INTRODUCTION À LA PIPELINE CI/CD

DÉFINITION DE CI

L'**Intégration Continue** (CI) est une pratique de développement logiciel qui consiste à fusionner fréquemment les changements de code dans un dépôt centralisé et à exécuter des tests automatiques.

PRINCIPES

Avantages	Exemples d'outils CI
Détection rapide d'erreurs	Jenkins, GitLab CI
Collaboration améliorée	Travis CI, CircleCI
Livraison plus rapide	TeamCity, Bamboo

PRINCIPES DE BASE

- **Fusion fréquente** des changements de code
- Exécution de **tests automatiques**
- **Rapports rapides** des problèmes

NOTE

Dans le contexte du DevOps, ces principales pratiques permettent de réduire les risques liés aux erreurs, d'optimiser la gestion du temps et d'assurer une livraison continue et fiable.

AVANTAGES

- Réduction des erreurs de compilation
- Détection rapide des problèmes de qualité
- Livraison du produit plus rapide

1. RÉDUCTION DES ERREURS

L'un des principaux avantages du CI/CD est la réduction des erreurs de compilation.

Grâce à l'intégration continue, le code est compilé et vérifié à chaque modification apportée au référentiel.

1. RÉDUCTION DES ERREURS

Cela permet de détecter rapidement les erreurs de syntaxe, les incompatibilités ou les problèmes de configuration.

1. RÉDUCTION DES ERREURS

Par conséquent, les erreurs sont identifiées et corrigées rapidement, ce qui évite les problèmes de compilation plus tard dans le processus de développement.

EXEMPLE

Imaginons que vous travaillez sur un projet où plusieurs développeurs collaborent. Grâce au CI/CD, chaque fois qu'un développeur pousse du code, le processus de CI vérifie automatiquement si le code compile correctement.

EXEMPLE

Si une erreur de compilation est détectée, une notification est envoyée à l'équipe de développement, permettant ainsi une correction rapide avant que le code ne soit fusionné avec la branche principale.

2. DÉTECTION RAPIDE DES PROBLÈMES

Les tests automatisés sont exécutés à chaque étape du processus de déploiement, ce qui permet de vérifier la conformité du code avec les normes de qualité et les exigences fonctionnelles.

2. DÉTECTION RAPIDE DES PROBLÈMES

Les tests peuvent couvrir différents aspects tels que les tests unitaires, les tests d'intégration et les tests fonctionnels.

2. DÉTECTION RAPIDE DES PROBLÈMES

En détectant rapidement les problèmes de qualité, les équipes de développement peuvent prendre des mesures correctives plus tôt dans le cycle de développement, minimisant ainsi l'impact des problèmes sur la stabilité du logiciel.

EXEMPLE

Supposons que vous développiez une application web e-commerce.

Grâce au CI/CD, des tests automatisés sont exécutés à chaque fois qu'une nouvelle fonctionnalité est ajoutée ou modifiée.

EXEMPLE

Ces tests vérifient si les fonctionnalités existantes ne sont pas impactées et si les nouvelles fonctionnalités fonctionnent conformément aux attentes.

EXEMPLE

Ainsi, si une modification introduit un bogue ou affecte d'autres parties du système, il est détecté rapidement, permettant une intervention immédiate.

3. LIVRAISON DU PRODUIT PLUS RAPIDE

Le CI/CD permet d'accélérer la livraison du produit en automatisant les tâches de déploiement et de tests.

Les déploiements continus garantissent que les changements validés sont rapidement déployés sur l'environnement de production.

EXEMPLE

Dans un contexte de CI/CD, vous pouvez configurer des pipelines d'intégration et de déploiement automatisés qui permettent de livrer rapidement les changements.

Supposons que vous ayez ajouté une nouvelle fonctionnalité à votre application.

EXEMPLE

Grâce au CI/CD, cette fonctionnalité est automatiquement déployée sur l'environnement de test, où des tests automatisés sont exécutés pour vérifier son bon fonctionnement.

EXEMPLE

Si les tests réussissent, la fonctionnalité est ensuite déployée sur l'environnement de production, prête à être utilisée par les utilisateurs.

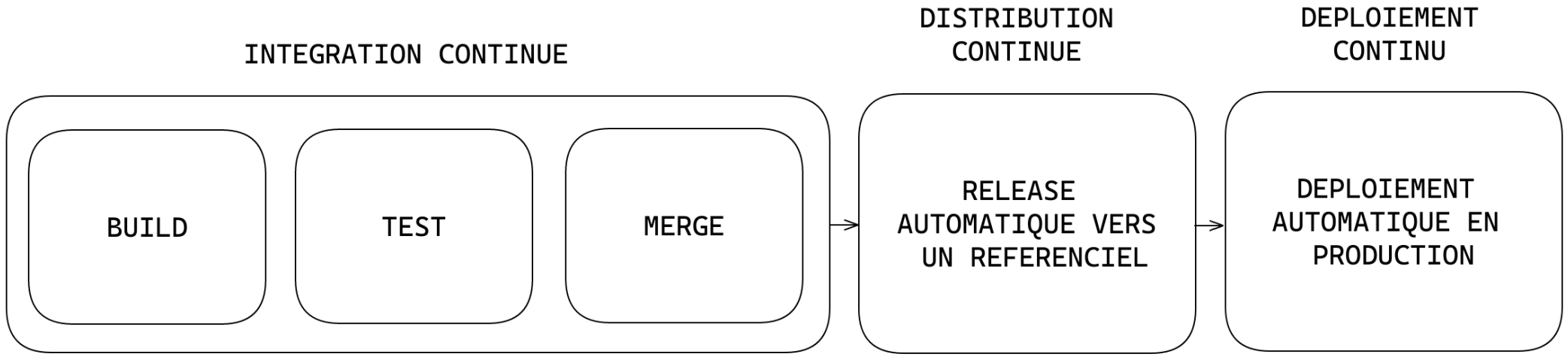
DÉFINITION DE CD

La **Livraison Continue** (Continuous Delivery, CD) / **Déploiement Continu** (Continuous Deployment, CD) sont des pratiques étroitement liées.

DIFFÉRENCE ENTRE LIVRAISON ET DÉPLOIEMENT CONTINU

- **Livraison Continue**: Déployer automatiquement les changements jusqu'à un certain stade (pré-production)
- **Déploiement Continu**: Déployer automatiquement les changements jusqu'en production

SCHEMA


INTEGRATION CONTINUE
BUILD
TEST
MERGE
DISTRIBUTION CONTINUE
RELEASE AUTOMATIQUE VERS UN REFERENCIEL
DEPLOIEMENT CONTINU
DEPLOIEMENT AUTOMATIQUE EN PRODUCTION

LA PLANIFICATION AVANT LA PIPELINE CI/CD

PLANIFICATION

La **planification** est la première étape de la **Pipeline CI/CD**. Elle comprend la gestion des **exigences**, l'estimation du **travail** et la **planification des sprints**.

GESTION DES EXIGENCES

La gestion des exigences consiste à déterminer les **besoins des utilisateurs** et à les transformer en **fonctionnalités à implémenter**.

GESTION DES EXIGENCES

Elle garantit une bonne communication entre les différents acteurs du projet et guide les choix techniques.

EXEMPLE

EXIGENCE : "L'APPLICATION DOIT PERMETTRE AUX UTILISATEURS DE CRÉER UN COMPTE ET DE SE CONNECTER."

EXEMPLE

- Capturez cette exigence en détail en spécifiant les fonctionnalités attendues, telles que la création de compte avec des informations personnelles, la vérification des identifiants lors de la connexion, etc.

EXEMPLE

- Utilisez des techniques de documentation appropriées, telles que les cas d'utilisation, les diagrammes de séquence ou les histoires utilisateur, pour clarifier cette exigence.

USER STORIES

Les user stories sont un outil de communication clé utilisé dans les méthodologies de développement de logiciels agiles comme Scrum et XP.

USER STORIES

Elles aident à concentrer le développement sur l'utilité réelle que le produit apportera à l'utilisateur final.

CRÉATION DE USER STORIES

- Format standard : "En tant que **[rôle]**, je veux **[objectif]** afin de **[raison]**"
- Contient des **critères d'acceptation**

Les user stories doivent être claires, concises et pas trop complexes.

SYNTAXE

Une user story est généralement écrite selon ce format :

```
En tant que <type d'utilisateur>, je veux <objectif> afin que <bénéfice>.
```

COMMENT ÉCRIRE UNE USER STORY?

IDENTIFIEZ L'UTILISATEUR

Qui est l'utilisateur de cette fonctionnalité ? Il pourrait s'agir d'un utilisateur final, d'un administrateur système, etc.

DÉFINISSEZ L'OBJECTIF

Qu'est-ce que l'utilisateur veut réaliser ? Quelle est la fonctionnalité qu'il souhaite ?

IDENTIFIEZ LE BÉNÉFICE

Pourquoi l'utilisateur veut-il cette fonctionnalité ? Quel est le bénéfice qu'il obtiendra en l'utilisant ?

EXEMPLE

Voici un exemple de user story :

```
En tant qu'utilisateur de l'application bancaire, je veux pouvoir consulter mon solde actuel afin de gérer mes finances.
```

INVEST CRITERIA

Les bonnes user stories suivent généralement le critère INVEST

INDÉPENDANT

Chaque user story doit être indépendante des autres. Cela permet de les développer et de les prioriser de manière indépendante.

NÉGOCIABLE

Une user story n'est pas une spécification détaillée. Elle est plutôt une conversation ouverte entre l'équipe de développement et le propriétaire du produit.

VALUABLE

Chaque user story doit apporter de la valeur à l'utilisateur final.

ESTIMABLE

Une user story doit être suffisamment claire et concise pour que l'équipe de développement puisse estimer le temps qu'il faudra pour la développer.

SMALL

Les user stories doivent être suffisamment petites pour pouvoir être développées dans un sprint.

TESTABLE

Chaque user story doit avoir des critères d'acceptation clairs pour savoir quand elle a été correctement mise en œuvre.

CONCLUSION

Les user stories aident à garder l'accent sur la valeur que le produit apporte à l'utilisateur final, ce qui facilite la création de logiciels qui répondent aux besoins et aux désirs de leurs utilisateurs.

CODAGE

INTRODUCTION

Dans cette étape, les **développeurs** écrivent et révisent le **code source**, en utilisant des **outils** pour gérer les différentes versions et résoudre les conflits.

ETAPES

Codage	Outils
Écriture du code	Éditeurs de code
Révision	Revue de code
Gestion des versions	Git, SVN
Résolution des conflits	Merge tools

NOTE

Il est important d'utiliser des méthodes de travail standardisées, telles que des conventions de codage, pour assurer la lisibilité et la maintenabilité du projet.

VERSIONING

Le **versioning** consiste à gérer et suivre les modifications apportées au **code source** au fil du temps.

- Permet le travail collaboratif
- Facilite le retour en arrière si nécessaire
- Garde une trace des modifications et de leurs auteurs

SYSTÈMES DE CONTRÔLE DES VERSIONS (GIT, SVN, ETC.)

Il existe plusieurs systèmes de contrôle des versions, tels que **Git**, **SVN**, **Mercurial**, etc. Ils permettent de suivre les modifications du code et de revenir à une version précédente si nécessaire.

CONTRÔLE DE VERSIONS

Système de contrôle des versions	Avantages	Inconvénients
Git	- Décentralisé - Rapide - Facile à apprendre	- Interface utilisateur parfois complexe
SVN	- Centralisé - Historique de commits linéaire	- Moins performant que Git
Mercurial	- Similaire à Git - Facilité d'utilisation	- Moins populaire que Git

BRANCHES ET FUSION DE CODES

Les **branches** sont des copies du code source qui permettent de travailler sur une **fonctionnalité spécifique** ou une **erreur** sans affecter le code principal.

Une fois la tâche terminée, la branche est fusionnée avec la **branche principale**.

AVANTAGES

Avantages des branches	Exemple d'utilisation
Isolation du travail	Fonctionnalités spécifiques
Facilité de fusion	Correction de bugs
Gestion des versions	Expérimentations

GESTION DES CONFLITS

Les **conflits** surviennent lorsque deux développeurs apportent des modifications **incompatibles** au même fichier.

Les systèmes de **contrôle des versions** permettent de résoudre ces conflits **manuellement** ou **automatiquement**.

REVUE DE CODE

La revue de code est un processus d'**inspection** du code source d'une **application** pour s'assurer qu'il respecte les **bonnes pratiques** de codage et qu'il ne contient pas d'**erreurs**.

BONNES PRATIQUES DE CODAGE

Les bonnes pratiques de codage comprennent :

- Utilisation de **noms de variables** et de **fonctions clairs**
- Le respect des principes **DRY** (Don't Repeat Yourself) et **SOLID**
- L'écriture de **commentaires** pour expliquer le fonctionnement du code.

COMPILATION

OUTILS DE COMPILATION

Pour compiler le code source en code exécutable, on utilise des **outils de compilation** comme :

OUTILS

Langage	Outils de compilation
Java	javac, Maven, Gradle
C++	gcc, g++, clang
C#	csc

GESTION DES DÉPENDANCES

La **gestion des dépendances** permet de contrôler et gérer les **bibliothèques externes** requises par votre projet. Des outils populaires sont :

OUTILS

Langage	Outils
Pour Java	Maven et Gradle
Pour JavaScript	npm et Yarn
Pour Python	pip et conda
Pour C	Makefile

TESTS

TYPES DE TESTS

Il existe plusieurs types de tests qui peuvent être automatisés dans une pipeline **CI** :

TESTS UNITAIRES

Vérifient le bon fonctionnement des unités logiques individuelles

TESTS D'INTÉGRATION

Vérifient la cohérence entre différentes parties du système

TESTS FONCTIONNELS

Vérifient si les fonctionnalités répondent aux exigences des utilisateurs

TESTS UNITAIRES

Supposons que vous développiez une application financière qui a une fonctionnalité pour calculer l'intérêt composé.

TESTS UNITAIRES

Un test unitaire pour cette fonction pourrait être de vérifier que la formule mathématique est correctement implémentée en fournissant des valeurs d'entrée et en vérifiant si la sortie correspond à la valeur attendue.

TESTS UNITAIRES

Par exemple, si l'entrée est un capital initial de 1000€, un taux d'intérêt annuel de 5%, et une durée de 5 ans, alors la sortie attendue, en utilisant la formule de l'intérêt composé, devrait être 1276,28€.

TESTS D'INTÉGRATION

Supposons que votre application ait une fonctionnalité qui permet à l'utilisateur de sauvegarder des transactions financières dans une base de données et de les récupérer plus tard.

TESTS D'INTÉGRATION

Un test d'intégration pourrait vérifier si l'API de l'application et la base de données interagissent correctement.

TESTS D'INTÉGRATION

Par exemple, vous pouvez créer une nouvelle transaction via l'API, la sauvegarder dans la base de données, puis la récupérer et vérifier si les données correspondent à ce que vous avez enregistré.

TESTS FONCTIONNELS

Ces tests sont axés sur la fonctionnalité globale de l'application du point de vue de l'utilisateur. Supposons que votre application ait un processus d'inscription.

TESTS FONCTIONNELS

Un test fonctionnel pourrait consister à remplir le formulaire d'inscription avec des données d'utilisateur valides, à soumettre le formulaire, et à vérifier si l'utilisateur est correctement enregistré et peut se connecter avec les informations d'inscription fournies.

INTÉGRATION

STRATÉGIES D'INTÉGRATION

Il existe plusieurs stratégies pour intégrer le travail de différents développeurs :

- **Intégration régulière** : intégrer le code au moins une fois par jour
- **Feature branches** : créer une branche pour chaque fonctionnalité et la merger lorsqu'elle est prête
- **Pull requests** : soumettre le code pour revue avant de l'intégrer dans la branche principale

SERVEURS D'INTÉGRATION CONTINUE

Un serveur d'intégration continue est un outil qui **automatise** les étapes de **compilation**, de **test** et d'**intégration**. Exemples :

- **Jenkins**
- **GitLab CI/CD**
- **GitHub Actions**

LIVRAISON CONTINUE (CD)

La **livraison continue** (Continuous Delivery) est une approche pour automatiser le processus de livraison de logiciels aux environnements de production.

ENVIRONNEMENTS DE DÉPLOIEMENT

Environnement	Description
Développement	Utilisé pour les tests internes et le développement
Test	Utilisé pour valider les changements et exécuter des tests automatisés
Pré-production	Réplique de l'environnement de production pour effectuer des tests finaux
Production	Environnement où les utilisateurs finaux utilisent l'application

MAINTENANCE

GESTION DES INCIDENTS

La **gestion des incidents** fait référence au processus de **résolution des problèmes** rencontrés dans un système en **production**.

ETAPES

Étape	Description
1. Détection	Identifier rapidement les problèmes
2. Diagnostique	Déterminer la cause du problème
3. Réparation	Corriger les erreurs et rétablir le système
4. Prévention	Appliquer des mesures pour éviter de futurs incidents

MISES À JOUR ET MISES À NIVEAU

Les mises à jour et les mises à niveau sont essentielles pour maintenir un système **performant**, **sécurisé** et à jour avec les **nouvelles fonctionnalités**.

MAJ VS MISE À NIVEAU

- Mise à jour : appliquer des correctifs et des améliorations mineures
- Mise à niveau : introduire de nouvelles fonctionnalités et de grandes modifications

MAJ VS MISE À NIVEAU

Mises à jour	Mises à niveau
Correctifs de sécurité	Nouvelles fonctionnalités
Améliorations mineures	Modifications majeures
Maintenance régulière	Changements dans l'architecture

RÉTROCOMPATIBILITÉ

- La **rétrocompatibilité** garantit que les nouvelles mises à jour ne perturbent pas les fonctionnalités existantes

TEST DE NON-RÉGRESSION

- Les **tests de non-régression** sont utilisés pour s'assurer que les modifications apportées n'ont pas d'effets indésirables sur les fonctionnalités existantes

TEST DE NON-RÉGRESSION

Par exemple, supposons que vous ayez une application de commerce électronique avec une fonctionnalité de "panier".

TEST DE NON-RÉGRESSION

Cette fonctionnalité vous permet d'ajouter des produits à votre panier, de visualiser les produits dans votre panier et de retirer des produits de votre panier.

TEST DE NON-RÉGRESSION

L'équipe de développement décide d'ajouter une nouvelle fonctionnalité qui permet aux utilisateurs de modifier la quantité de chaque produit dans leur panier.

TEST DE NON-RÉGRESSION

Une fois la nouvelle fonctionnalité implémentée, l'équipe de test effectuerait des tests de non-régression pour s'assurer que les fonctionnalités existantes du panier (ajouter des produits, visualiser le panier, retirer des produits) fonctionnent toujours comme prévu.

AMÉLIORATION CONTINUE

FEEDBACK DES UTILISATEURS

COLLECTE ET ANALYSE

- **Questionnaires**
- **Entretiens**
- **Retours d'expérience**

Les avis des **utilisateurs** aident à identifier les problèmes et les axes d'amélioration.

PLANIFICATION DES AMÉLIORATIONS

- **Prioriser** les améliorations en fonction des **retours utilisateurs** et des **objectifs de l'entreprise**
- Intégrer les améliorations dans le futur

ANALYSE DES MÉTRIQUES

- **Erreurs de déploiement**
- **Temps de rétablissement** après les pannes
- **Qualité du code**
- **Temps de réponse** aux incidents