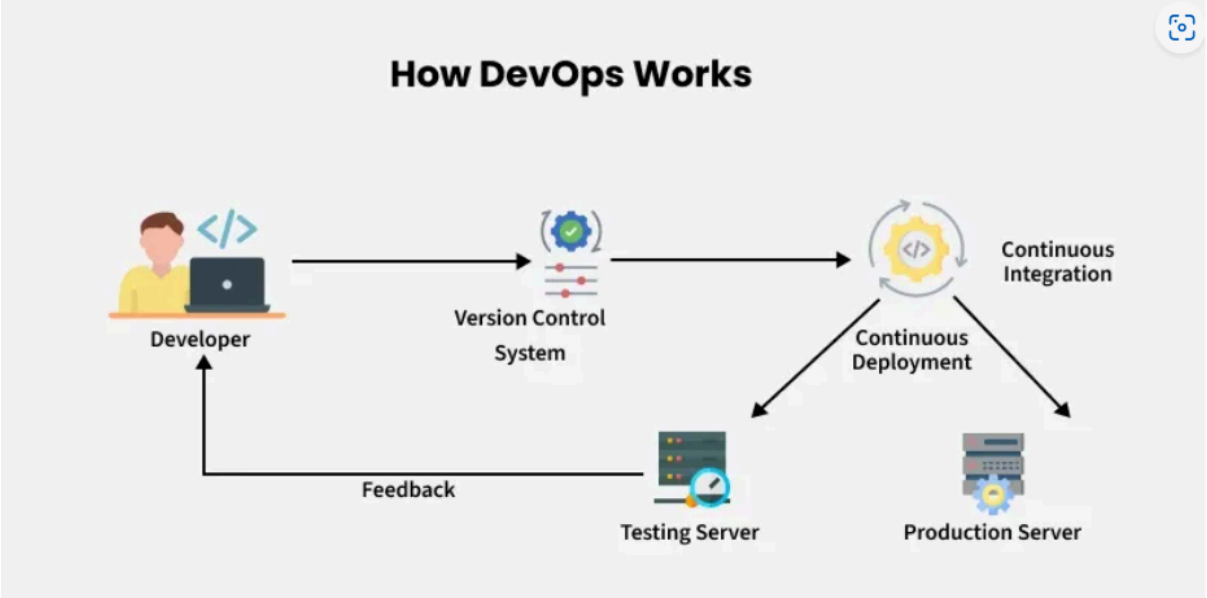
DevOps



Le DevOps est une méthode moderne de travail dans le développement logiciel où l'équipe de développement (qui écrit le code et construit le logiciel) et l'équipe des opérations (qui configure, exécute et gère le logiciel) collaborent comme une seule équipe.

Avant le DevOps, les équipes de développement et d'opérations travaillaient séparément, ce qui entraînait :

- Des retards dans le lancement des logiciels
- Une mauvaise communication entre les équipes

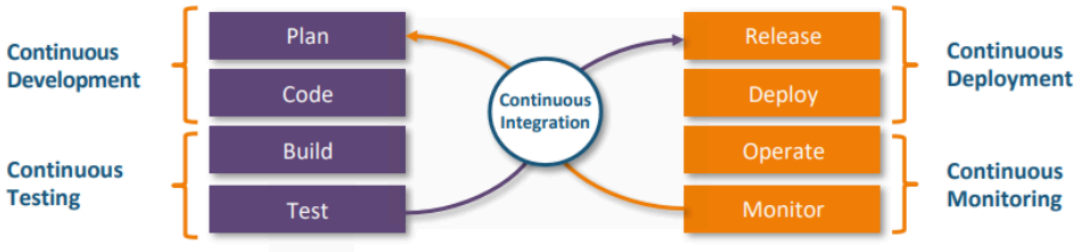
Une lenteur dans la résolution des problèmes

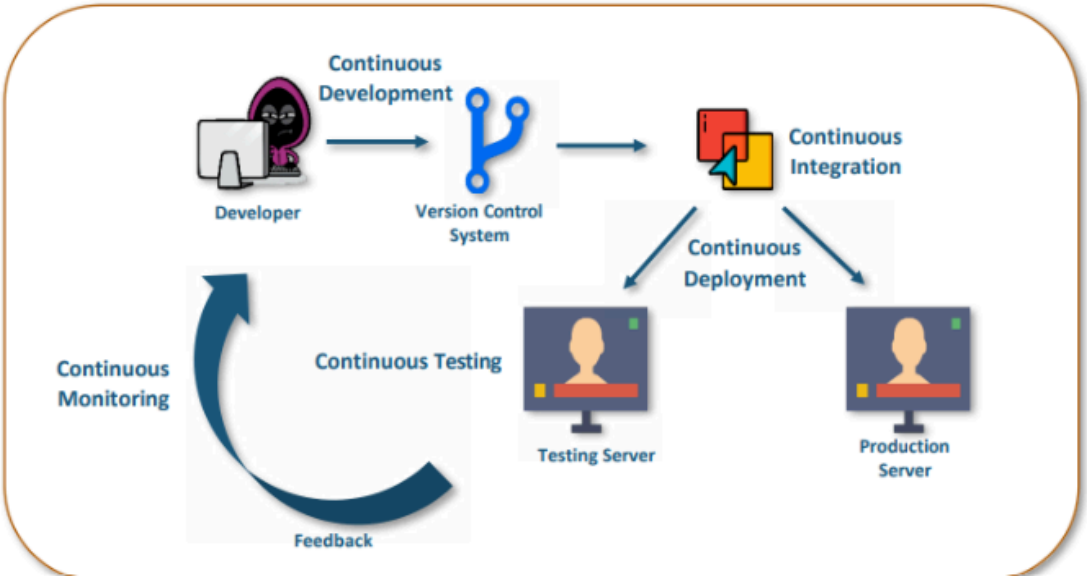
Comment fonctionne le DevOps ?
Le DevOps est une méthode de travail où les équipes de développement, des opérations, d'assurance qualité (QA) et de sécurité collaborent ensemble du début à la fin. Au lieu de travailler séparément, elles travaillent en étroite collaboration pour créer, tester et déployer des logiciels plus rapidement et avec moins d'erreurs.

Le DevOps met également l'accent sur l'automatisation des tâches répétitives, comme les tests et les déploiements, grâce à des outils appelés pipelines CI/CD. Cela rend le processus plus rapide et plus fiable. Les équipes utilisent des outils

utiles comme Git pour la gestion du code, Jenkins pour l'automatisation et Prometheus pour la surveillance.

En combinant le travail d'équipe, l'automatisation et les bons outils, le DevOps permet de livrer des logiciels de haute qualité et sécurisés rapidement et efficacement.





DevOps Life Cycle

1. Développement Continu

Dans le développement continu, le code est écrit par petites itérations régulières plutôt qu'en une seule fois. Cela améliore l'efficacité, car chaque morceau de code est testé, construit et déployé en production dès sa création. Ce processus élève la qualité du code et simplifie la correction des erreurs, vulnérabilités et défauts. Cela permet aux développeurs de se concentrer sur la création d’un code de haute qualité.

2. Intégration Continue

L'intégration continue se déroule en quatre étapes principales dans DevOps :

- **Récupération du code source à partir du SCM** (gestion de contrôle de source).
- **Construction du code** avec des outils comme Maven.
- **Examen de la qualité du code** avec SonarQube.
- **Stockage des artefacts construits** dans un dépôt comme Nexus.

Dans ce flux, les développeurs utilisent un outil comme GitHub pour gérer leur code source. Après avoir développé et poussé leur code, celui-ci est construit sous forme de package (comme `.war`, `.jar`) et soumis à des tests unitaires (JUnit). SonarQube examine la qualité du code, et les artefacts générés sont stockés dans Nexus pour un usage futur. Jenkins orchestre l'ensemble de ce processus.

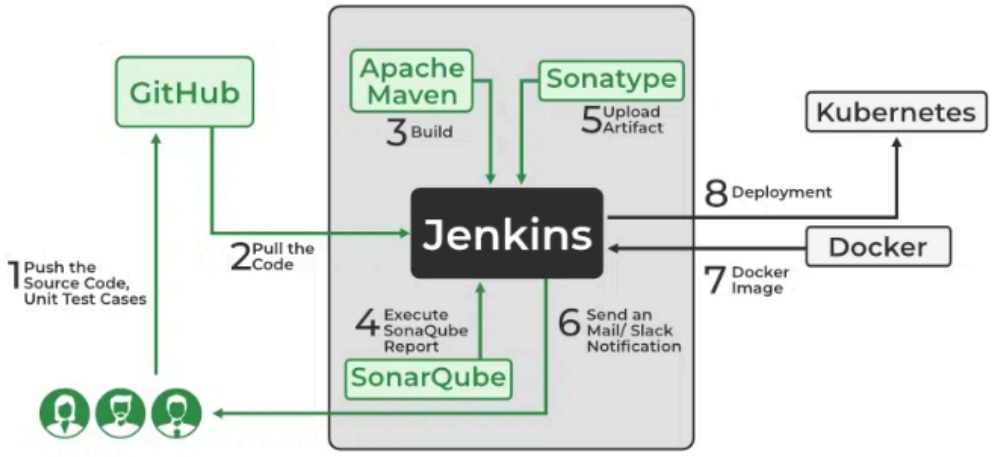
3. Tests Continus

Les tests continus sont déployés en utilisant des outils et méthodologies agiles ou DevOps. Selon les besoins, on peut utiliser des outils d’automatisation comme **Selenium**, **Testsigma** ou **LambdaTest**. Ceux-ci permettent de tester rapidement et intelligemment, d'éviter les problèmes et de détecter les odeurs de code. Avec des outils comme Jenkins, l’automatisation des tests dans un processus d’intégration continue est encore plus fluide.

4. Déploiement Continu / Livraison Continue

- **Déploiement Continu** : C'est le processus d’automatisation complète du déploiement des applications en production une fois les étapes de test et de construction terminées.

- **Livraison Continue** : Ce processus automatise l'intégration continue, mais nécessite une intervention manuelle pour déployer l’application en production.



5. Surveillance Continue

La surveillance continue est essentielle dans le cycle de vie DevOps. À l’aide d’outils comme **Prometheus** et **Grafana**, on peut surveiller en permanence les performances (utilisation CPU, mémoire, trafic réseau, temps de réponse) et recevoir des alertes en cas d'anomalies. Grafana permet de visualiser ces données sous forme de tableaux de bord interactifs.

6. Feedback Continu

Une fois l’application déployée, les utilisateurs finaux fournissent des retours sur ses performances et signalent des bugs ou problèmes. Ces retours sont analysés par l’équipe DevOps et transmis aux développeurs, qui corrigent les erreurs et améliorent la qualité du produit. Le feedback continu réduit les erreurs dans le code et optimise l’expérience utilisateur.

7. Opérations Continues

Les opérations continues visent à maintenir une haute disponibilité des applications tout en minimisant les interruptions dues à la maintenance. Cela améliore la productivité, réduit les coûts de production et garantit un meilleur contrôle qualité.

Phases du cycle de vie DevOps

1. **Planification** : Identifier les besoins métiers et recueillir les retours des utilisateurs. Un plan de projet est conçu pour optimiser l'impact et produire les résultats attendus.

2. **Codage** : Développement du code avec des outils comme Git pour éviter les problèmes de sécurité et améliorer la qualité.

3. **Construction** : Les développeurs soumettent leur code au référentiel commun à l'aide d'outils comme Maven.

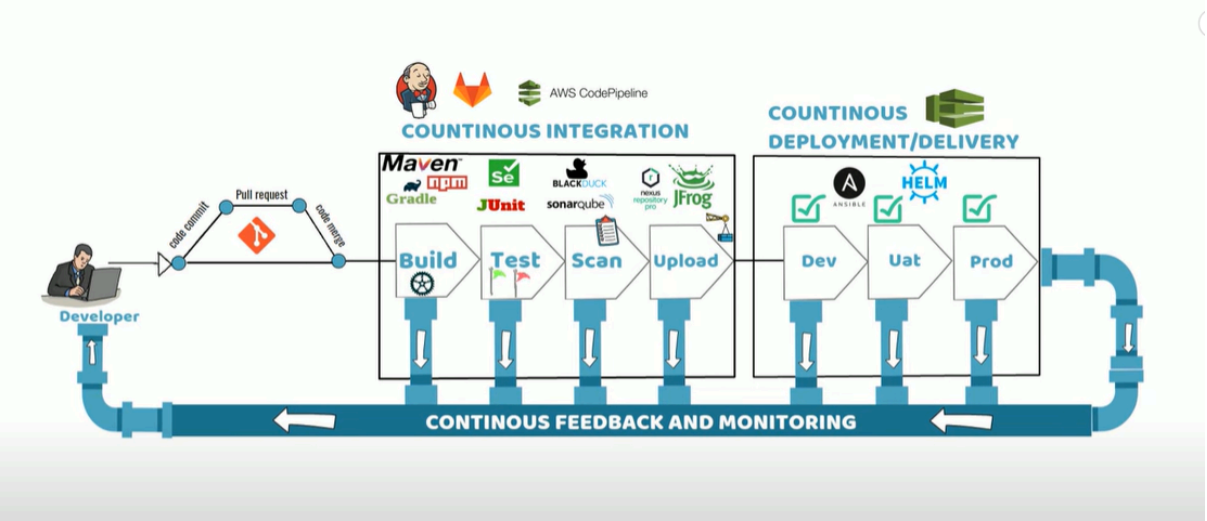
4. **Tests** : Différents types de tests (acceptation, sécurité, intégration, performance) sont effectués avec des outils comme Selenium ou JUnit.

5. **Publication** : Le build validé est préparé pour la production. Les mises à jour ou les versions multiples sont déployées selon les besoins organisationnels.

6. **Déploiement** : Infrastructure-as-Code aide à configurer l'infrastructure et à publier les builds.

7. **Exploitation** : La version est mise à disposition des utilisateurs, avec des outils comme Chef pour la configuration des serveurs.

8. **Surveillance** : Les performances sont analysées grâce aux données utilisateur et autres sources pour identifier et corriger les goulots d'étranglement.



Qu'est-ce que le CI/CD ?

Le CI/CD (Intégration Continue et Livraison/Déploiement Continu) est une pratique DevOps qui automatise et améliore les processus de livraison logicielle. Elle garantit une intégration fluide du code, des tests rigoureux, et un déploiement efficace, permettant des mises à jour rapides et fiables.

Concepts Clés :

1. Intégration Continue (CI)

- Intégration fréquente des modifications de code dans un dépôt partagé.
- Déclenchement automatisé de builds, tests et revues à chaque commit.
- Objectifs : Identifier les bugs tôt, réduire les conflits d'intégration, et améliorer la collaboration.
- **Outils populaires** : Jenkins, GitHub Actions, Bamboo, TeamCity.

2. Livraison Continue (CD)

- Automatisation du déploiement dans des environnements de staging proches de la production.
- Assure que le code est toujours prêt à être publié après avoir passé des tests standardisés.
- Le déploiement en production reste une décision manuelle.

3. Déploiement Continu

- Automatise complètement le déploiement en production après des tests réussis.
- Aucune intervention humaine n'est nécessaire.
- Nécessite des tests rigoureux et une infrastructure robuste.

Étapes d'un Pipeline CI/CD :

1. Étape de Build (Construction)

- Compilation et packaging du code dans un format déployable.
- Intégration des dépendances (bibliothèques externes, outils).
- **Outils courants** : Git, Jenkins, Gradle, Maven.

2. Étape de Test

- Le logiciel subit des tests automatisés pour garantir sa stabilité et détecter les erreurs.
- Types de tests :
    - Tests unitaires : Vérification des petites portions de code.
    - Tests d'intégration : Validation du fonctionnement des modules ensemble.
    - Tests de performance : Analyse de la vitesse et scalabilité.
    - Tests de sécurité : Identification des vulnérabilités.
- **Outils courants** : Selenium, JUnit, TestNG, SonarQube.

3. Étape de Release (Publication)

- Déploiement dans un environnement de staging pour des validations finales.
- Stratégies de déploiement en production :
    - **Blue-Green Deployment** : Alternance entre deux environnements identiques pour minimiser les interruptions.
    - **Canary Deployment** : Déploiement progressif à un petit groupe d'utilisateurs avant un déploiement complet.
    - **Rolling Updates** : Mise à jour progressive pour tous les utilisateurs.
- **Outils courants** : Docker, Kubernetes, Helm, ArgoCD.

4. Boucle de Rétroaction Continue

- Suivi des performances du système et collecte des retours utilisateurs pour améliorer les prochaines versions.
- Surveillance et journalisation des incidents.
- Analyse des versions précédentes pour optimiser les flux de travail.
- **Outils courants** : Prometheus, Grafana, ELK Stack, Datadog.

Sécurité dans le CI/CD :

Risques :

- Dépendances vulnérables.
- Contrôles d'accès insuffisants au pipeline.
- Artéfacts non vérifiés.
- Infrastructure mal configurée.

Solutions :

- Mise en place de politiques d'accès sécurisées.
- Utilisation d'outils d'analyse statique du code (ex. : SonarQube).
- Vérification des images de conteneurs.
- Scans réguliers des vulnérabilités.

Outils CI/CD :

- **CI** : Jenkins, GitHub Actions, Bamboo.
- **CD** : Spinnaker, AWS CodeDeploy, GitLab.
- **Tests** : Selenium, JUnit, Testsigma.

- **Sécurité** : SonarQube, Snyk.
- **Stockage des Artéfacts** : Nexus, Artifactory.
- **Surveillance** : Prometheus, Grafana, ELK Stack.

Infrastructure as Code (IaC)

- **Définition** :

  Au lieu de configurer les serveurs manuellement, les équipes écrivent du code pour les créer et les gérer. Cela rend le processus plus rapide, cohérent et moins sujet aux erreurs.

  Gestion de l'infrastructure (VMs, réseaux, K8s) via du code.
  Outils populaires : Terraform, CloudFormation, Ansible.

- **Concepts Clés** :
  - Approche **déclarative** : Décrire **ce que** l'on souhaite, pas comment le faire.
  - Approche **Impérative** : Définir chaque étape de manière explicite pour atteindre le résultat.
  - **Contrôle de version** : Stocker le code IaC dans GitHub (ex. fichiers YAML Kubernetes).
- **Outils** : Terraform, Ansible, Kubernetes Manifests

Gestion de Configuration

- **Définition** : Configuration automatique des serveurs pour atteindre un état souhaité.
- **Concepts Clés** :
  - **Idempotence** : Exécuter plusieurs fois un script sans causer de problème.
  - Configurations dynamiques avec des variables (ex. `env: prod` ou `env: dev`).
- **Outils** : Ansible, Chef, Puppet

Surveillance et Observabilité

- **Définition** : Suivi des performances des applications et collecte des journaux en temps réel.
- **Concepts Clés** :
    - **Métriques** : CPU, mémoire, latence des requêtes (via Prometheus).
    - **Logs** : Débogage des erreurs (via ELK Stack).
    - **Traces** : Suivi des requêtes dans des microservices (via Jaeger).
- **Outils** : Prometheus, Grafana, Loki
  Voici un résumé en français des concepts demandés :

Containerisation

La containerisation est une technologie qui permet d'isoler une application et toutes ses dépendances dans un environnement léger appelé **conteneur**. Cela garantit que l'application fonctionne de manière cohérente quel que soit l'environnement (développement, test, production).

Fondamentaux de Docker

- **Docker** est une plateforme de containerisation.
- Un **conteneur Docker** est une unité légère et portable qui contient une application et tout ce dont elle a besoin pour fonctionner (bibliothèques, configurations, etc.).
- Une **image Docker** est un modèle immuable à partir duquel on crée un conteneur.
- Docker permet de construire, déployer et exécuter des applications en conteneurs facilement.

Orchestration de conteneurs

L'orchestration permet de gérer le déploiement, la mise à l'échelle, la disponibilité et la gestion des conteneurs dans un environnement distribué.
Les outils principaux sont :

- **Docker Swarm** : système natif Docker pour l'orchestration simple.
- **Kubernetes** : solution plus avancée et largement utilisée pour gérer des clusters de conteneurs à grande échelle.

Docker Compose

- **Usage principal :** Orchestrer plusieurs conteneurs **localement** sur une seule machine (ex. : pour le développement ou les petits déploiements).
- **Fonctionnalités :**
    - Permet de définir et de lancer plusieurs conteneurs avec un fichier `docker-compose.yml`.
    - Gère le réseau, les volumes, et les dépendances entre conteneurs.
- **Limitation :**
    - Ne gère pas le clustering ou la haute disponibilité.
    - Ne supporte pas nativement la mise à l'échelle sur plusieurs machines.

Docker Swarm

- **Usage principal :** Orchestration de conteneurs sur un **cluster de plusieurs machines** (multi-nœuds).
- **Fonctionnalités :**
    - Gestion native des clusters Docker avec plusieurs nœuds (managers et workers).
    - Supporte la mise à l'échelle automatique, la haute disponibilité et la tolérance aux pannes.
    - Équilibrage de charge intégré entre les conteneurs répartis sur différents hôtes.
- **Limitation :**
    - Moins puissant et moins riche en fonctionnalités comparé à Kubernetes, mais plus simple à configurer.

Concepts de registre d’images

Un **registre d’images** est un dépôt centralisé où sont stockées, gérées et distribuées les images Docker.

- **Nexus** ou **Docker Hub** sont des exemples de registres.
- Le registre permet de **pousser** (push) des images construites et de les **tirer** (pull) pour déploiement sur des environnements cibles.
- Il peut gérer des **proxy** pour accéder à d’autres registres et appliquer des règles de contrôle d’accès.

DevOps Tools

1. Git et GitHub

- **Git** : Système de contrôle de version distribué permettant de suivre les modifications du code et de faciliter la collaboration.
- **GitHub** : Plateforme basée sur Git offrant l’hébergement de dépôts, des outils collaboratifs (issues, pull requests) et des fonctionnalités CI/CD comme GitHub Actions.

Stratégies de Branches

- **Branches principales** :
    - **Main (ou Master)** : Code prêt pour la production.
    - **Develop** : Branche dédiée au développement actif.
- **Branches de fonctionnalités** :
    - Chaque fonctionnalité est développée dans une branche distincte.
    - Fusion dans "develop" ou "main" après validation.

GitHub Actions : Workflows

- **Qu'est-ce que GitHub Actions ?**
    - Outil CI/CD intégré à GitHub pour automatiser les workflows.
    - **Définition** : Fichiers YAML dans `.github/workflows/*.yml`.
    - **Déclencheurs** : Push, pull request, planifications, etc.
- **Structure d'un Workflow** :
    - **Déclencheur** (ex. : événement push ou PR).
    - **Jobs** : Ensemble de tâches exécutées (ex. : build, test).
    - **Actions** : Tâches prédéfinies issues du GitHub Marketplace.
- **Cas d'usage courants** :
    - Construire et tester le code à chaque pull request.
    - Déployer des applications après des fusions.
    - Automatiser l'étiquetage des issues ou des PR.

Webhooks et Déclencheurs dans DevOps

- **Qu'est-ce qu'un Webhook ?**
    - Messages automatiques envoyés d'une application à une autre lors d'événements spécifiques.
- **Fonctionnement** :
    - URL Webhook enregistrée sur l'application.
    - Lorsqu'un événement survient, une requête POST est envoyée avec des données associées.
    - Le serveur recevant traite les données et exécute des tâches.
- **Cas d'usage dans DevOps** :
    - **Déclencher des pipelines CI/CD** : Un push GitHub déclenche un pipeline Jenkins.

- **Notifications en temps réel** : Alertes sur Slack ou par email.
- **Intégration d’outils** : Automatisation avec Ansible, Kubernetes, etc.

Combinaison GitHub Actions et Webhooks

- Intégrer des webhooks et GitHub Actions dans les pipelines DevOps pour une automatisation efficace :
  - Exemple : Un webhook GitHub déclenche un pipeline Jenkins qui appelle ensuite un workflow GitHub Actions pour des étapes supplémentaires (tests, artefacts).

Jenkins est un outil open source de serveur d'automatisation. Il aide à automatiser les parties du développement logiciel liées au build, aux tests et au déploiement, et facilite l'intégration continue et la livraison continue. Écrit en Java, Jenkins fonctionne dans un conteneur de servlets tel qu’Apache Tomcat, ou en mode autonome avec son propre serveur Web embarqué.

Il s'interface avec des systèmes de gestion de versions tels que CVS, Git et Subversion, et exécute des projets basés sur Apache Ant et Apache Maven aussi bien que des scripts arbitraires en shell Unix ou batch Windows. Ses fonctionnalités peuvent être étendues facilement à l'aide de plugins supplémentaires qui vous permettront, par exemple, de connecter l'outil à d'autres environnements de test.

Pipeline as Code (Jenkinsfile)

- Jenkins permet de définir des pipelines comme du code dans un fichier nommé **Jenkinsfile**.
- Ce fichier est écrit en **Groovy** et stocké dans le dépôt du projet, ce qui permet une gestion versionnée des pipelines.

Exemple de Jenkinsfile :

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying...'
            }
        }
    }
}
```

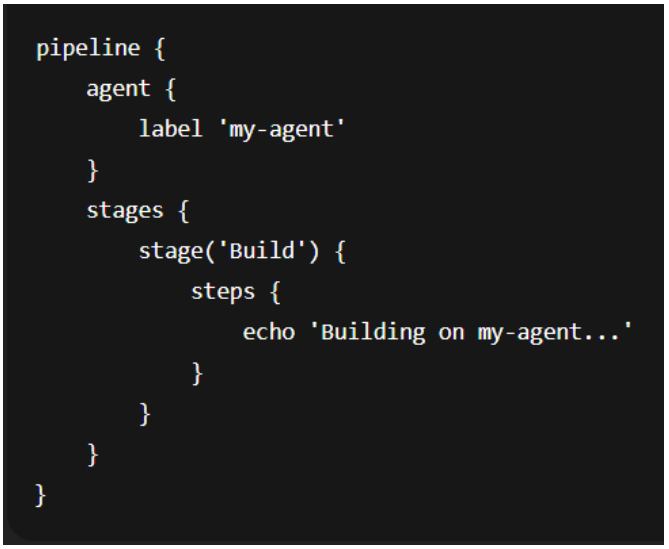
Écosystème de Plugins

- Jenkins dispose d'un **large écosystème de plugins** qui étendent ses fonctionnalités, notamment :
    - **Git** : Pour intégrer les dépôts Git.
    - **Slack** : Pour notifier les équipes.
    - **Pipeline** : Pour la gestion des pipelines as code.
    - **Docker** : Pour la construction et le déploiement de conteneurs.
- Les plugins permettent de personnaliser Jenkins selon les besoins spécifiques des projets.

Configuration des Agents

- Les agents (ou nœuds) exécutent les tâches des pipelines.

- Configuration des agents :
    - **Agent statique** : Toujours disponible et configuré manuellement.
    - **Agent dynamique** : Créé à la demande, souvent avec des solutions comme Kubernetes ou Docker.
- Modes de connexion :
    - **SSH** : Pour connecter un agent à Jenkins.
    - **JNLP** : Utilisé par les agents Java Network Launch Protocol.
- Exemple d'assignation d'un agent dans un pipeline déclaratif :



```
pipeline {
    agent {
        label 'my-agent'
    }
    stages {
        stage('Build') {
            steps {
                echo 'Building on my-agent...'
            }
        }
    }
}
```

5. Nexus (Dépôt d'Artéfacts)

Gestion des Artéfacts (Artifact Management)

Qu'est-ce qu'un Artéfact ?

Un **artéfact** est un fichier binaire ou tout autre composant généré au cours du cycle de vie du développement logiciel.
Exemples :

- **Binaires** : Fichiers JAR, WAR, ou EXE générés par le build.
- **Images Docker** : Conteneurs prêts à être déployés.
- **Fichiers de configuration** ou scripts nécessaires pour le déploiement.

Qu’est-ce que Nexus ?

Nexus est un **gestionnaire de dépôts** qui permet de stocker, gérer et distribuer des artéfacts.

- Il prend en charge plusieurs formats (Maven, NPM, Docker, etc.).
- Utilisé comme **repository central** pour centraliser les artéfacts d’une équipe ou d’une organisation.
- Il simplifie la gestion des dépendances et l’accès aux composants externes.

Dépôt Nexus

Types de Dépôts

1. **Repository Hosted** :
    - Stocke localement les artéfacts générés en interne.
    - Exemples : Dépôts Maven ou Docker privés.
2. **Repository Proxy** :
    - Proxy pour accéder à des dépôts externes comme Maven Central ou Docker Hub.
    - Permet la mise en cache locale pour améliorer les performances.
3. **Repository Group** :
    - Combine plusieurs dépôts (proxy et hosted) sous une seule URL pour simplifier l’accès.

Configurer un Docker Registry avec Nexus

1. **Installer Nexus Repository Manager** :
    - Disponible sous forme de conteneur Docker ou application standalone.
2. **Configurer un Dépôt Docker Hosted** :

- Créez un dépôt Docker pour stocker vos images.
- Activez les **ports HTTP/HTTPS** pour le dépôt.

3. **Configurer Docker pour utiliser Nexus** :
   - Modifiez le fichier `/etc/docker/daemon.json` pour inclure Nexus comme registre sécurisé.

Exemple :

```
{
  "insecure-registries": ["<Nexus-IP>:<Port>"]
}
```

4.
5. **Pousser des Images Docker** :
   - Connectez-vous avec `docker login`.
   - Poussez une image avec `docker push <Nexus-IP>:<Port>/<nom-image>`.

Contrôle d'Accès (Access Control)

1. **Gestion des Utilisateurs et Rôles** :
   - Créez des utilisateurs avec des permissions spécifiques.
   - Attribuez des rôles (lecture, écriture, déploiement).
2. **Authentification** :
   - Intégration avec LDAP ou Active Directory pour la gestion centralisée.
3. **Restrictions** :
   - Définissez des règles d'accès par dépôt (par exemple, certains utilisateurs ne peuvent accéder qu'à des dépôts spécifiques).

Nexus Repository Manager est un outil essentiel pour assurer la gestion, la sécurité et la distribution efficace des artéfacts dans une organisation.

SonarQube : Analyse de Code et Intégration CI/CD

Analyse de Code Statique (Static Code Analysis)

SonarQube est une plateforme qui effectue une **analyse de code statique** pour identifier les problèmes de qualité du code, tels que :

- **Bugs** : Problèmes fonctionnels qui peuvent causer des erreurs d'exécution.
- **Vulnérabilités** : Failles de sécurité dans le code.
- **Code Smells** : Pratiques de codage qui réduisent la lisibilité et la maintenabilité.
- **Endettement Technique** : Évaluation de la complexité du code et du travail nécessaire pour améliorer sa qualité.

Portes de Qualité (Quality Gates)

Les **portes de qualité** permettent de définir des critères minimums que le code doit respecter avant d'être validé :

- Exemple de critères :
    - **Couverture des tests** : Un pourcentage minimum de code testé.
    - **Nombre maximum de bugs** ou vulnérabilités critiques.
    - **Complexité cyclomatique** sous une certaine limite.
- Si les critères ne sont pas remplis, le pipeline de CI peut échouer.

Intégration avec les Pipelines CI/CD

SonarQube s'intègre facilement dans les pipelines CI/CD pour analyser automatiquement le code :

1. **Configuration de l'analyse dans un pipeline CI/CD** :

- Ajout d'une étape dans le pipeline pour exécuter l'analyse avec un scanner SonarQube (SonarScanner).

- Exemple avec Jenkins :

```
stage('SonarQube Analysis') {
    steps {
        script {
            def scannerHome = tool 'SonarQube Scanner'
            withSonarQubeEnv('SonarQube') {
                sh "${scannerHome}/bin/sonar-scanner"
            }
        }
    }
}
```

- Exemple avec GitHub Actions :

```
steps:
  - name: Checkout code
    uses: actions/checkout@v3
  - name: SonarQube Scan
    run: sonar-scanner
    env:
      SONAR_HOST_URL: "http://sonarqube-server"
      SONAR_LOGIN: ${{ secrets.SONAR_TOKEN }}
```

2. **Analyse des rapports** :

   - Les résultats de l'analyse sont envoyés à SonarQube pour une visualisation via son tableau de bord.

   - Les développeurs peuvent consulter les détails des erreurs et suggestions d'amélioration.

3. **Blocage des déploiements** :

   - Si les critères de la porte de qualité ne sont pas remplis, le pipeline peut empêcher le déploiement.

Avantages de SonarQube dans DevOps

- **Amélioration continue** : Détection des problèmes en temps réel lors des commits.
- **Sécurité renforcée** : Identification des vulnérabilités avant déploiement.
- **Standardisation** : Uniformisation des normes de qualité dans les projets.

SonarQube est un outil clé pour garantir un code de haute qualité tout au long du cycle de développement logiciel.

Flux de Travail

1.