



Intégration Docker et CI/CD

Optimiser votre chaîne de développement logiciel

Introduction à l'intégration Docker et CI/CD

L'intégration de **Docker** dans une chaîne **CI/CD** (Intégration Continue / Livraison Continue) représente une évolution majeure dans les pratiques DevOps modernes. Cette combinaison permet d'automatiser et d'optimiser le cycle de développement logiciel, de la phase de codage jusqu'au déploiement en production.

En associant la conteneurisation Docker avec des outils CI/CD comme Jenkins, les équipes peuvent créer des pipelines de déploiement cohérents, reproductibles et hautement automatisés.



Docker

Plateforme de conteneurisation qui permet d'empaqueter une application et ses dépendances dans un conteneur isolé.



CI/CD

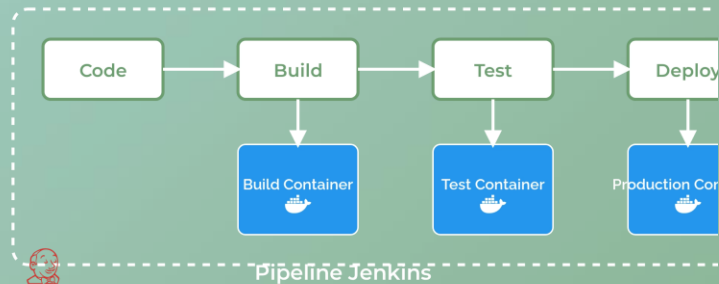
Ensemble de pratiques qui automatisent les processus d'intégration, de test et de déploiement des applications.



Pipeline

Séquence automatisée d'étapes qui permet de transformer le code source en application déployée.

Intégration Docker dans un pipeline CI/CD



Apport des conteneurs dans une chaîne de développement logiciel

Les conteneurs Docker ont révolutionné le développement logiciel en offrant une solution légère et portable pour empaqueter, distribuer et exécuter des applications. Leur intégration dans une chaîne de développement apporte de nombreux avantages qui optimisent l'ensemble du cycle de vie des applications.



Cohérence des environnements

Élimination du problème "ça marche sur ma machine" grâce à des environnements identiques du développement à la production.



Rapidité de déploiement

Démarrage instantané des conteneurs comparé aux machines virtuelles, accélérant les cycles de développement et de test.



Isolation et modularité

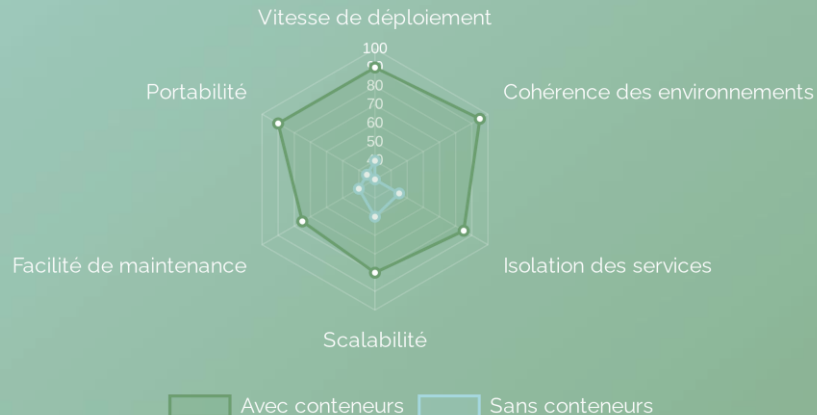
Séparation des composants applicatifs en microservices isolés, facilitant le développement, les tests et la maintenance.



Versionnement des environnements

Possibilité de versionner les environnements complets via des images Docker, garantissant la reproductibilité des builds.

Impact des conteneurs sur le cycle de développement



Infrastructure as Code (IaC) : Les conteneurs permettent de définir l'infrastructure sous forme de code, rendant les environnements reproductibles et versionables.

Intégration avec les outils CI/CD : Docker s'intègre parfaitement avec les outils d'intégration continue comme Jenkins, GitLab CI ou GitHub Actions, permettant d'automatiser les builds, les tests et les déploiements.

Avantages des conteneurs pour le CI/CD

Dans un contexte d'intégration et de livraison continues (CI/CD), les conteneurs Docker apportent des avantages spécifiques qui optimisent chaque étape du pipeline, de l'intégration au déploiement.



Builds reproductibles

Garantie que chaque build s'exécute dans un environnement identique, éliminant les problèmes liés aux différences d'environnement.



Tests isolés et parallèles

Possibilité d'exécuter des tests en parallèle dans des conteneurs isolés, accélérant considérablement la phase de test.



Artefacts standardisés

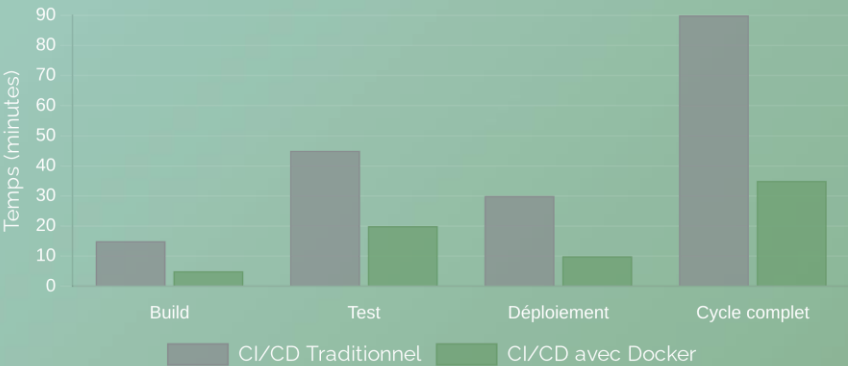
Les images Docker servent d'artefacts de build standardisés, contenant l'application et toutes ses dépendances.



Déploiements simplifiés

Déploiement identique dans tous les environnements, du développement à la production, avec une configuration minimale.

Réduction des temps de cycle CI/CD avec Docker



Caractéristique	CI/CD Traditionnel	CI/CD avec Docker
Cohérence des environnements	×	✓
Isolation des tests	×	✓
Parallélisation facile	×	✓
Artefacts portables	×	✓
Déploiement immuable	×	✓

Intégration de Docker à Jenkins - Vue d'ensemble

L'intégration de **Docker** avec **Jenkins** offre une solution puissante pour créer des pipelines CI/CD modernes et efficaces. Cette combinaison permet d'automatiser les builds, les tests et les déploiements dans des environnements conteneurisés cohérents et reproductibles.

Jenkins peut utiliser Docker de plusieurs façons, créant une synergie qui améliore considérablement le processus de développement logiciel.



Agents Jenkins dans des conteneurs

Exécution des agents Jenkins dans des conteneurs Docker pour des environnements de build isolés et spécialisés.



Build d'images Docker

Création d'images Docker directement dans les pipelines Jenkins pour emballer les applications.



Tests dans des conteneurs

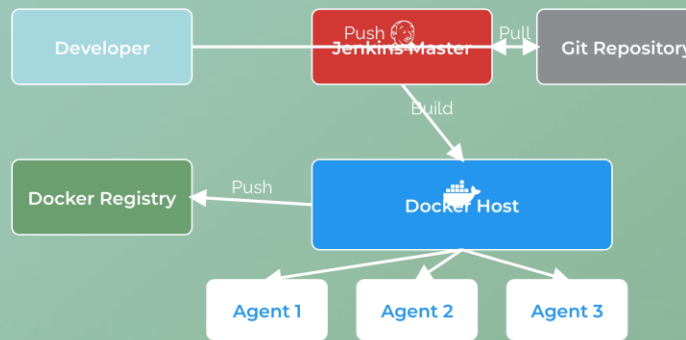
Exécution des tests dans des environnements conteneurisés pour garantir la cohérence et l'isolation.



Publication d'images

Push automatique des images Docker vers des registres après validation des tests.

Architecture d'intégration Docker-Jenkins



Plugins Jenkins pour Docker : Jenkins dispose de plusieurs plugins qui facilitent l'intégration avec Docker :

- Docker Plugin
- Docker Pipeline
- Docker Build Step Plugin
- CloudBees Docker Build and Publish

Ces plugins permettent de configurer facilement des agents Docker, d'exécuter des commandes Docker dans les pipelines et de gérer les images Docker directement depuis Jenkins.

Gestion des esclaves Jenkins avec Docker

L'utilisation de conteneurs Docker comme agents (esclaves) Jenkins offre une solution flexible et évolutive pour exécuter des builds dans des environnements isolés et reproductibles. Cette approche permet de créer dynamiquement des agents à la demande et de les supprimer une fois les tâches terminées.



Provisionnement à la demande

Création automatique d'agents Jenkins dans des conteneurs Docker au moment où un job en a besoin, optimisant l'utilisation des ressources.



Environnements isolés

Chaque build s'exécute dans son propre conteneur, évitant les conflits entre les différentes versions de dépendances ou d'outils.



Nettoyage automatique

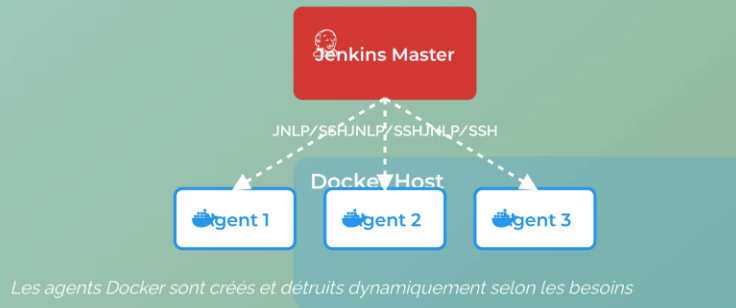
Les conteneurs peuvent être automatiquement supprimés après l'exécution des builds, garantissant un environnement propre pour chaque nouvelle tâche.



Réutilisation des images

Possibilité de créer des images Docker personnalisées pour les agents avec tous les outils et dépendances nécessaires préinstallés.

Architecture maître-esclaves avec Docker



Plugins Jenkins pour Docker: Le plugin "Docker" et le plugin "Docker Pipeline" permettent d'intégrer facilement les conteneurs Docker dans les jobs et pipelines Jenkins.

Méthodes de connexion: Les agents Docker peuvent se connecter au maître Jenkins via JNLP (Java Network Launch Protocol) ou SSH, selon les besoins de sécurité et de configuration.

Orchestration: Pour les environnements plus complexes, l'intégration avec Kubernetes permet une orchestration avancée des agents conteneurisés.

Configuration des agents Docker dans Jenkins

La configuration des agents Docker dans Jenkins permet d'exécuter des builds dans des conteneurs isolés et éphémères, garantissant un environnement propre à chaque exécution.

1 Installation du plugin Docker

Installer les plugins "Docker" et "Docker Pipeline" via le gestionnaire de plugins Jenkins.

2 Configuration du Cloud Docker

Dans "Administrer Jenkins" > "Configurer le système" > "Cloud" > "Ajouter un nouveau cloud" > "Docker".

3 Définition des modèles d'agents

Créer des modèles Docker avec les images appropriées pour chaque environnement de build.

4 Configuration des labels

Attribuer des labels aux modèles pour permettre leur sélection dans les pipelines.



Bonnes pratiques



Utiliser des images Docker officielles pour garantir la sécurité



Monter des volumes pour le cache des dépendances



Définir des ressources (CPU, mémoire) pour chaque agent



Utiliser des agents différents pour chaque étape spécifique

</> Exemple de configuration d'agent Docker dans Jenkinsfile

```
pipeline {
  agent {
    docker {
      image 'maven:3.8.4-openjdk-11'
      args '-v $HOME/.m2:/root/.m2'
      label 'docker-agent'
    }
  }

  stages {
    stage('Build') {
      steps {
        sh 'mvn -B -DskipTests clean package'
      }
    }

    stage('Test') {
      steps {
        sh 'mvn test'
      }
    }
  }
}
```

Pilotage des conteneurs de Docker build depuis Jenkins

Jenkins peut piloter directement la construction d'images Docker, permettant d'intégrer le processus de build des conteneurs dans le pipeline CI/CD. Cette approche automatise la création et la publication des images Docker.

1 Configuration de l'environnement

Installation du plugin Docker Pipeline et configuration des credentials pour les registres Docker.

2 Définition du Dockerfile

Création d'un Dockerfile dans le dépôt de code source qui définit l'image à construire.

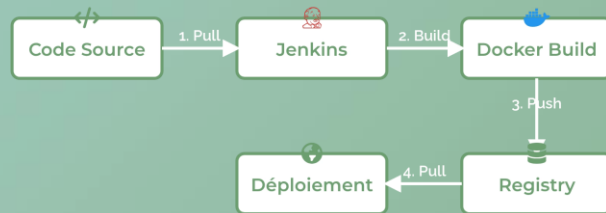
3 Intégration dans le pipeline

Ajout des étapes de build Docker dans le Jenkinsfile pour automatiser la construction.

4 Tests et validation

Exécution de tests sur l'image construite pour valider son fonctionnement.

Flux de travail Docker build dans Jenkins



Pilotage des conteneurs de Docker build depuis Jenkins

</> Exemple de Jenkinsfile avec Docker build

```
pipeline {
  agent any

  environment {
    DOCKER_REGISTRY = 'registry.example.com'
    IMAGE_NAME = 'my-app'
    IMAGE_TAG = "${env.BUILD_NUMBER}"
  }

  stages {
    stage('Build Docker Image') {
      steps {
        script {
          docker.build("${DOCKER_REGISTRY}/${IMAGE_NAME}:${IMAGE_TAG}")
        }
      }
    }

    stage('Test Image') {
      steps {
        script {
          sh "docker run --rm ${DOCKER_REGISTRY}/${IMAGE_NAME}:${IMAGE_TAG} npm test"
        }
      }
    }

    stage('Push to Registry') {
      steps {
        script {
          docker.withRegistry("https://${DOCKER_REGISTRY}", 'docker-registry-credentials') {
            docker.image("${DOCKER_REGISTRY}/${IMAGE_NAME}:${IMAGE_TAG}").push()
            docker.image("${DOCKER_REGISTRY}/${IMAGE_NAME}:${IMAGE_TAG}").push('latest')
          }
        }
      }
    }
  }
}
```

Pipeline Jenkins avec Docker

Un pipeline Jenkins utilisant Docker permet de définir un workflow CI/CD complet et automatisé, où chaque étape s'exécute dans un environnement conteneurisé. Cette approche garantit la cohérence et la reproductibilité du processus de build, de test et de déploiement.



Checkout

Récupération du code source depuis le dépôt Git, généralement exécutée sur le nœud Jenkins principal.



Build

Compilation du code dans un conteneur Docker avec tous les outils nécessaires (Maven, Node.js, etc.).



Test

Exécution des tests unitaires et d'intégration dans des conteneurs isolés, permettant des tests parallèles.



Package

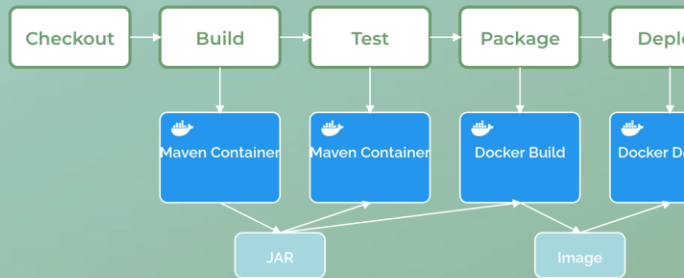
Création d'une image Docker contenant l'application et ses dépendances, prête pour le déploiement.



Deploy

Déploiement de l'image Docker dans l'environnement cible (dev, staging, production).

Pipeline Jenkins avec Docker



Chaque étape s'exécute dans un conteneur Docker spécifique avec les outils nécessaires.

Pipeline Jenkins avec Docker

```
pipeline {
  agent none

  stages {
    stage('Build') {
      agent {
        docker {
          image 'maven:3.8.4-openjdk-11'
        }
      }
      steps {
        sh 'mvn -B -DskipTests clean package'
        stash includes: 'target/*.jar', name: 'app'
      }
    }
    stage('Test') {
      agent {
        docker {
          image 'maven:3.8.4-openjdk-11'
        }
      }
      steps {
        unstash 'app'
        sh 'mvn test'
      }
      post {
        always {
          junit 'target/surefire-reports/*.xml'
        }
      }
    }
  }
}
```

</> Exemple de Jenkinsfile multi-stage avec Docker

```
stage('Build Docker Image') {
  agent any
  steps {
    unstash 'app'
    sh 'docker build -t myapp:${BUILD_NUMBER} .'
  }
}


stage('Deploy') {
  agent any
  steps {
    sh 'docker push myapp:${BUILD_NUMBER}'
  }
}
}
```

Gestion des artefacts sous forme d'images Docker


L'utilisation des images Docker comme artefacts de build représente un changement de paradigme dans la gestion des livrables logiciels. Au lieu de produire des packages spécifiques à une plateforme, les pipelines CI/CD génèrent des images Docker standardisées qui encapsulent l'application et toutes ses dépendances.




Artefacts autonomes
Les images Docker contiennent l'application et toutes ses dépendances, garantissant un comportement identique dans tous les environnements.



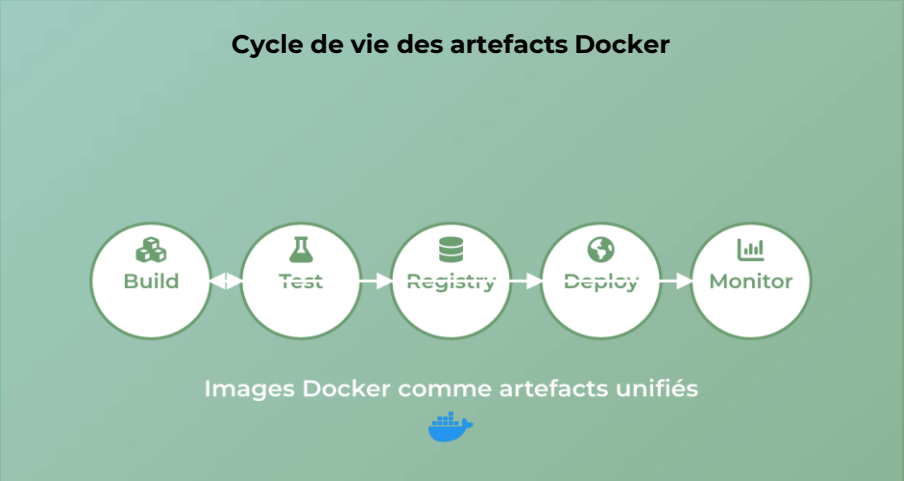
Versionnement efficace
Le système de tags Docker permet de gérer facilement les versions des artefacts et de maintenir un historique complet.



Optimisation par couches
La structure en couches des images Docker permet de réutiliser des composants communs et d'optimiser le stockage et les transferts.



Déploiement immuable
Les images Docker favorisent une approche d'infrastructure immuable où les mises à jour se font par remplacement complet plutôt que par modification.



Caractéristique	Artefacts traditionnels	Images Docker
Portabilité	✗ Limitée	✓ Élevée
Dépendances	✗ Externes	✓ Incluses
Reproductibilité	✗ Variable	✓ Garantie
Déploiement	✗ Complexe	✓ Simplifié
Rollback	✗ Difficile	✓ Instantané

Gestion des artefacts sous forme d'images Docker

Les images Docker constituent une forme standardisée d'artefacts de build qui encapsulent non seulement l'application, mais aussi toutes ses dépendances et son environnement d'exécution. Cette approche offre une solution élégante pour la gestion des artefacts dans un pipeline CI/CD.



Artefacts autonomes

Les images Docker contiennent tout ce qui est nécessaire pour exécuter l'application, garantissant un comportement cohérent dans tous les environnements.



Versionnement intégré

Le système de tags des images Docker permet un versionnement clair et une traçabilité complète des artefacts de build.



Système de couches

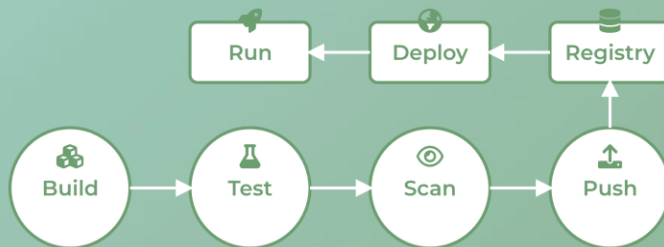
L'architecture en couches des images Docker optimise le stockage et accélère les transferts en ne téléchargeant que les couches modifiées.



Sécurité et validation

Possibilité d'analyser automatiquement les vulnérabilités des images et de signer numériquement les artefacts pour garantir leur intégrité.

Cycle de vie des images Docker dans un pipeline CI/CD



Registre Docker	Public/Privé	Intégration CI/CD	Scan de sécurité
Docker Hub	Les deux	✓	✓
AWS ECR	Privé	✓	✓
Google Container Registry	Privé	✓	✓
GitHub Container Registry	Les deux	✓	✓
GitLab Container Registry	Les deux	✓	✓