ai plusieurs tests unitaire basé sur un job de test commun par exemple.

```
.tests:  
  stage: test  
  script:  
    - echo Tests basiques  
.tests_unit:  
  extends: .tests  
  script:  
    - pytest tests/  
    - echo Test unitaires
```

EXTENDS

- **extends** est une fonctionnalité spécifique à GitLab CI/CD qui permet à un job d'hériter d'un ou plusieurs autres jobs.
- Contrairement aux ancres, extends permet de faire hériter un job d'un autre job complet, y compris des étapes, des variables, des artefacts, etc.
- Cela rend extends particulièrement puissant pour la **réutilisation de jobs complets** ou la création de variations de jobs existants.

EXTENDS UTILISATION

- Définir un Job de Base : Crée un job standard avec les étapes et configurations partagées.
- Étendre un Job de Base : Utilise extends pour permettre à d'autres jobs de tirer parti de cette configuration de base.

EXTENDS

- Définir un Job de Base : Crée un job standard avec les étapes et configurations partagées.
- Étendre un Job de Base : Utilise extends pour permettre à d'autres jobs de tirer parti de cette configuration de base.

```
.default_job_template:
  stage: build
  script:
    - echo "Configuration commune"
  cache:
    paths:
      - node_modules/
  artifacts:
    paths:
      - build/

build_job:
  extends: .default_job_template
  script:
    - npm install

test_job:
  extends: .default_job_template
  stage: test
  script:
    - npm test
```

EXTENDS VS ANCRES

- **Usage** : Les **ancres sont spécifiques au langage YAML** et permettent de réutiliser des **blocs de configuration**, tandis **que extends est spécifique à GitLab CI/CD** et permet à un job de **réutiliser la configuration d'un ou de plusieurs autres jobs**.
- **Réutilisation Partielle vs Complète** :
 - Les ancres permettent de réutiliser des parties de configuration (comme le cache ou les artefacts) et de les combiner comme souhaité.
 - extends copie tout le contenu d'un job de base dans un autre, avec la possibilité d'ajouter ou de remplacer des éléments spécifiques.
- **Important**
 - Dans GitLab CI/CD, avec un extends, le **script défini dans le job enfant remplace celui du job parent**. Si on définit un script dans le template et ensuite un autre script dans le job enfant, **le script du template sera ignoré. Vous devez mettre le script lui-même en ancre pour l'utiliser dans un job enfant**.

EXTENDS VS ANCRES

```
.default_cache: &default_cache
  key:
    files:
      - package-lock.json
    paths:
      - node_modules/
.artifacts_config: &artifacts_config
  paths:
    - node_modules/
#####
.test_job_template:
  stage: test
  script: &test_script_init
    - echo "Exécution des tests depuis le template"
  cache:
    <<: *default_cache
  artifacts:
    reports:
      junit: test-results/test-results.xml
      codequality: gl-code-quality-report.json
  paths:
    - test-results/
    - gl-code-quality-report.json
```

```
build_job:
  stage: build
  cache:
    <<: *default_cache
  script:
    - echo "Compilation de
l'application..."
  artifacts:
    <<: *artifacts_config
```

```
test_job_heritage:
  extends: .test_job_template
  script:
    - *test_script_init
    - echo "test supp"
```

LES TEMPLATES GITLAB


- Vous pouvez appeler les templates au seins d'un projet que vous avez, sous forme de fichier YAML. Ou vous pouvez les appeler depuis un autre projet. Ceci grâce à la balise "include"

```
include:
  - local: '/templates/unit_tests.yml'

stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Construction du projet..."

# L'utilisation du template pour les tests unitaires
unit_tests:
  extends: .unit_tests
```



```
.unit_tests:
  stage: test
  script:
    - echo "Exécution des tests unitaires..."
```

LES TEMPLATES GITLAB

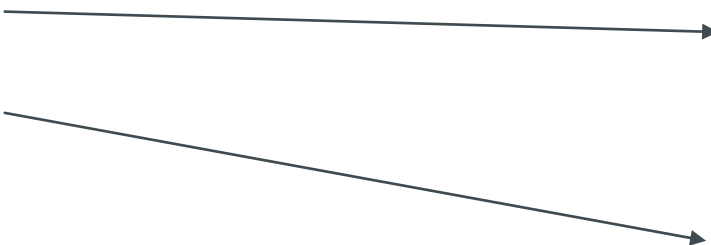
- Vous pouvez appeler les templates au seins d'un projet que vous avez, sous forme de fichier YAML. Ou vous pouvez les appeler depuis un autre projet. Ceci grâce à la balise "include"

```
include:  
  - project: 'vlaine1/templates-cicd'  
    file: 'install_npm.yml'  
    ref: 'main'  
  - project: 'vlaine1/templates-cicd'  
    file: 'unit_test.yml'  
    ref: 'main'
```

```
stages:  
  - build  
  - test  
  - deploy
```

```
build_job:  
  stage: build  
  script:  
    - echo "Construction du projet..."
```

```
unit_tests:  
  extends: .unit_tests  
install_npm:  
  image: node:18-slim  
  extends: .install_npm
```



```
.install_npm:  
  script:  
    - npm install
```

```
.unit_tests:  
  stage: test  
  script:  
    - echo "Exécution des tests  
unitaires..."
```

DÉPLOIEMENT - LES ENVIRONNEMENTS

- Un environnement dans le contexte de GitLab CI/CD et plus largement dans le développement logiciel, désigne un contexte ou un cadre dans lequel une application ou un logiciel est exécuté.
- Cet environnement comprend les serveurs sur lesquels l'application est déployé et s'exécute, les logiciels tiers qu'elle utilise, les variables d'environnement, la configuration réseau et d'autres composants logiciels et matériels.
- Le concept d'environnement est central en et dans les pratiques DevOps, où différentes phases du développement et du déploiement nécessitent des contextes distincts.
- Les **environnements statiques** ont des noms statiques comme staging, testing, development ou encore production.
- Les **environnements dynamiques** ont des noms dynamiques utilisant des variables CI/CD

DÉPLOIEMENT - LES ENVIRONNEMENTS

Importance des environnements multiples

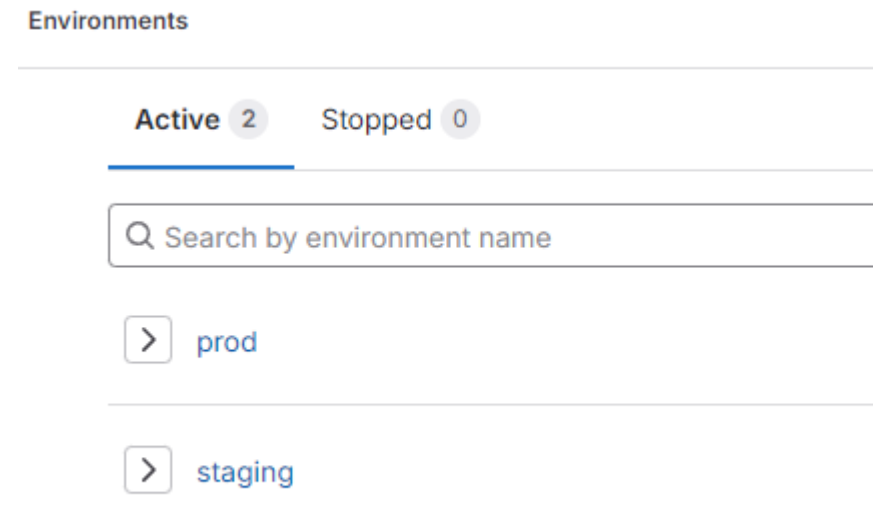
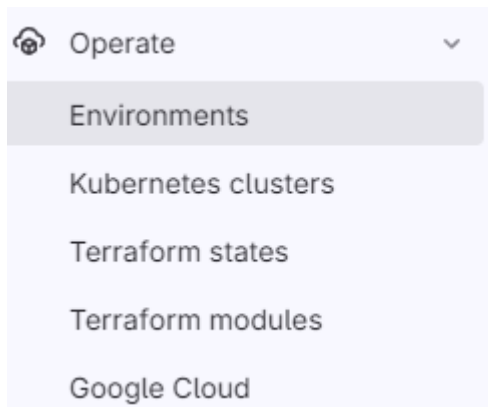
- Développement : Environnement de développement, qui implémentera des features avant la pré-prod par exemple.
- Staging : Environnement de pré-production pour tester les nouvelles fonctionnalités dans des conditions proches de la production.
- Production : Environnement utilisé par les utilisateurs finaux.
- Tests : Qui "imite" la production pour être au plus proche de celui-ci.

Avantages :

- Détection précoce des bugs.
- Isolation des environnements
- Rôle dédié à un environnement
- Tests d'intégration avec d'autres services.
- Validation par les parties prenantes avant mise en production.

CONFIGURATION DES ENVIRONNEMENTS DANS GITLAB

- Accès via GitLab : Aller dans Opérations > Environnements pour voir la liste des environnements.
- Informations disponibles :
- Historique des déploiements.
- URLs des environnements.
- Possibilité de rollback.



CONFIGURATION DES ENVIRONNEMENTS DANS GITLAB

- Définition par environnement : Vous pouvez définir des variables spécifiques à un environnement dans GitLab.

```
variables:  
  DATABASE_URL: "postgres://staging_user:pass@staging-db:5432/dbname"
```

```
variables:  
  DATABASE_URL: "postgres://prod_user:pass@prod-db:5432/dbname"
```

CONFIGURATION DES ENVIRONNEMENTS STATIQUES DANS GITLAB

```
stages:
  - deploy

deploy_staging:
  stage: deploy
  environment:
    name: staging #Statique
    url: http://staging.example.com
  script:
    - echo "Déploiement sur staging"
    - export
      DATABASE_URL="postgres://staging_user:pass@staging-
      db:5432/dbname"
    - ./deploy.sh staging
  only:
```

```
  - develop

deploy_production:
  stage: deploy
  environment:
    name: production
    url: http://prod.example.com
  script:
    - echo "Déploiement sur production"
    - export
      DATABASE_URL="postgres://prod_user:pass@prod-
      db:5432/dbname"
    - ./deploy.sh production
  only:
    - main
```

VISUALISATION

▼ production

Open

Stop



Success

Latest Deployed

#12

3f1ad5ec



Deployed 1 minute ago

Merge branch 'exo3_env' into 'main'

Triggerer	Job	Branch
@vlaine	deploy... roduction	main

Running

#13

6d5e682f



Created just now

Update .gitlab-ci.yml file

Triggerer	Job	Branch
@vlaine	deploy... roduction	main

▼ staging

Open

Stop



Success

Latest Deployed

#11

ef96d869



Deployed 1 minute ago

Update .gitlab-ci.yml file

Triggerer	Job	Branch
@vlaine	deploy_staging	exo3_env

CONFIGURATION DES ENVIRONNEMENTS DYNAMIQUES DANS GITLAB

- Les environnements dynamiques sont générés à la volée en fonction de certaines conditions ou actions, comme la création d'une nouvelle branche ou d'une merge request.
- Utilisé pour des tests de features, ou un environnement associé à une branche dans affecter l'environnement principal de test.
- Environnements statiques : constants et prévisibles, idéaux pour les déploiements réguliers,
- Environnements dynamiques : offrent flexibilité et spécificité, adaptés aux besoins de développement et de test en constante évolution.

CONFIGURATION DES ENVIRONNEMENTS DYNAMIQUES DANS GITLAB

```
image: debian:stable-slim
stages:
  - deploy
deploy_staging:
  stage: deploy
  environment:
    name: staging
    url: http://staging.example.com
  script:
    - echo "Déploiement sur staging"
    - export DATABASE_URL="postgres://staging_user:pass@staging-db:5432/dbname"
    - apt-get update && apt-get install -y dos2unix
    - dos2unix deploy.sh
    - chmod +x ./deploy.sh
    - /bin/bash deploy.sh staging
  only:
    - exo3_env
deploy_production:
  stage: deploy
  environment:
    name: production
    url: http://production.example.com
  script:
    - echo "Déploiement sur production"
    - export DATABASE_URL="postgres://prod_user:pass@prod-db:5432/dbname"
```

```
- apt-get update && apt-get install -y dos2unix
- dos2unix deploy.sh
- chmod +x ./deploy.sh
- ./deploy.sh production
```

```
only:
  - main
```

```
deploy_dynamic:
  stage: deploy
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_COMMIT_REF_NAME.example.com
  script:
    - echo "Déploiement sur dynamic"
    - export DATABASE_URL="postgres://prod_user:pass@prod-db:5432/dbname"
    - apt-get update && apt-get install -y dos2unix
    - dos2unix deploy.sh
    - chmod +x ./deploy.sh
    - ./deploy.sh production
  only:
    - branches
  except:
    - main
    - exo3_env
```

CONFIGURATION DES ENVIRONNEMENTS DYNAMIQUES DANS GITLAB

- Dans cet exemple, un nouvel environnement de revue est créé à chaque fois qu'une nouvelle branche est poussée, à l'exception des branches main et exo3_env.
- Cela signifie que chaque nouvelle branche aura son propre environnement unique, basé sur le nom de la branche.
- Ces environnements sont dynamiques, car ils sont créés et détruits dynamiquement en fonction du flux de travail de développement.

VISUALISATION D'ENV DYNAMIQUES



production

Open

Stop



Success

Latest Deployed

#16

d466e1da



Deployed just now

Merge branch 'exo3_env' into 'main'

Triggerer	Job	Branch
@vlaine	deploy...roduction	main



review/exo3_dyn

Open

Stop



Success

Latest Deployed

#19

8d2b6113



Deployed just now

Merge branch 'exo3_env' into 'exo3_dyn'

Triggerer	Job	Branch
@vlaine	deploy_dynamic	exo3_dyn



staging

Open

Stop



Success

Latest Deployed

#18

9e98d3fd



Deployed just now

Update .gitlab-ci.yml file

Triggerer	Job	Branch
@vlaine	deploy_staging	exo3_env

CONFIGURATION DES ENVIRONNEMENTS DYNAMIQUES DANS GITLAB

L'idée ici est de donner un exemple, mais vous devrez évidemment gérer les endpoints où vos environnements seront déployés.

Ce pourrait être par exemple un serveur ou un cluster kubernetes suivant l'environnement. Par exemple on pourrait faire un test avec un déploiement qui utilise SSH, et un déploiement en production qui envoie sur un cluster.

- Mettre en place l'infrastructure de déploiement :
 - Serveurs pour staging et production.
 - Cluster Kubernetes ou Docker pour les environnements dynamiques.
- Configurer GitLab pour le déploiement :
 - Utiliser SSH, Docker ou Kubernetes pour envoyer et déployer le code sur les serveurs.
- Gestion des URL :
 - Les environnements que tu as définis auront des URLs spécifiques (<http://staging.example.com>, [https://\\$CI_COMMIT_REF_NAME.example.com](https://$CI_COMMIT_REF_NAME.example.com)). Assure-toi que ces URLs correspondent à ton infrastructure réelle (DNS, reverse proxy, etc.).

EXEMPLE AVEC SSH POUR LE STAGGING

deploy_staging:

stage: deploy

environment:

name: staging

url: http://staging.example.com

script:

- echo "Déploiement sur staging"
- export DATABASE_URL="postgres://staging_user:pass@staging-db:5432/dbname"
- apt-get update && apt-get install -y dos2unix
- dos2unix deploy.sh
- chmod +x ./deploy.sh
- scp -o StrictHostKeyChecking=no ./deploy.sh user@staging-server:/path/to/deploy/
- ssh user@staging-server 'bash /path/to/deploy/deploy.sh staging'

only:

- exo3_env

EXEMPLE AVEC UN CLUSTER KUBE POUR LES TESTS DYNAMIQUES (TEMPORAIRES)

```
deploy_dynamic:
  stage: deploy
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_COMMIT_REF_NAME.example.com
  script:
    - echo "Déploiement sur l'environnement dynamique pour $CI_COMMIT_REF_NAME"
    - kubectl apply -f k8s/deployment-$CI_COMMIT_REF_NAME.yml
  only:
    - branches
  except:
    - main
    - exo3_env
```

EXEMPLE AVEC UN CLUSTER KUBE POUR LA PRODUCTION

```
deploy_production:
  stage: deploy
  environment:
    name: production
    url: http://production.example.com
  script:
    - echo "Déploiement sur production"
    - export DATABASE_URL="postgres://prod_user:pass@prod-db:5432/dbname"
    - apt-get update && apt-get install -y kubectl
    - kubectl config set-cluster prod-cluster --server=https://k8s-api.prod.example.com --insecure-skip-tls-verify
    - kubectl config set-credentials gitlab --token=$KUBERNETES_TOKEN
    - kubectl config set-context prod-context --cluster=prod-cluster --user=gitlab
    - kubectl config use-context prod-context
    - kubectl apply -f k8s/production-deployment.yml
  only:
    - main
```

LES VARIABLES GITLAB OU PERSO PEUVENT ÊTRE UTILISÉES DANS LE PIPELINE COMME DANS LE CODE DU REPO

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "Erreur : Vous devez spécifier un environnement (staging ou production)."
```

```
    exit 1
fi

if [ "$1" == "staging" ]; then
    echo "Déploiement en environnement de staging..."
    echo $DATABASE_URL
elif [ "$1" == "production" ]; then
    echo "Déploiement en environnement de production..."
elif [ "$1" == "$CI_COMMIT_REF_NAME" ]; then
    echo "Déploiement en environnement de $CI_COMMIT_REF_NAME..."
else
    echo "Erreur : Environnement inconnu. Veuillez utiliser 'staging' ou 'production'."
    echo $DATABASE_URL
    exit 1
fi
```

VARIABLES SPÉCIFIQUES AUX ENVIRONNEMENTS GITLAB

Quand vous créez un environnement dans GitLab, il est possible d'accéder à des variables spécifiques qui sont automatiquement définies et utiles pour adapter vos scripts en fonction de l'environnement dans lequel le pipeline s'exécute. Ces variables permettent une personnalisation facile et une automatisation des tâches selon l'environnement (staging, production, etc.).

- **CI_ENVIRONMENT_NAME**
 - Contient le nom de l'environnement actuel (ex. "Production" ou "Staging").
 - Utilisation : Pratique pour des scripts ou configurations qui dépendent de l'environnement spécifique.
- **CI_ENVIRONMENT_SLUG**
 - Version simplifiée de CI_ENVIRONMENT_NAME, utilisable dans des URL ou noms de domaines. Les caractères spéciaux sont automatiquement remplacés.
 - Utilisation : Utile pour créer des sous-domaines ou chemins basés sur l'environnement, par exemple pour des environnements de test.
- **CI_ENVIRONMENT_URL**
 - Contient l'URL de l'environnement, définie dans le fichier .gitlab-ci.yml. Permet d'accéder facilement à l'environnement depuis GitLab.
 - Utilisation : Donne un accès direct à l'environnement depuis les pipelines ou les notifications.

Vous pouvez surcharger la variable `$CI_ENVIRONMENT_NAME` si nécessaire, mais `$CI_ENVIRONMENT_SLUG` restera inchangée pour éviter des problèmes dans les URLs ou chemins.

LES CONDITIONS

- Les conditions dans les pipelines servent à faire en sorte d'exécuter ou non une partie de celle-ci, pour par exemple ne pas rééxecuter toute la pipeline à chaque commit/push, ou demander une approbation manuelle.
- Avec les conditions, on optimise notre temps, notre pipeline et donc nos ressources en général.
- Variables : [Predefined CI/CD variables reference | GitLab](#)

LES CONDITIONS

- Pour contrôler l'exécution des jobs dans un pipeline, vous utilisez la section **rules**, qui permet de définir des conditions spécifiques.

job:

script:

- `echo "Cette étape s'exécute si la condition est satisfaite"`

rules:

- `if: '$CI_COMMIT_BRANCH == "main"'`

- Ce job s'exécutera uniquement si la branche du commit est "main".

LES CONDITIONS

Quelques règles courantes :

- Tag associé : rules: - if: '\$CI_COMMIT_TAG'
- Déclenchement manuel : rules: - if: '\$CI_PIPELINE_SOURCE == "manual"'

Utilisation de when :

- **on_success** : Job exécuté après un succès.
- **on_failure** : Job exécuté après un échec.
- **manual** : Job déclenché manuellement.
- **delayed** : Job exécuté avec un délai, par exemple start_in: 30 minutes.
- Ces conditions permettent de personnaliser le déclenchement des jobs en fonction des besoins du projet, optimisant ainsi l'exécution des pipelines.

ON_SUCCESS

rules:

```
- if: '$CI_COMMIT_BRANCH == "main"'  
  when: on_success
```

- Le job est exécuté si la branche du commit est “main” et si tous les jobs précédents ont réussi.

ON_FAILURE

rules:

```
- if: '$CI_COMMIT_BRANCH == "main"'  
  when: on_failure
```

- Le job s'exécute si la branche est "main" et si au moins un job précédent a échoué.

ALWAYS

rules:

```
- if: '$CI_COMMIT_BRANCH == "main"'  
  when: always
```

- Le job est toujours exécuté, quelle que soit la réussite ou l'échec des jobs précédents.

MANUAL

rules:

```
- if: '$CI_COMMIT_BRANCH == "main"'  
  when: manual
```

- Le job est exécuté manuellement par l'utilisateur, indépendamment des autres conditions.

DELAYED

rules:

- `if: '$CI_COMMIT_BRANCH == "main"'`
 `when: delayed`
 `start_in: 30 minutes`

- Le job est exécuté automatiquement, mais avec un délai de 30 minutes.

SCHEDULED

rules:

```
- if: '$CI_COMMIT_BRANCH == "main"'  
  when: scheduled  
  cron: "0 12 * * *"
```

- Le job est planifié pour s'exécuter tous les jours à midi.

CONDITIONS SUR LES FICHIERS

- Il existe des conditions "change" et "exist", seules ou combinées.
- **change** : spécifie des fichiers ou des répertoires qui, lorsqu'ils sont modifiés dans un commit, autorise l'exécution d'un job.
 - Ex : utile pour des tâches telles que la compilation du code uniquement lorsqu'un fichier source est modifié.
- **exist** : vérifie l'existence d'un fichier ou d'un répertoire dans le référentiel.
 - Si le fichier ou le répertoire existe, le job est exécuté.

```
job:
  script:
    - echo "fichier lol.txt"
  rules:
    - changes:
      - lol.txt
```

```
rules:
  - exists:
    - data.csv

# exécuté si le
# fichier data.csv
# existe
```

```
job:
  script:
    - echo "fichier lol.txt"
  rules:
    - changes:
      - lol.txt
    - exists:
      - data.csv
```


INTRODUCTION À WORKFLOW

- La directive workflow détermine si un pipeline doit être exécuté, en fonction de critères globaux comme la branche ou l'événement.
- Utile dans des projets complexes avec beaucoup de branches et conditions.
- Peut être basé sur des variables dans des conditions.
- Cas courants :
 - Exécutions de pipeline uniquement pour certains events ou branche.
 - Par exemple, exécution du pipeline uniquement lors d'un push sur la branche main mais pas sur les branches de corrections de bug.
- Il est possible de combiner les rules et les workflow.
- [GitLab CI/CD `workflow` keyword | GitLab](#)

EXEMPLE DE WORKFLOW

workflow:

rules:

- if: '\$CI_PIPELINE_SOURCE == "push"'
- if: '\$CI_COMMIT_BRANCH == "main"'
- if: '\$CI_COMMIT_BRANCH =~ /^feature-/'

- Ce workflow exécute un pipeline lors d'un **push** sur la branche **main** ou une branche commençant par **feature-**.

ÉVITER LES PIPELINES REDONDANTS

workflow:

rules:

- if: '\$CI_COMMIT_TAG'
when: never
- if: '\$CI_PIPELINE_SOURCE == "push"'

- Ce pipeline ne s'exécute pas lors d'un tag, évitant des exécutions redondantes pour le même commit.

GESTION DES PIPELINES POUR MERGE REQUESTS

workflow:

rules:

- if: '\$CI_PIPELINE_SOURCE == "merge_request_event"'
- if: '\$CI_COMMIT_BRANCH && \$CI_OPEN_MERGE_REQUESTS'

when: never

- Ce pipeline est déclenché uniquement pour les événements de merge requests.

COMBINAISON DE RULES ET WORKFLOW

workflow:

rules:

- if: '\$CI_PIPELINE_SOURCE == "push" && \$CI_COMMIT_BRANCH == "main"'

deploy:

script: deploy.sh

rules:

- if: '\$CI_COMMIT_BRANCH == "main"'

when: manual

- Le job deploy est configuré pour s'exécuter manuellement sur la branche principale.

COMBINAISON DE RULES ET WORKFLOW

Avantages :

- **Optimiser les Ressources** : Réduire les exécutions inutiles ou redondantes de pipelines.
- **Améliorer la Clarté** : Clarifier la logique de déclenchement des pipelines.
- **Personnaliser les Pipelines** : Adapter les pipelines aux besoins spécifiques de différentes branches ou types d'événements dans votre projet.

EXEMPLE CONCRET

image: debian:stable-slim

.default_script: &default_script

- apt-get update && apt-get install -y dos2unix
- dos2unix deploy.sh
- chmod +x ./deploy.sh

variables:

STAGING_DATABASE_URL: "postgres://staging_user:pass@staging-db:5432/dbname"

PRODUCTION_DATABASE_URL: "postgres://prod_user:pass@prod-db:5432/dbname"

workflow:

rules:

- if: '\$CI_PIPELINE_SOURCE == "push" && \$CI_COMMIT_BRANCH == "main"'
- if: '\$CI_PIPELINE_SOURCE == "push" && \$CI_COMMIT_BRANCH == "exo3_env"'
- if: '\$CI_PIPELINE_SOURCE == "merge_request_event"'
- if: '\$CI_PIPELINE_SOURCE == "schedule"'
- if: '\$CI_PIPELINE_SOURCE == "push" && \$CI_COMMIT_BRANCH =~ /^feature/'

stages:

- deploy

deploy_staging:

stage: deploy

environment:

name: staging

url: http://staging.example.com

script:

- echo "Déploiement sur staging"
- export DATABASE_URL="\$STAGING_DATABASE_URL"
- *default_script
- /bin/bash deploy.sh staging

rules:

- if: '\$CI_COMMIT_BRANCH == "exo3_env"'
- when: on_success

deploy_production:

stage: deploy

environment:

name: production

url: http://production.example.com

script:

- echo "Déploiement sur production"
- export DATABASE_URL="\$PRODUCTION_DATABASE_URL"
- *default_script
- ./deploy.sh production

rules:

- if: '\$CI_COMMIT_BRANCH == "main" && CI_PIPELINE_SOURCE == "push"'
- when: manual

deploy_dynamic:

stage: deploy

environment:

name: review/\$CI_COMMIT_REF_NAME

url: https://\$CI_COMMIT_REF_NAME.example.com

script:

- echo "Déploiement sur dynamic"
- export DATABASE_URL="\$PRODUCTION_DATABASE_URL"
- *default_script
- ./deploy.sh \$CI_COMMIT_REF_NAME

rules:

- if: '\$CI_COMMIT_BRANCH =~ /^feature/' # Ne déclenche que pour les branches "feature-"
- when: on_success

- if: '\$CI_COMMIT_BRANCH != "main" && \$CI_COMMIT_BRANCH != "exo3_env"' # Exclure "main" et "exo3_env"

when: never # Bloquer le job si on est sur "main" ou "exo3_env"

Job qui sera déclenché par un pipeline planifié

deploy_scheduled:

stage: deploy

environment:

name: production

url: http://production.example.com

script:

- echo "Déploiement planifié sur production"
- export DATABASE_URL="\$PRODUCTION_DATABASE_URL"
- *default_script
- ./deploy.sh production

rules:

- if: '\$CI_PIPELINE_SOURCE == "schedule"' # Exécute uniquement si le pipeline est planifié

EXEMPLE CONCRET

image: debian:stable-slim

.default_script: &default_script

- apt-get update && apt-get install -y dos2unix
- dos2unix deploy.sh
- chmod +x ./deploy.sh

variables:

STAGING_DATABASE_URL: "postgres://staging_user:pass@staging-db:5432/dbname"

PRODUCTION_DATABASE_URL: "postgres://prod_user:pass@prod-db:5432/dbname"

workflow:

rules:

- if: '\$CI_PIPELINE_SOURCE == "push" && \$CI_COMMIT_BRANCH == "main"'
- if: '\$CI_PIPELINE_SOURCE == "push" && \$CI_COMMIT_BRANCH == "exo3_env"'
- if: '\$CI_PIPELINE_SOURCE == "merge_request_event"'
- if: '\$CI_PIPELINE_SOURCE == "schedule"'
- if: '\$CI_PIPELINE_SOURCE == "push" && \$CI_COMMIT_BRANCH =~ /^feature/'

stages:

- deploy

EXEMPLE CONCRET

Déploiement pour l'environnement staging

deploy_staging:

stage: deploy

environment:

name: staging

url: http://staging.example.com

script:

- echo "Déploiement sur staging"
- export DATABASE_URL="\$STAGING_DATABASE_URL"
- *default_script
- /bin/bash deploy.sh staging

rules:

- if: '\$CI_COMMIT_BRANCH == "exo3_env"'
- when: on_success

EXEMPLE CONCRET

Déploiement pour l'environnement production

deploy_production:

stage: deploy

environment:

name: production

url: http://production.example.com

script:

- echo "Déploiement sur production"
- export DATABASE_URL="\$PRODUCTION_DATABASE_URL"
- *default_script
- ./deploy.sh production

rules:

- if: '\$CI_COMMIT_BRANCH == "main" && \$CI_PIPELINE_SOURCE == "push"'
- when: manual

EXEMPLE CONCRET

Déploiement dynamique pour les branches de review

deploy_dynamic:

stage: deploy

environment:

name: review/\$CI_COMMIT_REF_NAME

url: https://\$CI_COMMIT_REF_NAME.example.com

script:

- echo "Déploiement sur dynamic"
- export DATABASE_URL="\$PRODUCTION_DATABASE_URL"
- *default_script
- ./deploy.sh \$CI_COMMIT_REF_NAME

rules:

- if: '\$CI_COMMIT_BRANCH =~ /^feature/' # Ne déclenche que pour les branches "feature-"
when: on_success
- if: '\$CI_COMMIT_BRANCH != "main" && \$CI_COMMIT_BRANCH != "exo3_env"' # Exclure "main" et "exo3_env"
when: never # Bloquer le job si on est sur "main" ou "exo3_env"

EXEMPLE CONCRET

Job qui sera déclenché par un pipeline planifié

deploy_scheduled:

stage: deploy

environment:

name: production

url: http://production.example.com

script:

- echo "Déploiement planifié sur production"
- export DATABASE_URL="\$PRODUCTION_DATABASE_URL"
- *default_script
- ./deploy.sh production

rules:

- if: '\$CI_PIPELINE_SOURCE == "schedule"' # Exécute uniquement si le pipeline est planifié

WARNING SUR LE SCHEDULE

- Sur les versions les plus récentes de gitlab, le schedule ne passe plus par la pipeline mais est configuré directement dans gitlab.

Étapes pour planifier des pipelines dans GitLab :

- Accéder à la section "CI / CD" de ton projet :
 - Va dans ton projet GitLab.
 - Clique sur Settings > CI/CD.
- Ajouter un pipeline planifié (Scheduled Pipeline) :
 - Dans la section "Pipelines", clique sur l'onglet "Pipeline Schedules".
 - Clique sur "New schedule".
 - Spécifie la fréquence (par exemple, tous les jours à midi avec 0 12 * * *) et sélectionne la branche sur laquelle le pipeline sera exécuté (comme main).
- Le job sera exécuté automatiquement par le pipeline planifié via l'interface GitLab

LES PIPELINES PARENTS/ENFANTS

- Concept général : Un pipeline parent peut déclencher un ou plusieurs pipelines enfants.
- Améliore la modularité, la gestion des ressources, et la séparation des tâches CI/CD.
- La directive "trigger" est utilisée.

```
stages:
  - builds

build1:
  stage: builds
  trigger:
    include: build.yml
    strategy: depend

build2:
  stage: builds
  trigger:
    include: vlaine1/projet1
    branch: main
    strategy: depend
```

LES PIPELINES PARENTS/ENFANTS

- Au niveau organisation, il est intéressant d'ajouter par exemple des variables d'une pipeline parent à une pipeline enfant.
- Cette surcharge peut être intéressante pour conditionner l'exécution du pipeline enfant.

LES PIPELINES PARENTS/ENFANTS

- Par défaut le pipeline parent se termine tout de suite en status success avant même que les enfants ne se soient exécutés.
- Pour l'obliger à attendre la fin de l'exécution des enfants et de lier son status avec ceux-ci, il faut indiquer une stratégie avec la clé **strategy: depend**

```
stages:
```

```
- builds
```

```
build1:
```

```
  stage: builds
```

```
  trigger:
```

```
    include: build.yml
```

```
    strategy: depend
```

```
build:
```

```
  stage: build
```

```
  script:
```

```
    - echo "coucou"
```


LES PIPELINES PARENTS/ENFANTS

- TRES IMPORTANT
- Attention, dans les pipelines enfants, les stages ne sont pas "personnalisable". C'est-à-dire que vous ne pouvez utiliser que
 - .pre
 - build
 - test
 - deploy
 - .post
- Si vous utiliser un nom de stage perso dans l'enfant, comme "monsuperbuild", même si vous l'avez défini en tant que stage dans le parent, vous aurez l'erreur :

build job: chosen stage does not exist; available stages are .pre, build, test, deploy, .post

De plus vous ne pouvez pas utiliser de variables dans la section include.

[Downstream pipelines | GitLab](#)

VARIABLE POUR LES ENFANTS

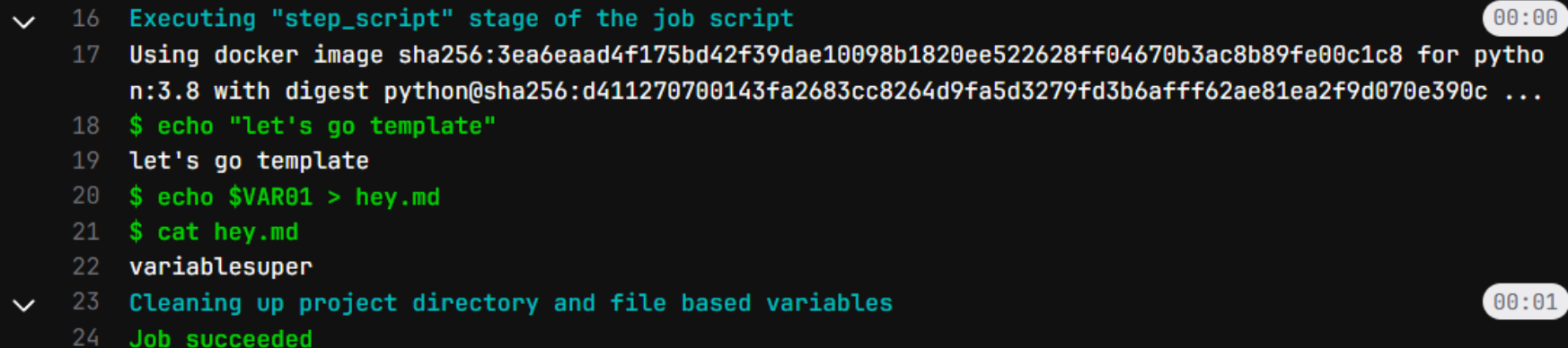
- Pour passer des variables aux enfants, il faut utiliser la clé variables.

Parents

```
build3:
  variables:
    VAR01: variablesuper
  stage: builds
  trigger:
    project: vlaine1/templates-cicd
    strategy: depend
```

Child

```
build-ci_job:
  stage: template
  script:
    - echo "let's go template"
    - echo $VAR01 > hey.md
    - cat hey.md
```



```
16 Executing "step_script" stage of the job script
17 Using docker image sha256:3ea6eaad4f175bd42f39dae10098b1820ee522628ff04670b3ac8b89fe00c1c8 for python:3.8 with digest python@sha256:d411270700143fa2683cc8264d9fa5d3279fd3b6afff62ae81ea2f9d070e390c ...
18 $ echo "let's go template"
19 let's go template
20 $ echo $VAR01 > hey.md
21 $ cat hey.md
22 variablesuper
23 Cleaning up project directory and file based variables
24 Job succeeded
```

NEEDS & TRIGGER

- Vous pouvez envoyer des artefacts utilisable par vos pipelines enfants :
- `needs: install_dependencies` : Cette ligne indique que le job `deploy` dépend du job `install_dependencies` , qui génère l'artefact contenant la variable `VERSION`.
- `trigger` : Utilisé pour déclencher le pipeline enfant, en lui passant la variable d'environnement `VERSION`.

```
install_dependencies:
  stage: install
  script:
    - echo "PYVER=$(python --version)" >> python.env
    - echo "${python --version}"
  artifacts:
    reports:
      dotenv: python.env

deploy:
  needs:
    - install_dependencies # Le job a besoin de
                           # l'artefact du job 'install'
  variables:
    VERSION: $PYVER # Utilisation de la variable
                  # d'environnement créée
  stage: deploy
  trigger:
    project: vlaine1/templates-cicd # Appel du pipeline
    enfant
  strategy: depend
```

```
stages:
  - build_ci

build_ci:
  stage: build_ci
  script:
    - echo "$VERSION" # Affiche la
                      # version passée par le parent
```

PIPELINE DU PROJET PRINCIPAL

```
image: python:3.8

include:
  - project: 'vlaine1/templates-cicd'
    file: 'unit_test.yml'
    ref: 'main'
```

```
stages:
  - builds
  - install
  - test
  - deploy
```

Installation des dépendances

```
build1:
  stage: builds
  trigger:
    include: build.yml
    strategy: depend
```

```
build3:
  variables:
    VAR01: variablesuper
  stage: builds
```

```
trigger:
  project: vlaine1/templates-cicd
  strategy: depend
```

```
install_dependencies:
  stage: install
  script:
    - python -m venv venv
    - source venv/bin/activate
    - pip install -r requirements.txt
    - echo "PYVER=$(python --version)" >> python.env
    - echo "$(python --version)"
```

```
artifacts:
  reports:
    dotenv: python.env
  paths:
    - venv/
```

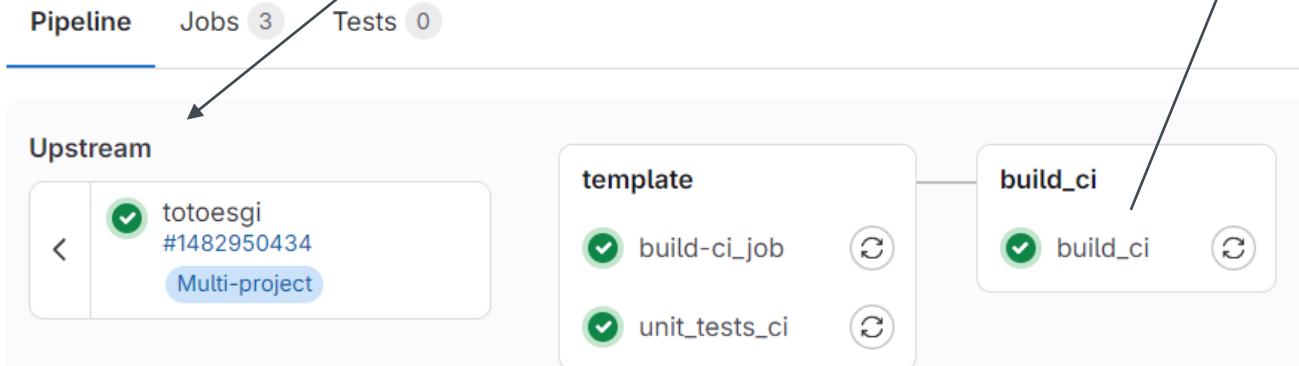
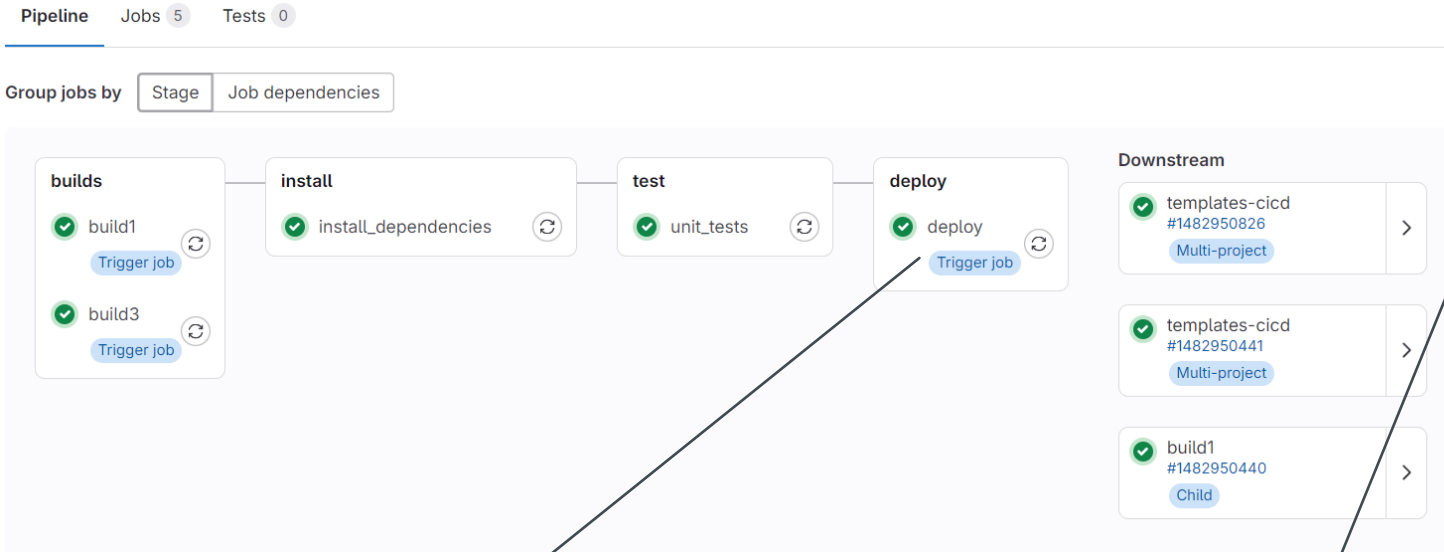
```
cache:
  paths:
    - .cache/pip
```

Exécution des tests unitaires

```
unit_tests:
  stage: test
  script:
    - source venv/bin/activate
    - export PYTHONPATH="$PYTHONPATH:."
    - pytest tests/
  dependencies:
    - install_dependencies
```

```
deploy:
  needs:
    - install_dependencies # Le job a besoin de l'artefact du
                             job 'install'
  variables:
    VERSION: $PYVER # Utilisation de la variable d'environnement
                   créée
  stage: deploy
  trigger:
    project: vlaine1/templates-cicd # Appel du pipeline enfant
    strategy: depend
```

VISUALISATION - MAIN PIPELINE



build_ci

✓ Passed Started 9 minutes ago by Vincent LAINE


```
1 Running with gitlab-runner 17.4.0~pre.110.g27400594 (27400594)
2   on blue-6.saas-linux-small-amd64.runners-manager.gitlab.com/default n
3   ✓ Preparing the "docker+machine" executor
4   Using Docker executor with image python:3.8 ...
5   Pulling docker image python:3.8 ...
6   Using docker image sha256:3ea6eaad4f175bd42f39dae10098b1820ee522628ff04d3b6afff62ae81ea2f9d070e390c ...
7   ✓ Preparing environment
8   Running on runner-nn8vmrs9z-project-62289170-concurrent-0 via runner-nn8vmrs9z-project-62289170-concurrent-0
9   ✓ Getting source from Git repository
10  ✓ Fetching changes with git depth set to 20...
11  Initialized empty Git repository in /builds/vlaine1/templates-cicd/.git/
12  Created fresh repository.
13  Checking out 59d31593 as detached HEAD (ref is main)...
14  Skipping Git submodules setup
15  $ git remote set-url origin "${CI_REPOSITORY_URL}"
16  ✓ Executing "step_script" stage of the job script
17  Using docker image sha256:3ea6eaad4f175bd42f39dae10098b1820ee522628ff04d3b6afff62ae81ea2f9d070e390c ...
18  $ echo "$VERSION"
19  Python 3.8.20
20  ✓ Cleaning up project directory and file based variables
21  Job succeeded
```

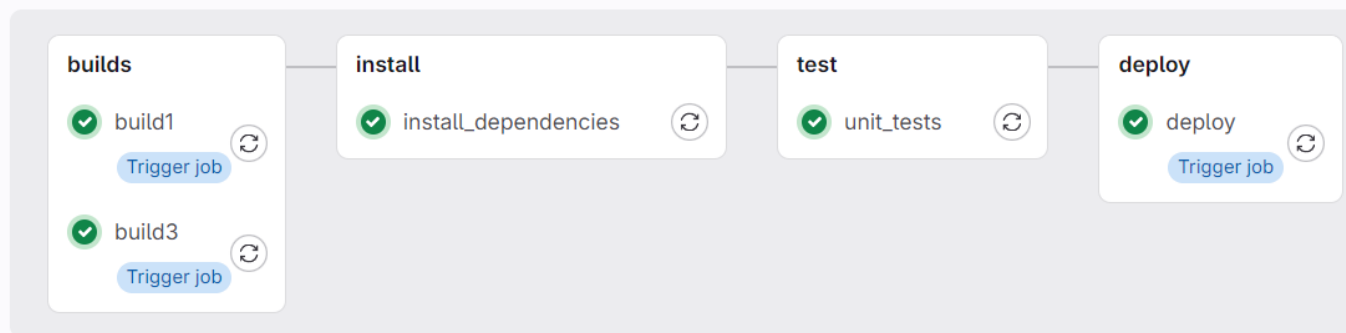
PIPELINE DU PROJET "TEMPLATES"

```
image: python:3.8
include:
  - project: 'vlaine1/templates-cicd'
    file: 'unit_test.yml'
    ref: 'main'
stages:
  - template
  - build_ci
build-ci_job:
  stage: template
  script:
    - echo "let's go template"
    - echo $VAR01 > hey.md
    - cat hey.md
unit_tests_ci:
  stage: template
  extends: .unit_tests
build_ci:
  stage: build_ci
  script:
    - echo "$VERSION" # Affiche la version
      passée par le parent
```



VISUALISATION - TEMPLATE PIPELINE

Upstream

>  totoesgi
#1482950434
Multi-project



build_ci

 Passed Started 5 minutes ago by  Vincent LAINE

```
1 Running with gitlab-runner 17.4.0~pre.110.g27400594 (27400594)
2   on blue-6.saas-linux-small-amd64.runners-manager.gitlab.com
3 Preparing the "docker+machine" executor
4 Using Docker executor with image python:3.8 ...
5 Pulling docker image python:3.8 ...
6 Using docker image sha256:3ea6ead4f175bd42f39dae10098b1820ee1d3b6afff62ae81ea2f9d070e390c ...
7 Preparing environment
8 Running on runner-nn8vmrs9z-project-62289170-concurrent-0 via
9 Getting source from Git repository
10 Fetching changes with git depth set to 20...
11 Initialized empty Git repository in /builds/vlaine1/templates...
12 Created fresh repository.
13 Checking out 59d31593 as detached HEAD (ref is main)...
14 Skipping Git submodules setup
15 $ git remote set-url origin "${CI_REPOSITORY_URL}"
16 Executing "step_script" stage of the job script
17 Using docker image sha256:3ea6ead4f175bd42f39dae10098b1820ee1d3b6afff62ae81ea2f9d070e390c ...
18 $ echo "$VERSION"
19 Python 3.8.20
20 Cleaning up project directory and file based variables
21 Job succeeded
```

VÉRIFICATIONS PARENTS/ENFANTS DE FIN

