Plateforme de CI/CD: Jenkins

Salah Gontara

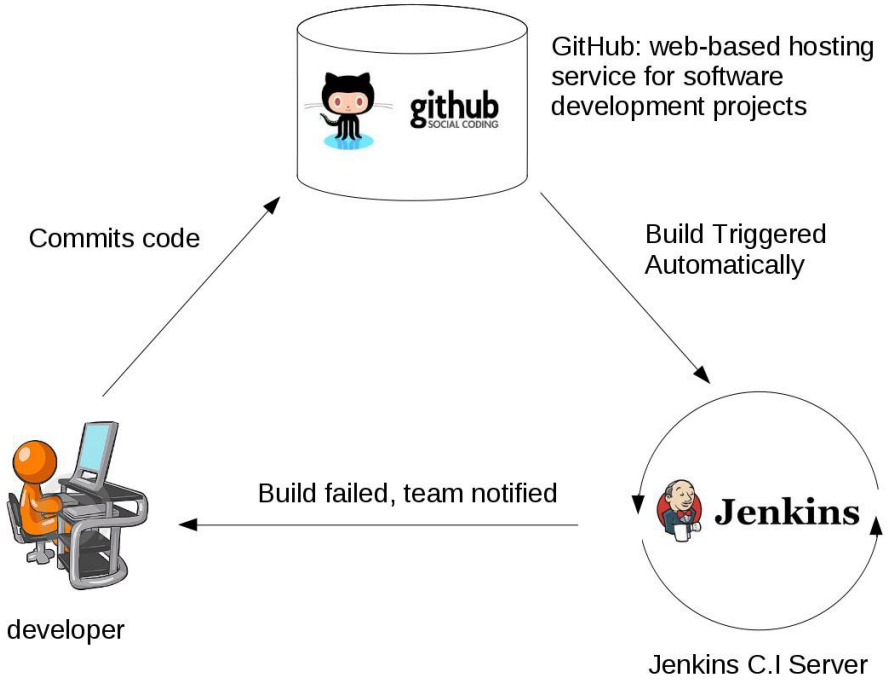
2022-2023

Intégration continue

- Qu’est-ce que l’intégration continue ?
- Pourquoi en avons-nous besoin?
- Différentes phases d’adoption de l’intégration continue

Qu’est-ce que l’intégration continue ?

- Les développeurs valident régulièrement du code dans un référentiel partagé.
- Le système de contrôle de version est surveillé. Lorsqu’une validation est détectée, une build est déclenchée automatiquement.
- Si la version n’est pas verte, les développeurs en seront immédiatement informés.

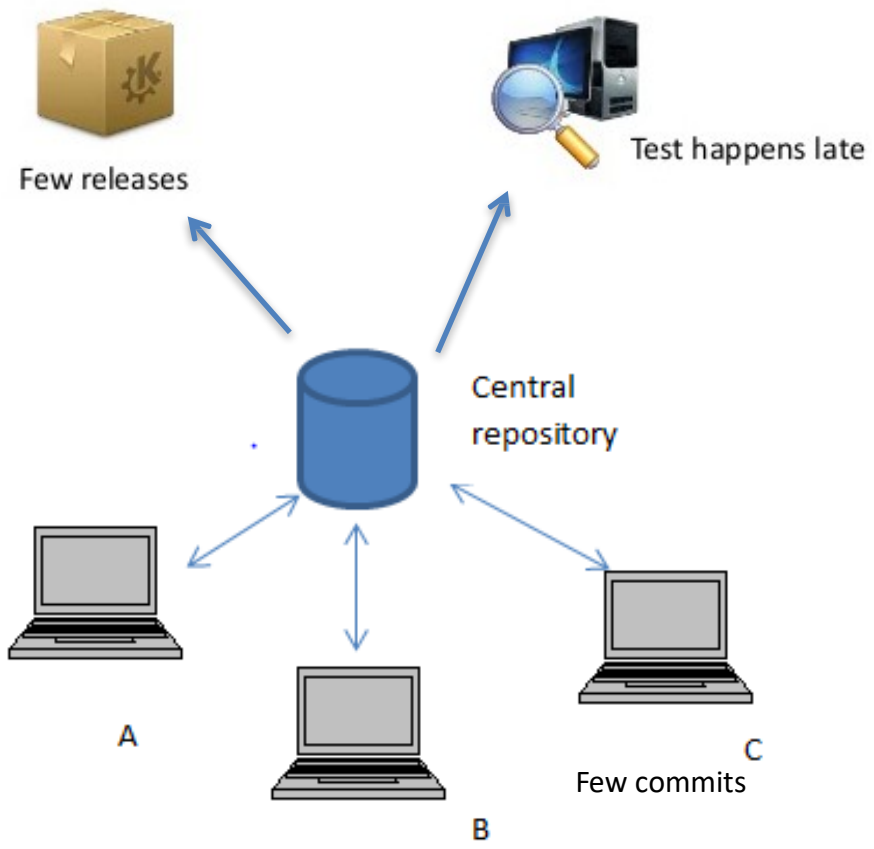


Pourquoi avons-nous besoin d'intégration continue?

- Détectez les problèmes ou les bogues, le plus tôt possible, dans le cycle de vie du développement.
- Étant donné que toute la base de code est intégrée, construite et testée en permanence, les bogues et erreurs potentiels sont détectés plus tôt dans le cycle de vie, ce qui se traduit par un logiciel de meilleure qualité.

Différentes étapes de l’adoption de l’intégration continue

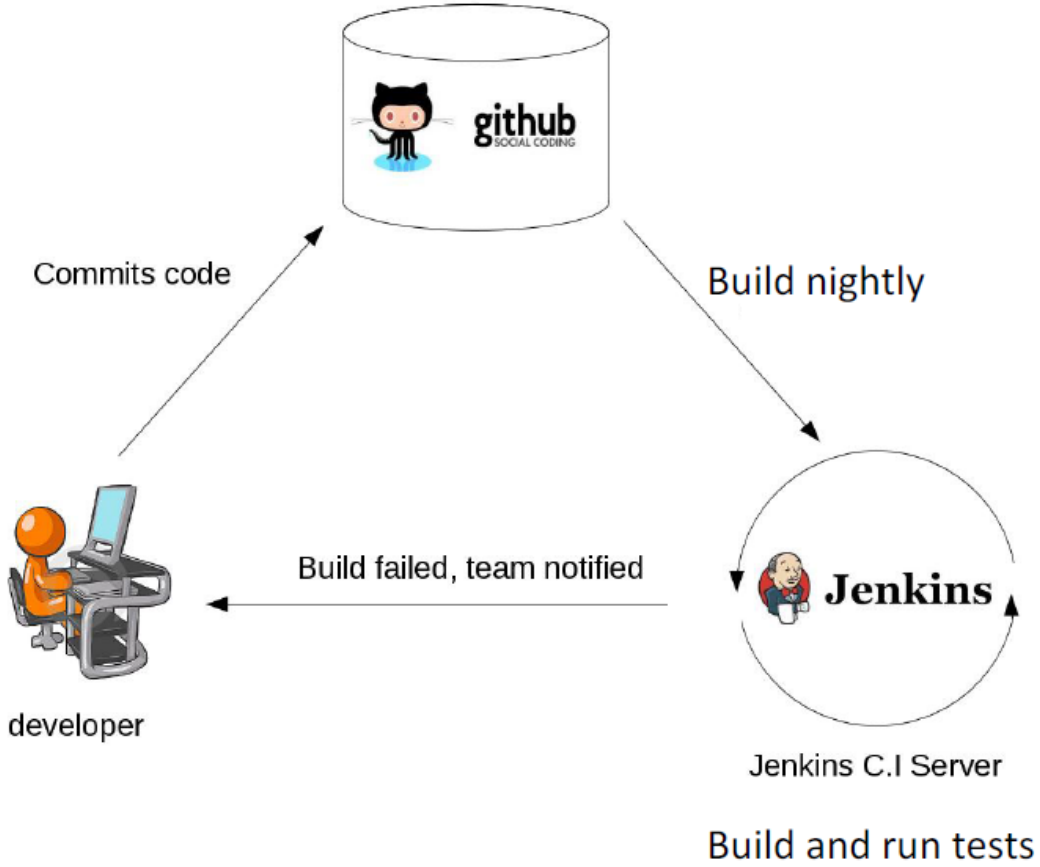
Etape 1:



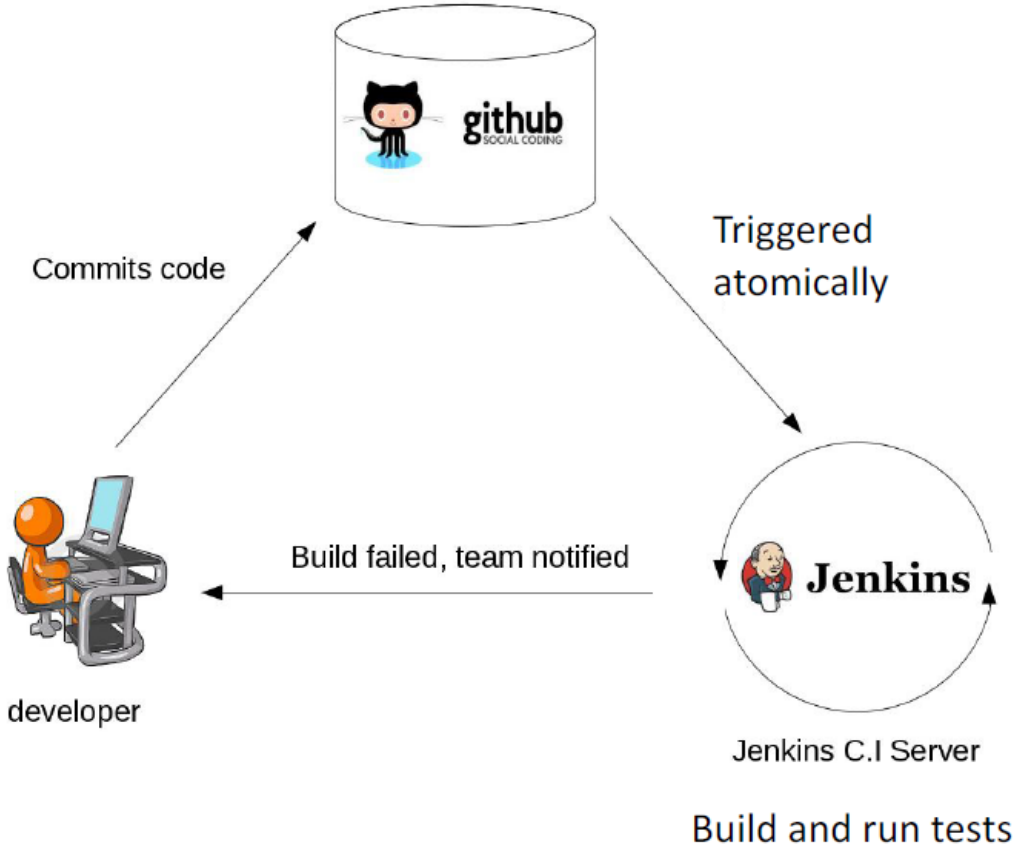
- Pas de serveurs de build.
- Les développeurs s’engagent régulièrement.
- Les modifications sont intégrées et testées manuellement.
- Moins de sorties.

Etape 2:

- Des builds automatisées sont planifiées régulièrement.
- Le script de génération compile l'application et exécute un ensemble de tests automatisés.
- Les développeurs valident désormais régulièrement leurs modifications.
- Les serveurs de build alerteraient les membres de l'équipe en cas d'échec de build.



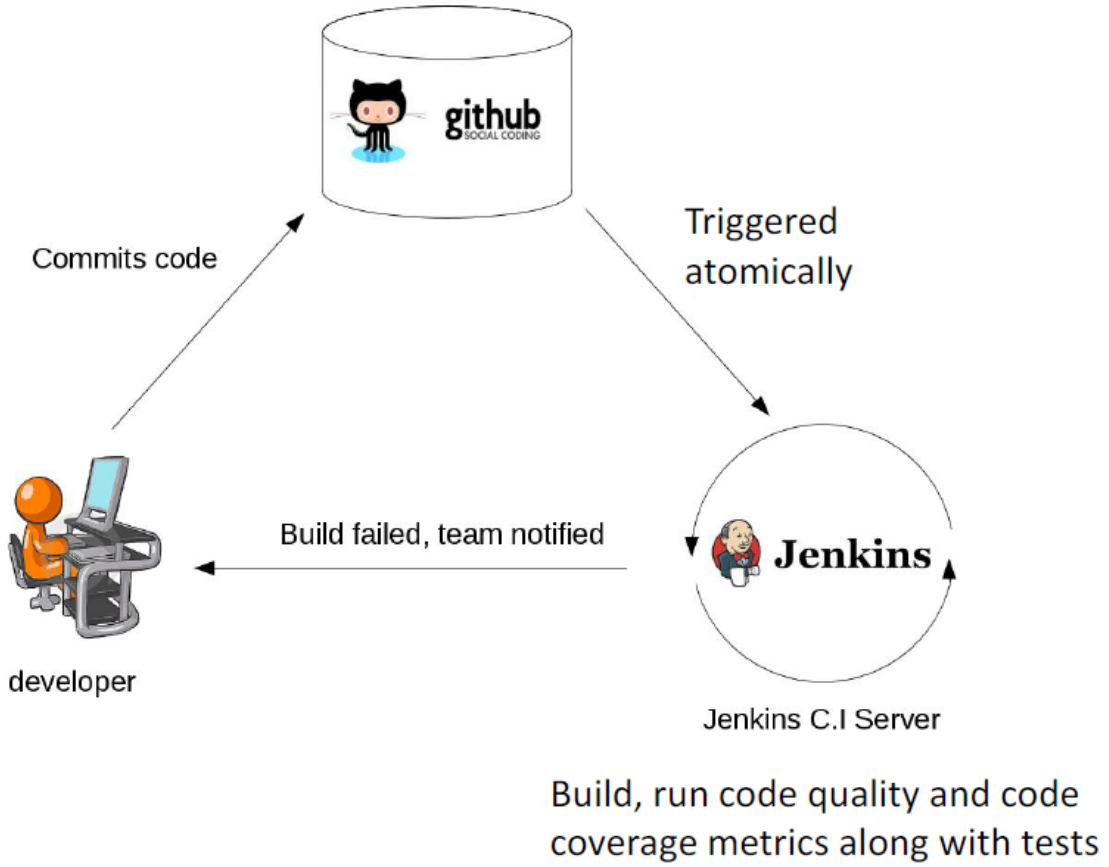
Etape 3:



- Une build est déclenchée chaque fois qu'un nouveau code est validé dans le référentiel central.
- Les builds cassées sont généralement traitées comme un problème de haute priorité et sont corrigées rapidement.

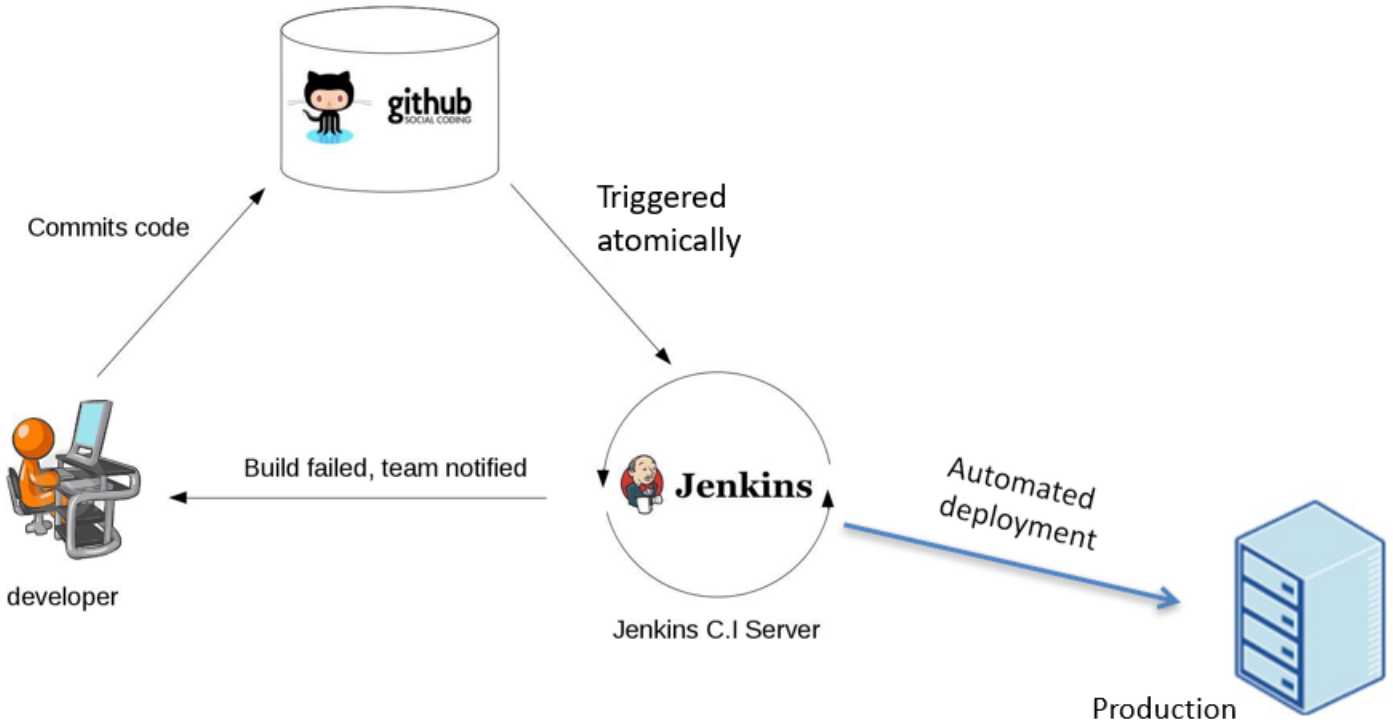
Etape 4:

- Les mesures automatisées de qualité et de couverture du code sont désormais exécutées avec des tests unitaires pour évaluer en permanence la qualité du code.
- La couverture du code augmente-t-elle ?
- Avons-nous de moins en moins d'échecs de build ?

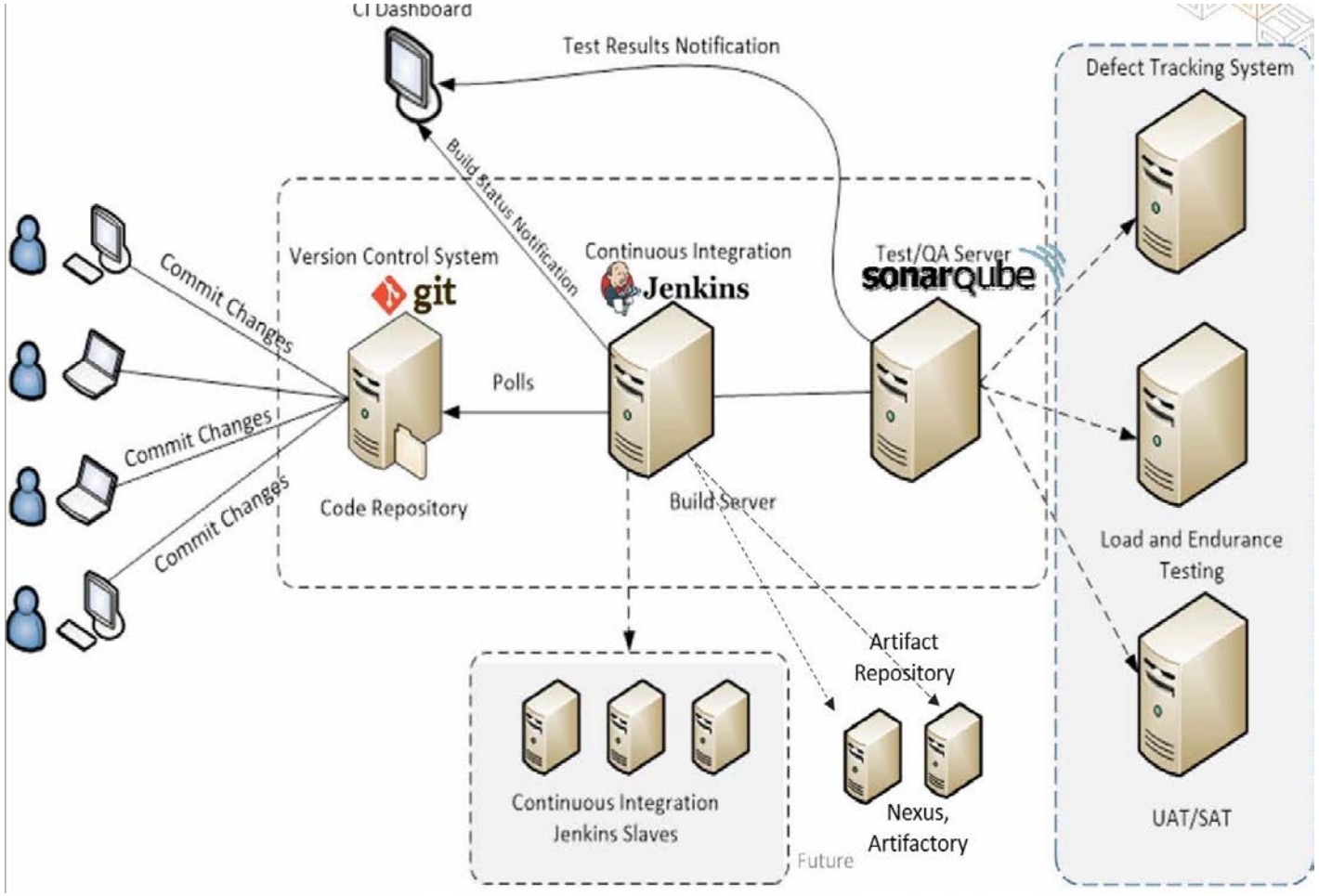


Etape 5:

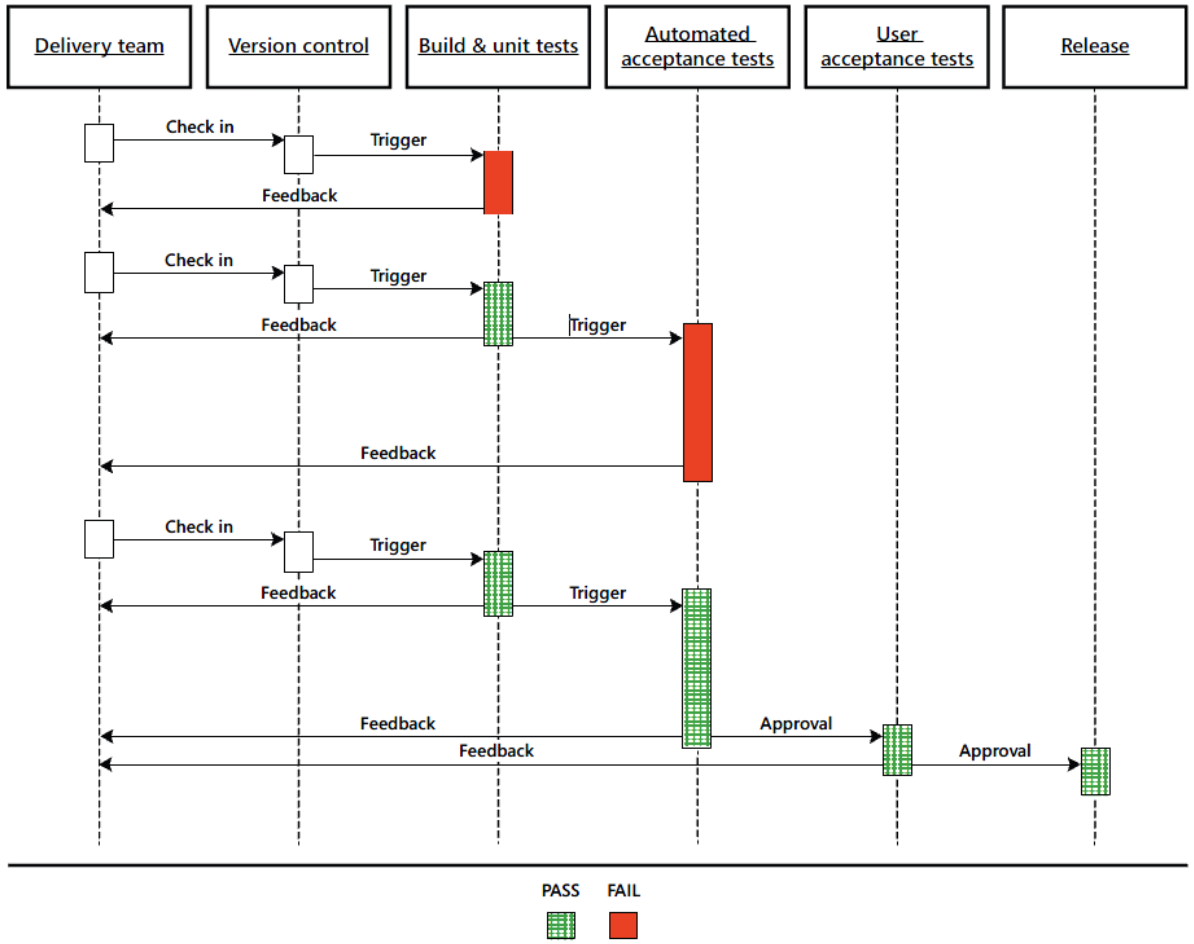
- Déploiement automatisé



Environnement CI/CD: Modélisation



Environnement CI/CD: Diagramme de Séquence



Qu’est-ce que Jenkins?

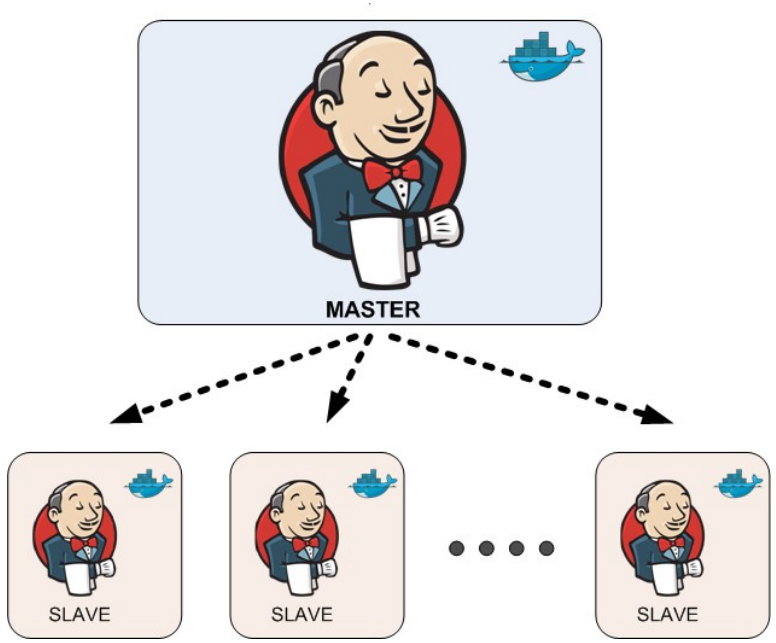
- Jenkins est un serveur d’intégration et de build continue.
- Il est utilisé pour créer manuellement, périodiquement ou automatiquement des projets de développement logiciel.
- Il s’agit d’un outil d’intégration continue open source écrit en Java.
- Jenkins est utilisé par des équipes de toutes tailles, pour des projets avec différentes langages.

Pourquoi Jenkins est Populaire?

- Facile à utiliser
- Grande extensibilité
  - Prise en charge de différents systèmes de contrôle de version
  - Mesures de qualité du code
  - Notifications de build
  - Personnalisation de l’interface utilisateur

Architecture ‘Master’ et ‘Slave’

- **Master:**
  - Crée des tâches.
  - Distribue les builds aux slaves pour l’exécution réelle du travail.
  - Surveille les esclaves et enregistre les résultats de la construction.
  - Peut également exécuter directement des tâches de génération.
- **Slaves**
  - Exécute les tâches de build distribuées par le maître.



Jenkinsfile

- La création d’un fichier **Jenkinsfile**, qui est archivé dans le contrôle de code source, offre un certain nombre d’avantages immédiats:
  - Itération/révision de code sur le pipeline
  - Piste d’audit pour le pipeline
  - Source unique de vérité pour le pipeline, qui peut être visualisée et modifiée par plusieurs membres du projet.

Jenkinsfile: création

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

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

Jenkinsfile: build

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'make' (1)
                archiveArtifacts artifacts: '**/target/*.jar', fingerprint: true (2)
            }
        }
    }
}
```

- L'étape sh appelle la commande make et ne se poursuit que si un code de sortie zéro est renvoyé par la commande. Tout code de sortie différent de zéro échouera dans le pipeline.
- archiveArtifacts capture les fichiers créés correspondant au modèle d'inclusion (**/target/*.jar) et les enregistre sur le contrôleur Jenkins pour une récupération ultérieure.

Jenkinsfile: Test

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Test') {
            steps {
                /* `make check` returns non-zero on test failures,
                * using `true` to allow the Pipeline to continue nonetheless
                */
                sh 'make check || true' (1)
                junit '**/target/*.xml' (2)
            }
        }
    }
}
```

- L’utilisation d’un conditionnel de shell en ligne (sh 'make check || true') garantit que l’étape sh voit toujours un code de sortie nul, donnant à l’étape junit la possibilité de capturer et de traiter les rapports de test.
- junit capture et associe les fichiers XML JUnit correspondant au modèle d’inclusion (**/target/*.xml).

Jenkinsfile: déploiement

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Deploy') {
            when {
              expression {
                currentBuild.result == null || currentBuild.result == 'SUCCESS' (1)
              }
            }
            steps {
                sh 'make publish'
            }
        }
    }
}
```

- L’accès à la variable currentBuild.result permet au pipeline de déterminer s’il y a eu des échecs de test. Dans ce cas, la valeur serait instable.

Les variables d’environnement (1/2)

- BUILD_ID
  - ID de build actuel, identique à BUILD_NUMBER pour les builds créées dans Jenkins versions 1.597+
- BUILD_NUMBER
  - Le numéro de build actuel, tel que « 153 »
- BUILD_TAG
  - Chaîne de jenkins-${JOB_NAME}-${BUILD_NUMBER}. Pratique à mettre dans un fichier de ressources, un fichier jar, etc. pour une identification plus facile
- BUILD_URL
  - URL où se trouvent les résultats de cette génération (par exemple http://buildserver/jenkins/job/MyJobName/17/)
- EXECUTOR_NUMBER
  - Numéro unique qui identifie l’exécuteur actuel (parmi les exécuteurs d’une même machine) exécutant cette génération. C’est le nombre que vous voyez dans l’état de l’exécuteur de build, sauf que le nombre commence à partir de 0, et non de 1

Les variables d’environnement (2/2)

- JENKINS_URL
  - URL complète de Jenkins, telle que https://example.com:port/jenkins/ (REMARQUE : disponible uniquement si l’URL Jenkins est définie dans « Configuration système »)
- JOB_NAME
  - Nom du projet de cette version
- NODE_NAME
  - Nom du nœud sur lequel la version actuelle s’exécute. Réglez sur 'master' pour le contrôleur Jenkins.
- WORKSPACE
  - Le chemin absolu de l’espace de travail

Jenkinsfile: Variables d’environnement (1/2)

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
            }
        }
    }
}
```

Jenkinsfile: Variables d’environnement (2/2)

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any
    environment { (1)
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment { (2)
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

- Une directive d’environnement utilisée dans le bloc de pipeline de niveau supérieur s’appliquera à toutes les étapes du pipeline.
- Une directive environnementale définie au sein d’une étape n’appliquera les variables d’environnement données qu’aux étapes de l’étape.