



DevOps

Présenté par :
Elies Jebri

CI/CD

Continuous Integration / Delivery - Deployment

Elies Jebri

■ **Titre:** Consultant expert en OSS et infrastructures IT.

■ **Distinctions & expérience:**

- Expert depuis 2000 auprès d'un grand nombre de compagnies locales et internationales.

■ **Formation:**

- Agrégé en informatique industrielle,
- Certifié RHCE, LPIC-1,2,301/303/304, NCLA, MCITP, CCNA, CEH, TenStep, AWS Practitioner, AWS SysOps, AWS Architect, VMware Foundations, VMware Data Center Professional, VMware Certified Instructor, Huawei Cloud Services, LPI-DevOps, DevOps Generalist, Cisco ENCOR, DCCOR, AZ-900, VMCE

■ **Contact:**

elies.jebri@gmail.com / ejebri@clevery.com



Introduction

- Ce module est un aperçu des concepts d'intégration continue et de livraison continue.
- Nous couvrirons les concepts de base de DevOps dans ce module.
- En ce qui concerne les pré-requis, vous devez être familier avec les bases du cycle de vie du développement logiciel





Qu'est-ce que le DevOps ?

- Combinant développement (Dev) et opérations (Ops), DevOps est l'union des personnes, des processus et des technologies destinés à fournir continuellement de la valeur aux clients.
- DevOps permet la coordination et la collaboration des rôles autrefois cloisonnés (développement, opérations informatiques, ingénierie qualité et sécurité) pour créer des produits plus performants et plus fiables.
- En adoptant une culture DevOps ainsi que des pratiques et outils DevOps, les équipes peuvent mieux répondre aux besoins des clients, accroître la confiance suscitée par les applications qu'elles développent, et atteindre plus rapidement les objectifs de leur entreprise.

Avantages de DevOps

- Les équipes qui adoptent la culture, les pratiques et les outils DevOps sont plus performantes et créent de meilleurs produits plus rapidement, à la grande satisfaction de leurs clients.
- Cette collaboration et cette productivité améliorées permettent également d'atteindre des d'objectifs commerciaux tels que les suivants :



Raccourcir le délai de commercialisation



S'adapter au marché et à la concurrence



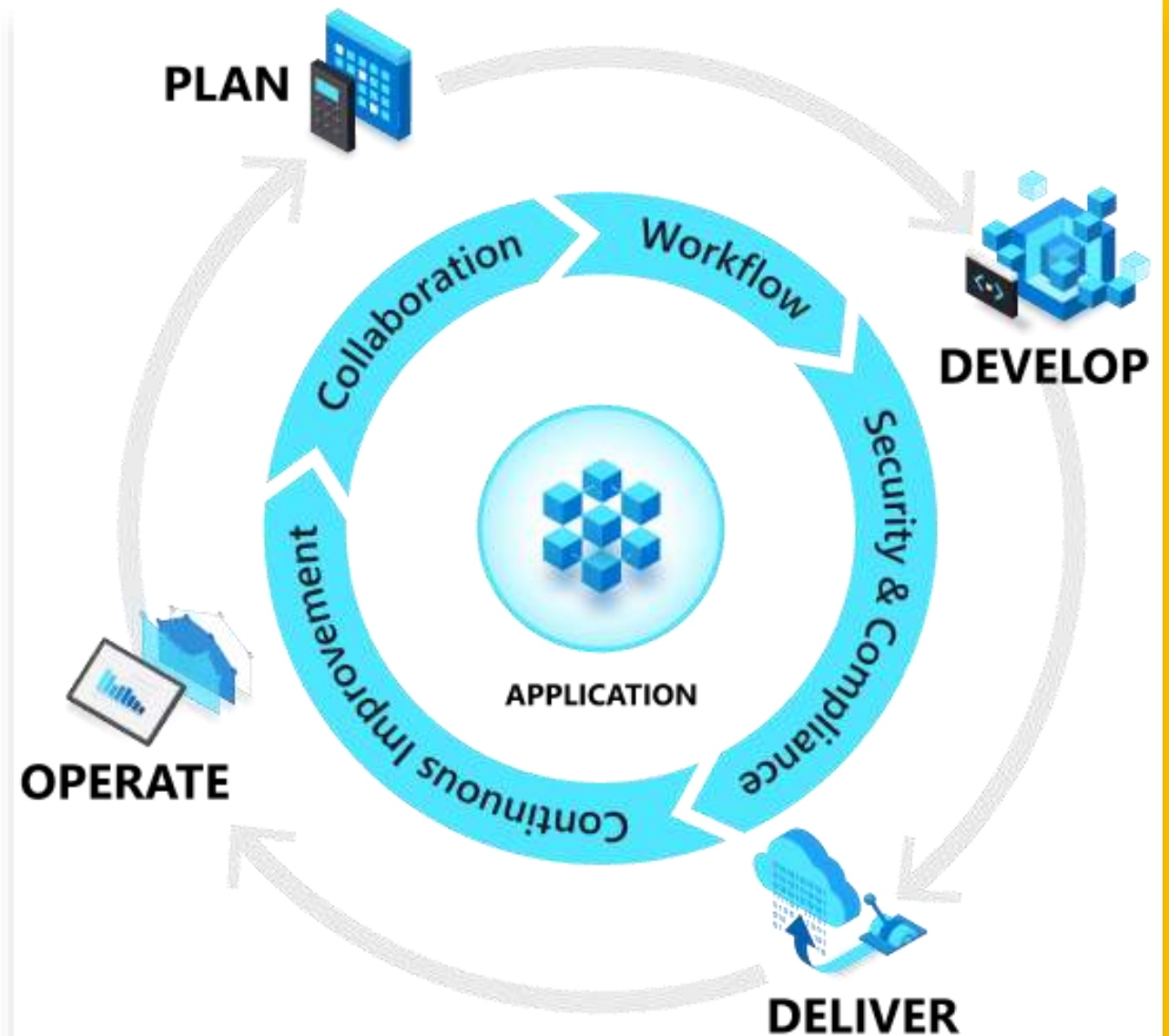
Maintien de la stabilité et de la fiabilité du système



Améliorer le temps moyen de récupération

DevOps et le cycle de vie des applications

- DevOps influence le cycle de vie des applications tout au long de leurs phases de planification, de développement, de livraison et d'exploitation.
- Ces phases reposent les unes sur les autres et ne sont pas spécifiques à un rôle.
- Au sein d'une culture DevOps, chaque rôle est, dans une certaine mesure, impliqué dans les différentes phases.






Plan

- Au cours de la phase de planification, les équipes DevOps imaginent, définissent et décrivent les fonctionnalités des applications et des systèmes qu'elles créent.
- Elles suivent leur progression à des niveaux de granularité faibles et élevés, des tâches portant sur un seul produit aux tâches couvrant des portefeuilles de plusieurs produits.
- La création de backlogs, le suivi des bogues, la gestion du développement logiciel agile avec Scrum, l'utilisation de tableaux Kanban et la visualisation des progrès réalisés grâce aux tableaux de bord illustrent les moyens dont disposent les équipes DevOps pour planifier avec agilité et visibilité.



Développement

- La phase de développement comprend tous les aspects du codage (écriture, test, révision et intégration du code par les membres de l'équipe), ainsi que la génération de ce code dans des artefacts de build pouvant être déployés dans divers environnements.
- Les équipes DevOps entendent innover rapidement, sans sacrifier la qualité, la stabilité et la productivité.
- Pour ce faire, elles utilisent des outils hautement productifs, automatisent des étapes simples et manuelles, et effectuent des itérations par petits incréments moyennant des tests automatisés et une intégration continue.



Fournir (Livrer)

- La livraison consiste à déployer des applications dans des environnements de production de manière cohérente et fiable.
- La phase de livraison englobe également le déploiement et la configuration de l'infrastructure de base entièrement régie qui constitue ces environnements.
- Lors de la phase de livraison, les équipes définissent un processus de gestion des mises en production ponctué d'étapes d'approbation manuelle claires.
- Elles définissent également des portes automatisées que franchissent les applications entre les étapes jusqu'à leur mise à la disposition des clients.
- L'automatisation de ces processus les rend évolutifs, reproductibles, contrôlés.
- Ainsi, les équipes qui utilisent DevOps peuvent procéder à des livraisons plus fréquentes avec facilité, confiance et tranquillité d'esprit.



Opérer

- La phase d'exploitation implique la maintenance, la supervision et le dépannage des applications dans les environnements de production.
- En adoptant les pratiques DevOps, les équipes veillent à assurer la fiabilité, la haute disponibilité du système, et visent à éliminer les temps d'arrêt tout en renforçant la sécurité et la gouvernance.
- Les équipes DevOps entendent identifier les problèmes avant qu'ils n'affectent l'expérience client, et à les atténuer rapidement lorsqu'ils surviennent.
- Maintenir cette vigilance nécessite une riche télémétrie, des alertes exploitables et une visibilité totale sur les applications et le système sous-jacent.



Culture DevOps

- Si l'adoption de pratiques DevOps automatise et optimise les processus grâce à la technologie, elle repose essentiellement sur la culture de l'organisation, et les personnes qui en font partie.
- Le défi consistant à entretenir une culture DevOps implique de profonds changements dans la manière dont chacun travaille et collabore.
- Cela étant, lorsque les entreprises s'engagent sur la voie d'une culture DevOps, elles peuvent créer un environnement propice au développement d'équipes hautement performantes.



Cycles de mise en production plus courts

- Les équipes DevOps restent agiles en publiant des logiciels dans des cycles courts.
- Les cycles de mise en production plus courts facilitent la planification et la gestion des risques car la progression est incrémentielle, ce qui réduit également tout impact sur la stabilité du système.
- En outre, le raccourcissement du cycle de mise en production permet aux organisations de s'adapter et de réagir aux besoins évolutifs des clients, de même qu'à la pression de la concurrence.



Apprentissage continu

- Les équipes DevOps hautement performantes adoptent un état d'esprit de croissance.
- Elles effectuent un Fail-fast et intègrent ce qu'elles apprennent dans leurs processus, s'améliorant continuellement, renforçant la satisfaction des clients, accélérant leur capacité à innover et à s'adapter au marché.
- DevOps est un cheminement qui fait la part belle à la croissance.

Pratiques DevOps

En plus d'établir une culture DevOps, les équipes donnent vie à DevOps en implémentant diverses pratiques tout au long du cycle de vie des applications. Certaines de ces pratiques permettent d'accélérer, d'automatiser et d'améliorer une phase spécifique. D'autres couvrent plusieurs phases, aidant les équipes à créer des processus homogènes qui contribuent à renforcer la productivité.



Intégration continues (CI)

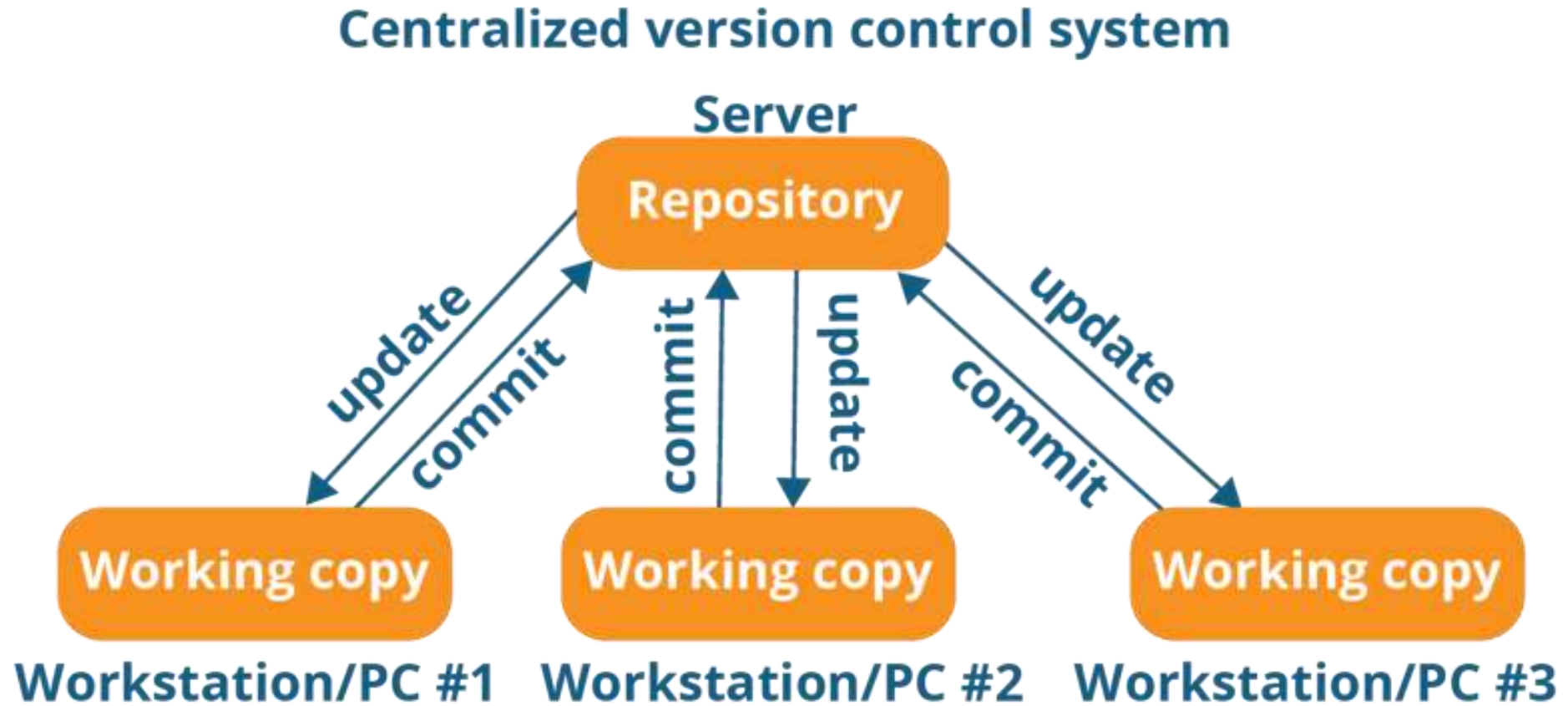
- L'intégration continue est une pratique de développement logiciel dans laquelle les développeurs fusionnent fréquemment les modifications de code dans la branche de code principale.
- L'intégration continue utilise des tests automatisés, qui s'exécutent chaque fois qu'un nouveau code est validé afin d'assurer la stabilité du code de la branche principale.



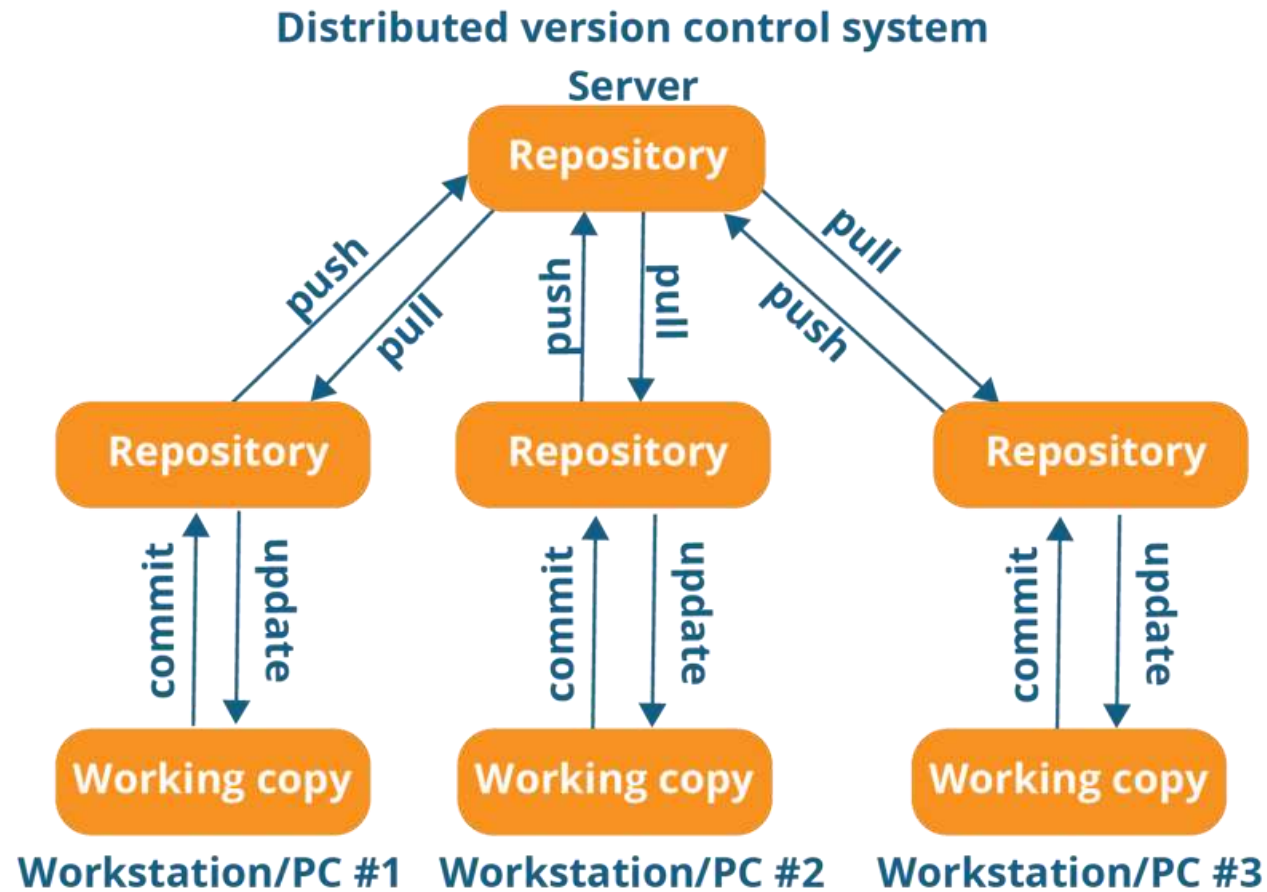
Gestion de version

- Consiste à gérer le code dans les versions, à suivre les révisions et l'historique des modifications afin de faciliter l'examen et la récupération du code.
- Implémentée à l'aide de systèmes de gestion de version tels que **Git**, qui permettent à plusieurs développeurs de collaborer sur la création du code.
- Propose un processus pour fusionner les modifications de code intervenant dans les mêmes fichiers, gérer les conflits et restaurer les modifications apportées aux états antérieurs.
- Une pratique DevOps fondamentale.
- La gestion de version est également indissociable d'autres pratiques telles que l'intégration continue et l'infrastructure en tant que code.

Centralized Version Control Systems



Distributed Version Control Systems





Livraison continues (CD)

- La livraison continue correspond au déploiement fréquent et automatisé de nouvelles versions d'application dans un environnement de production.
- L'automatisation des étapes requises à des fins de déploiement permet aux équipes de limiter les problèmes susceptibles de survenir lors du déploiement et de procéder à des mises à jour plus fréquentes.

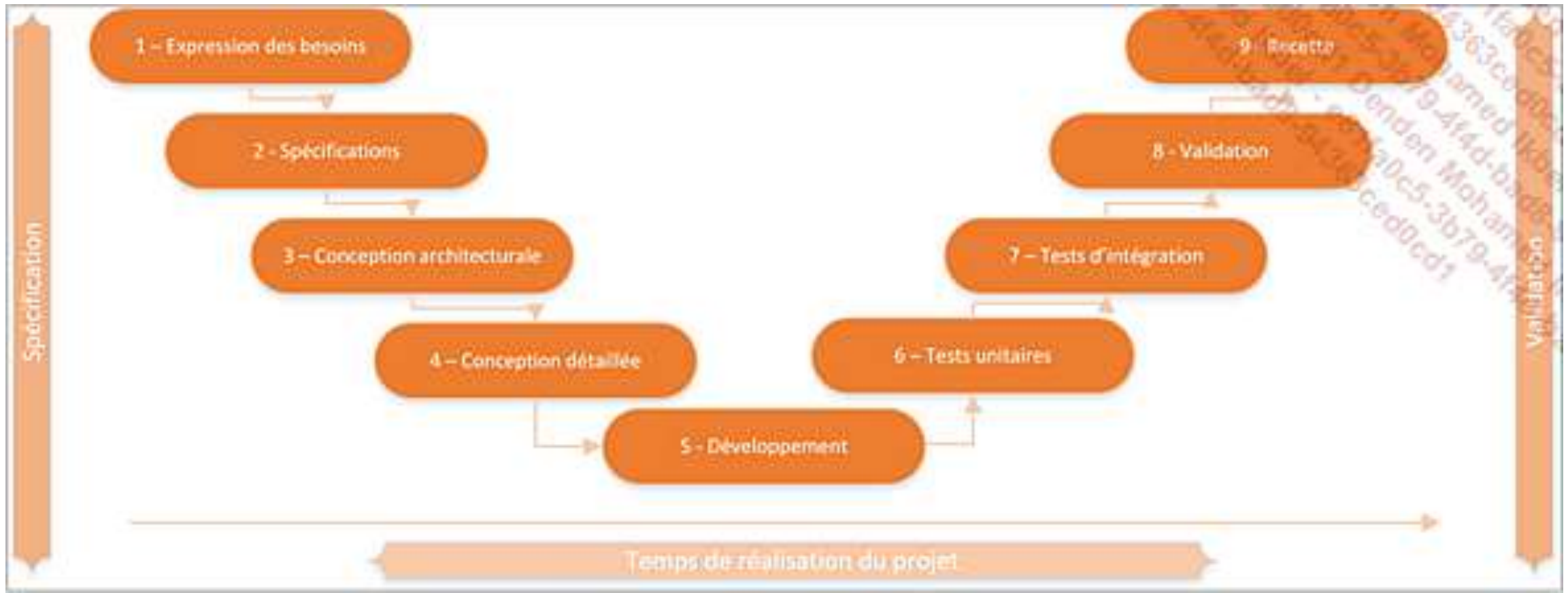


Intégration et livraison continues (CI/CD)

- Ces deux pratiques ouvrent la voie au processus CI/CD, qui comprend l'automatisation complète de toutes les étapes entre la validation du code et le déploiement de production.
- CI/CD permet aux équipes de se concentrer sur la création du code et élimine la surcharge et les erreurs humaines potentielles liées aux simples étapes manuelles.
- CI/CD permet de déployer du code de façon plus rapide et moins risquée.
- Les déploiements interviennent alors plus fréquemment et plus progressivement, ce qui renforce l'agilité et la productivité des équipes, de même que leur confiance quant à leur code d'exécution.

L'évolution de la gestion de projet en informatique

- Evolution #1 : La taille des projets
- Evolution #2 : Les méthodologies
- Evolution #3 : les cycles projets
- Evolution #4 : la taille des équipes
- Evolution #5 : la complexité du code n'est plus la même
- Evolution #6 : La communication et l'organisation avec le client

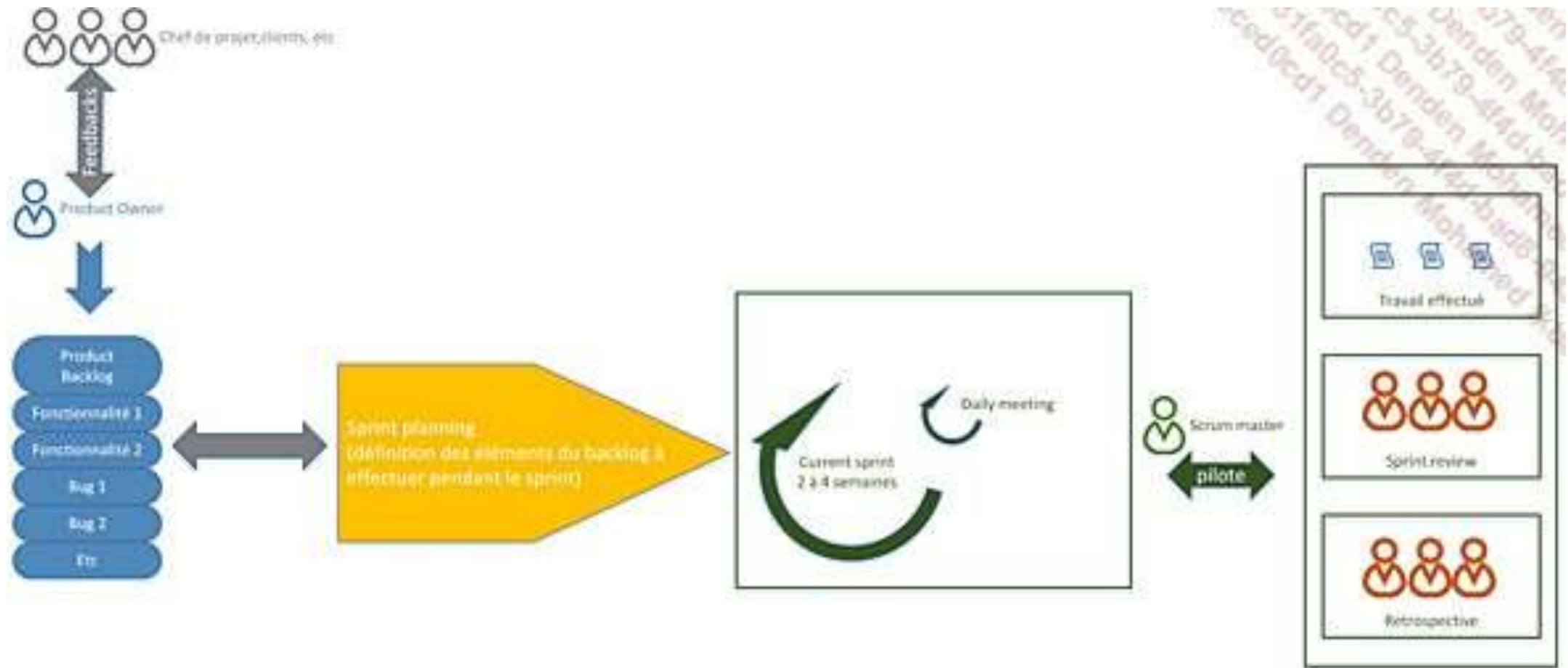




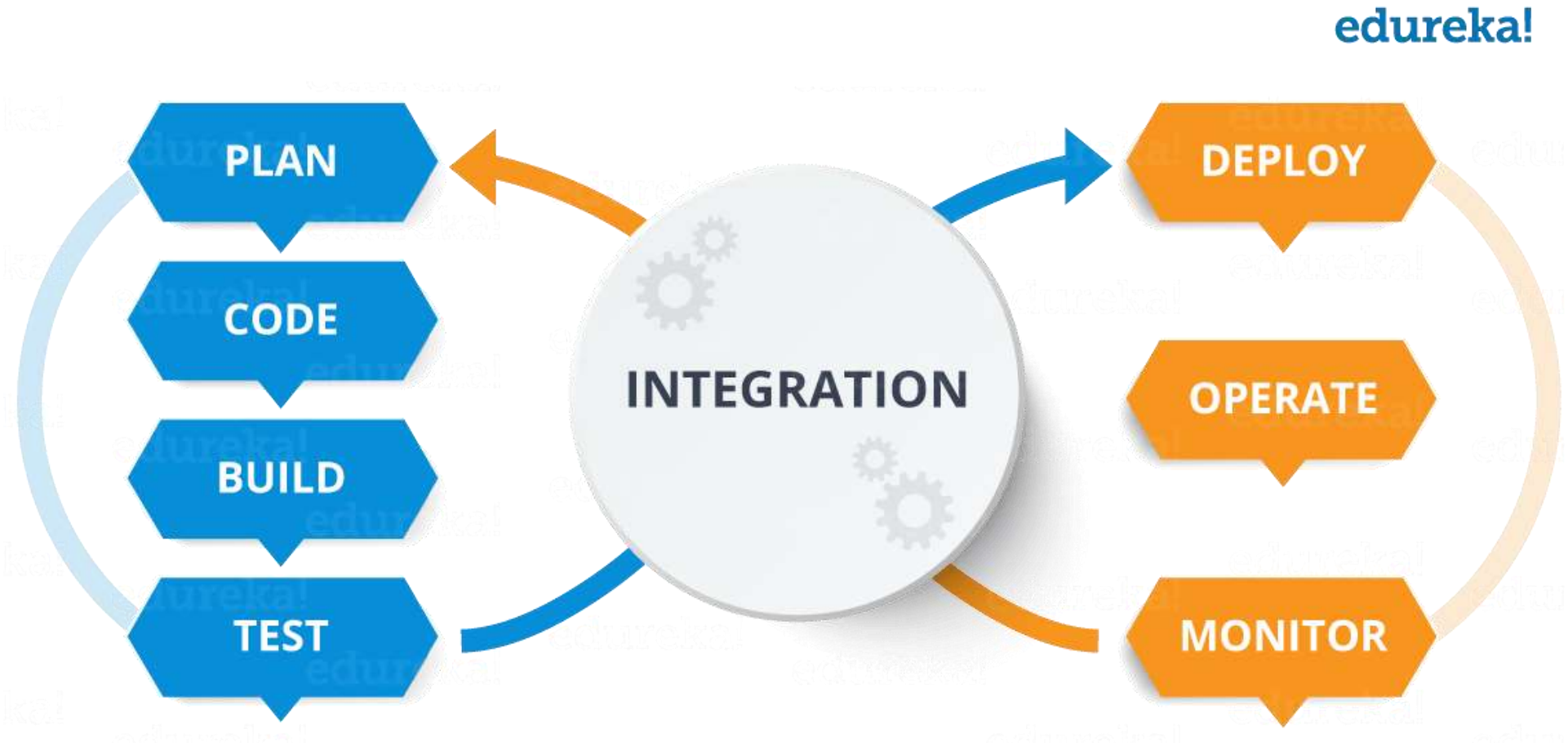
Agile software development

- Agile est une approche de développement logiciel qui met l'accent sur la collaboration d'équipe, les commentaires des clients et des utilisateurs, ainsi que la capacité à s'adapter aux changements moyennant des cycles de mise en production courts.
- Les équipes qui utilisent Agile apportent des modifications et des améliorations constantes aux clients, recueillent leurs commentaires, puis en tirent des enseignements et s'adaptent en fonction des attentes et des besoins de ces clients.
- Agile diffère sensiblement des approches traditionnelles telles que l'approche en cascade, qui se caractérise par de longs cycles de mise en production définis par des phases séquentielles.
- Kanban et Scrum sont deux approches très populaires associées à Agile.

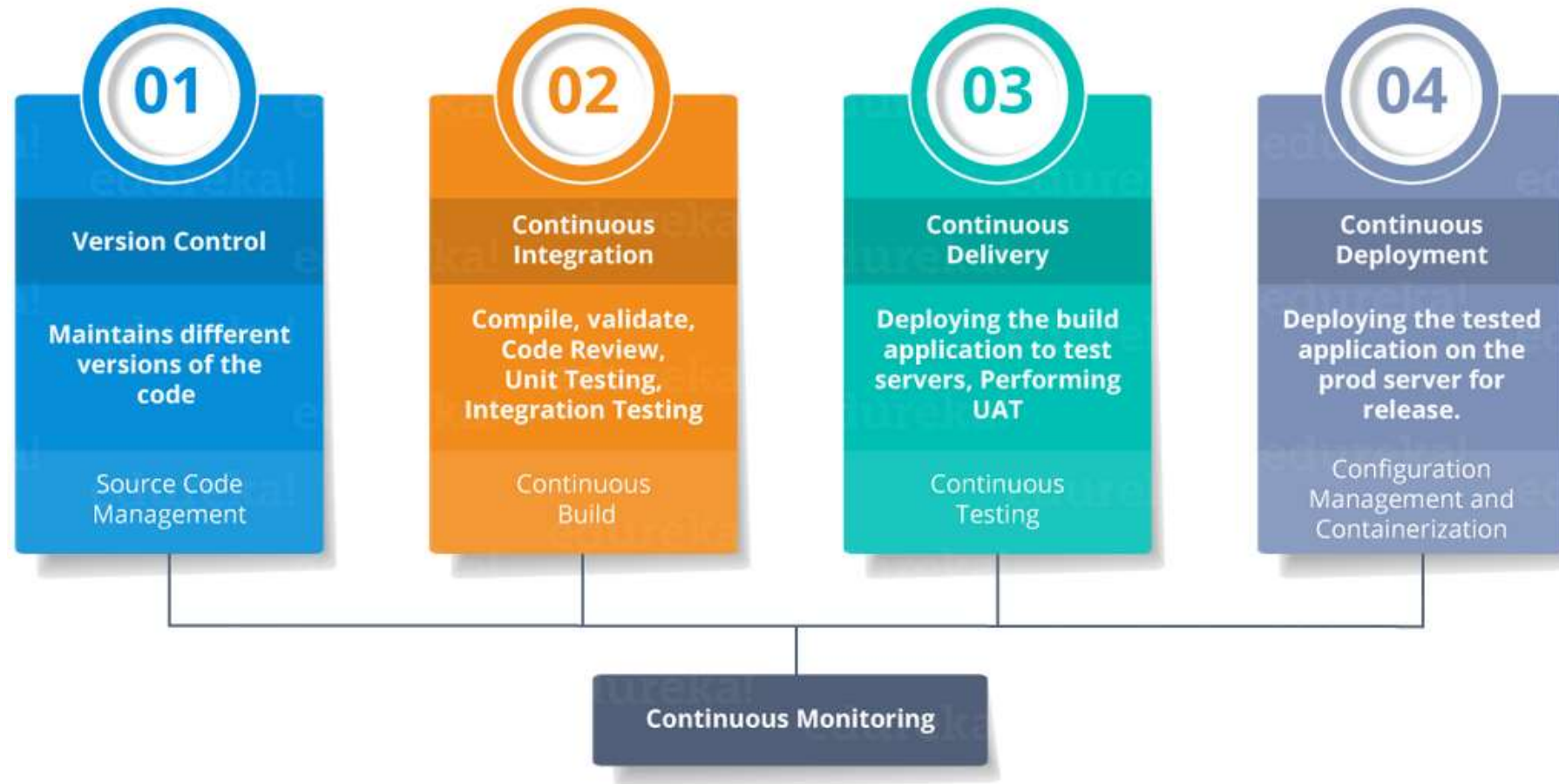
Vers les méthodes Agiles



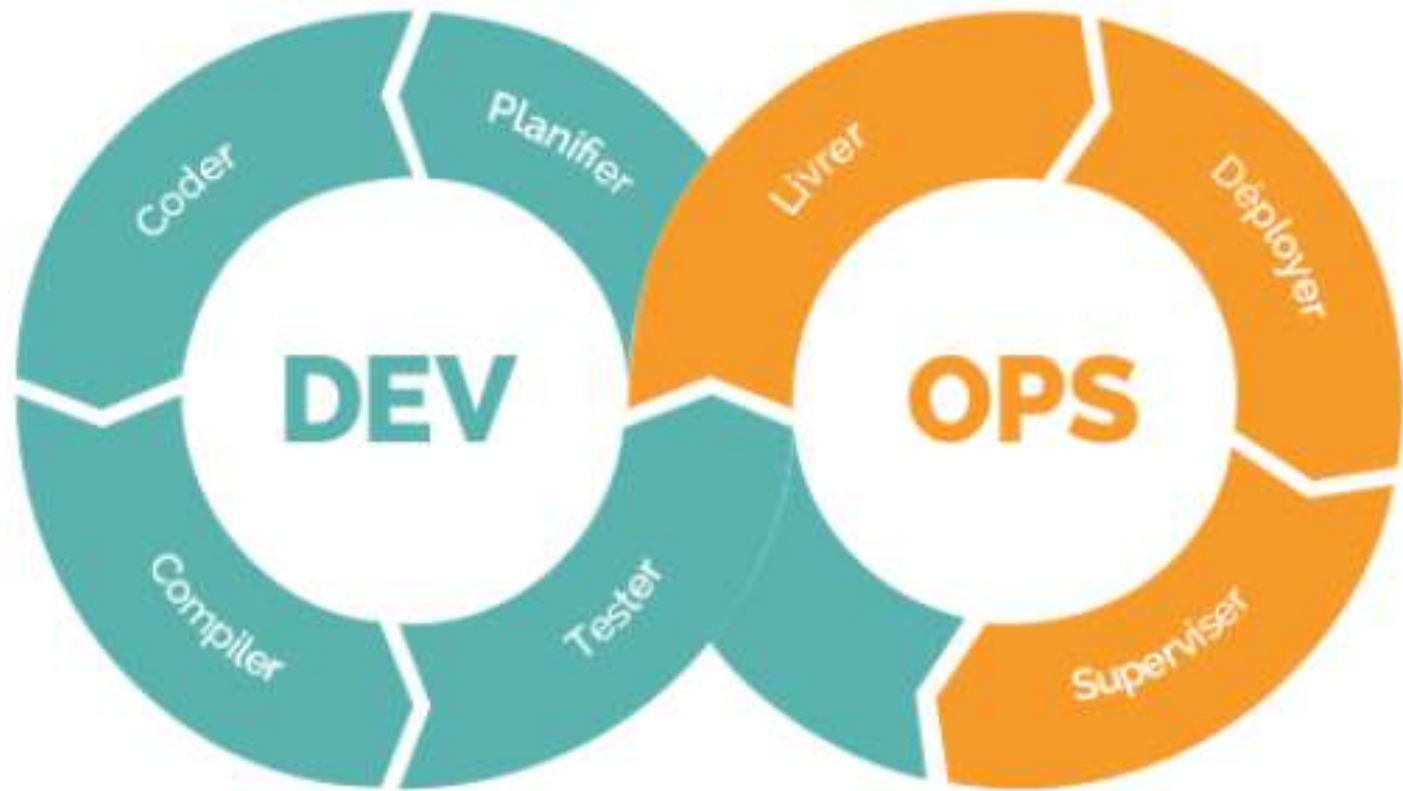
Comprendre DevOps



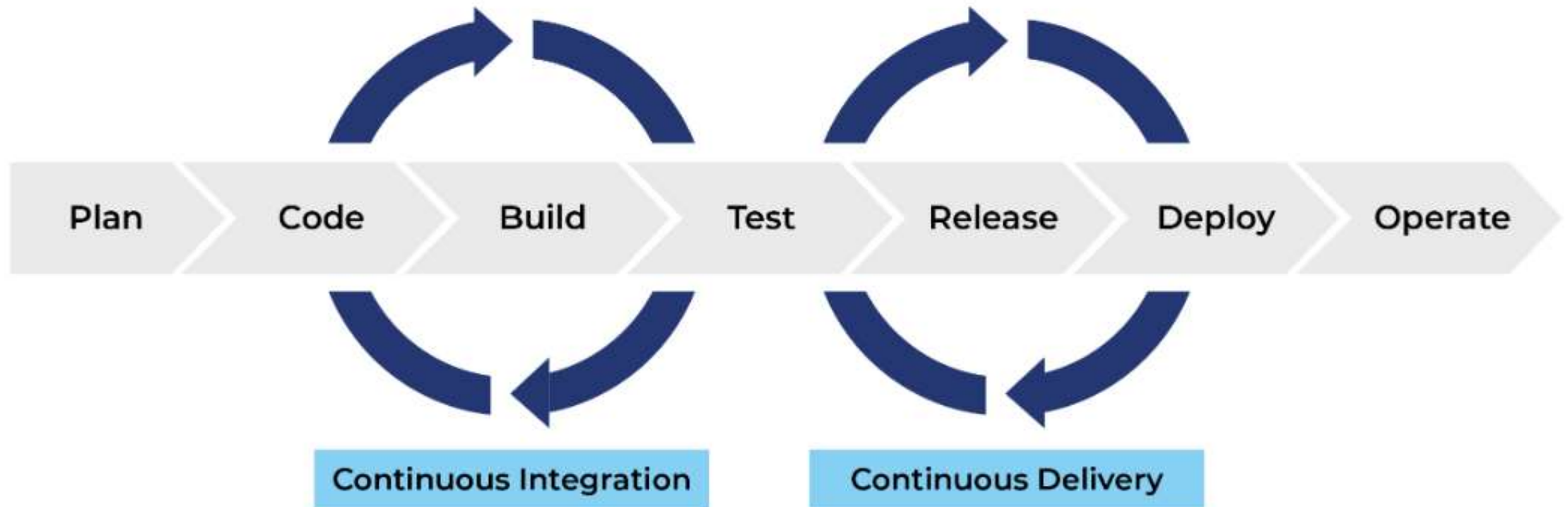
Activités de DevOps



Cycle DevOps



DevOps



AGILE VS DEVOPS

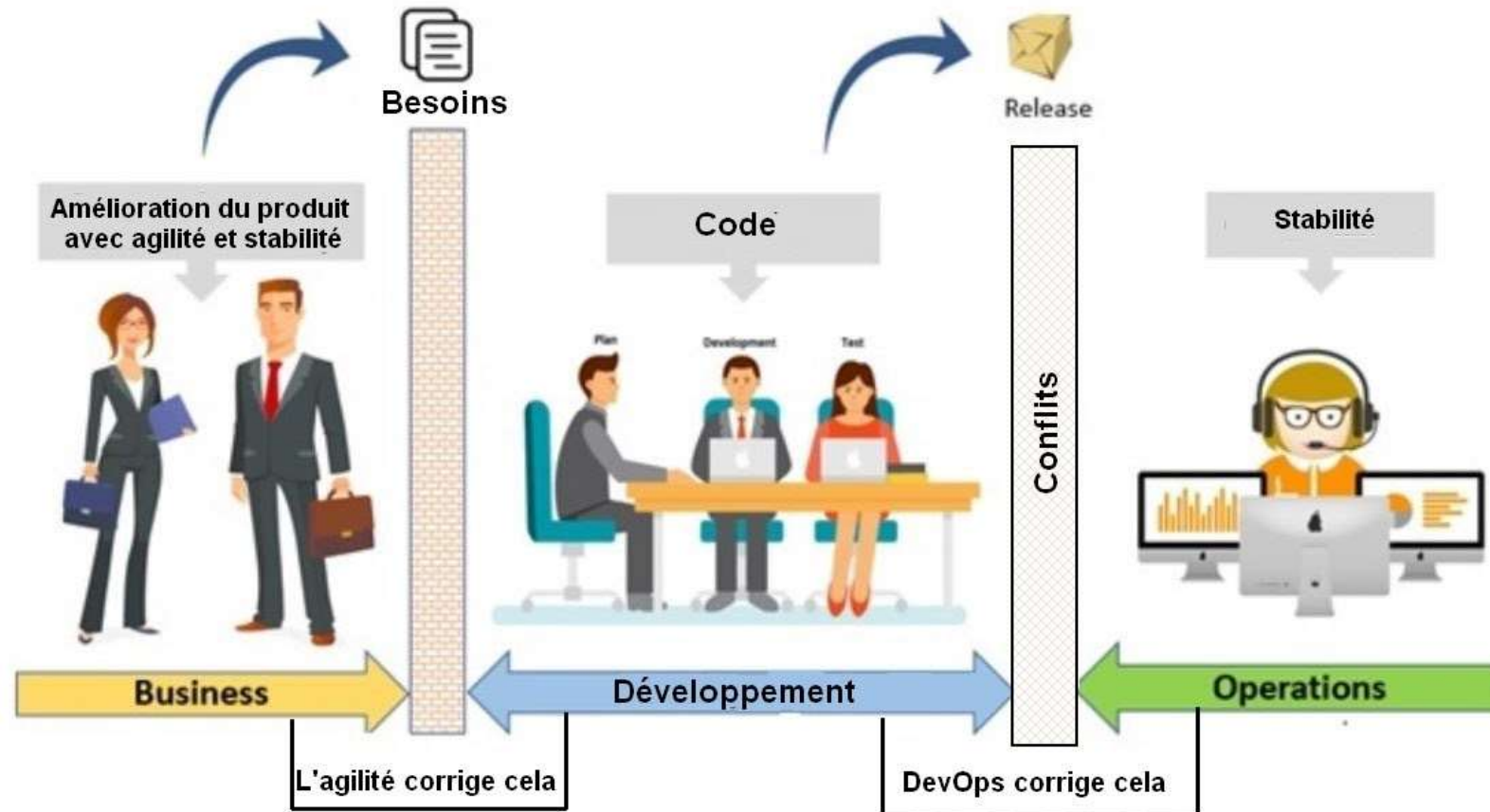
DEVOPS

AGILE



Agile
vs
DevOps

Agile vs DevOps





Cycle en V



Agilité



DevOps



Infrastructure en tant que code

- Définit les ressources système et les topologies de manière descriptive, ce qui permet aux équipes de gérer ces ressources comme du code.
- Ces définitions peuvent également être stockées et versionnées dans des systèmes de gestion de version permettant leur révision et leur restauration, là encore comme du code.
- Aide les équipes à déployer des ressources système de manière fiable, reproductible et contrôlée.
- De plus, l'infrastructure en tant que code permet d'automatiser le déploiement et réduit le risque d'erreur humaine, notamment au sein d'environnements aussi vastes que complexes.
- Fiable et reproductible, cette solution de déploiement d'environnement offre aux équipes la possibilité de gérer des environnements de développement et de test identiques aux environnements de production.
- La duplication des environnements dans différents centres de données et plateformes cloud gagne également en simplicité et en efficacité.

Configuration Management

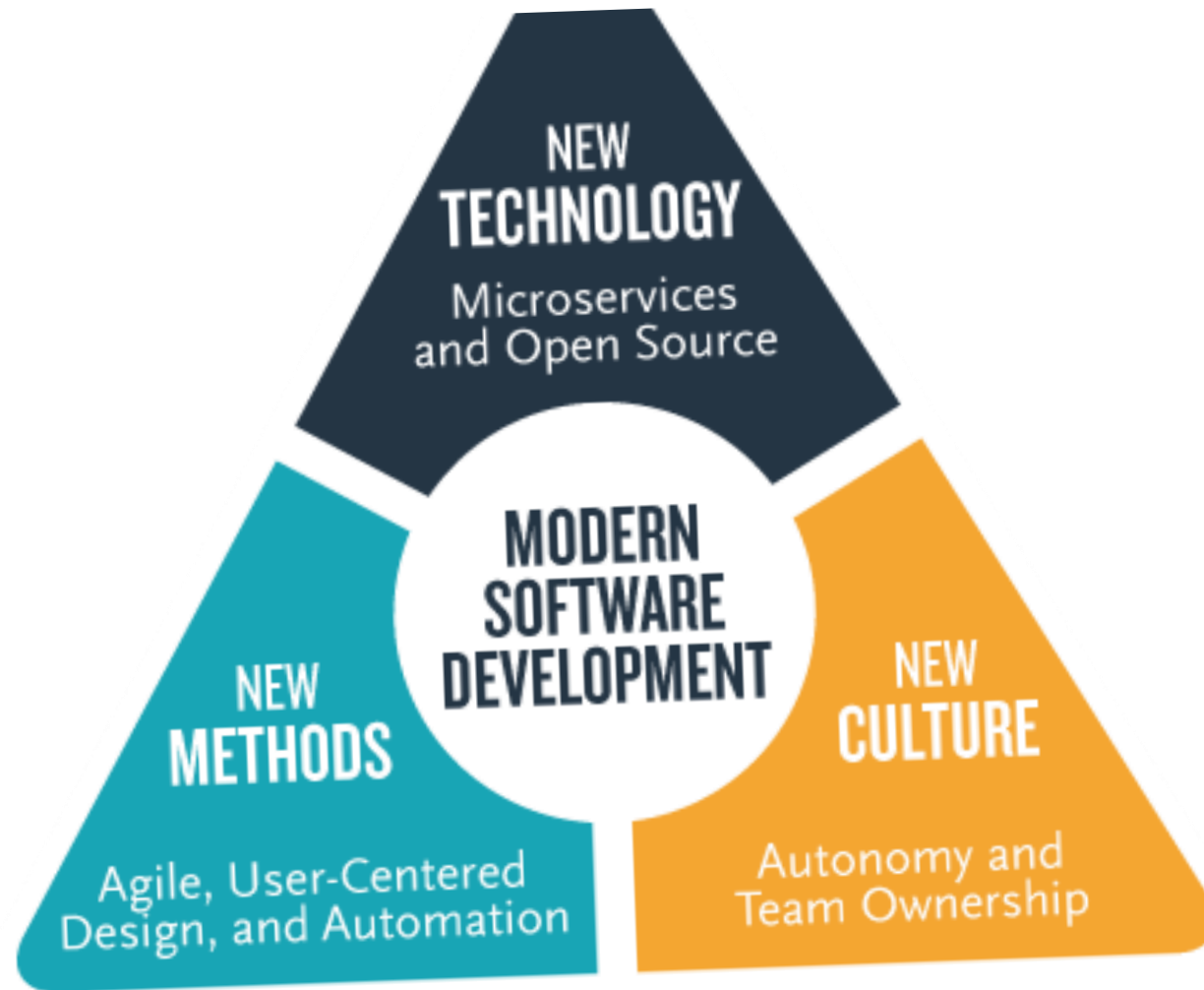
- La gestion de la configuration fait référence à la gestion de l'état des ressources d'un système, ce qui englobe les serveurs, machines virtuelles et bases de données.
- Moyennant des outils de gestion de la configuration, les équipes peuvent déployer les modifications de façon systématique et contrôlée, réduisant ainsi les risques de modification de la configuration système.
- Les équipes utilisent des outils de gestion de la configuration pour suivre l'état du système et éviter toute dérive de configuration, à savoir toute dérive de configuration d'une ressource système par rapport à son état souhaité au fil du temps.
- Combinées à l'infrastructure en tant que code, la définition et la configuration système sont faciles à modéliser et à automatiser, ce qui aide les équipes à exploiter des environnements complexes à grande échelle.





Supervision continue

- Une supervision continue offre une visibilité complète en temps réel sur les performances et l'intégrité de toute la pile d'applications, de l'infrastructure sous-jacente exécutant les applications aux composants logiciels de niveau supérieur.
- Cette visibilité comprend la collecte de données de télémétrie et de métadonnées, ainsi que la définition d'alertes correspondant à des conditions prédéfinies qui doivent susciter l'attention d'un opérateur.
- La télémétrie englobe les données d'événement et les journaux issus de différentes parties du système et stockés à un emplacement où ils peuvent être analysés et interrogés.
- Les équipes DevOps hautement performantes veillent à définir des alertes exploitables et explicites, et à collecter des données de télémétrie enrichies afin de mieux les exploiter.
- Fortes de ces informations, les équipes peuvent résoudre les problèmes en temps réel et améliorer les applications lors des cycles de développement ultérieurs.

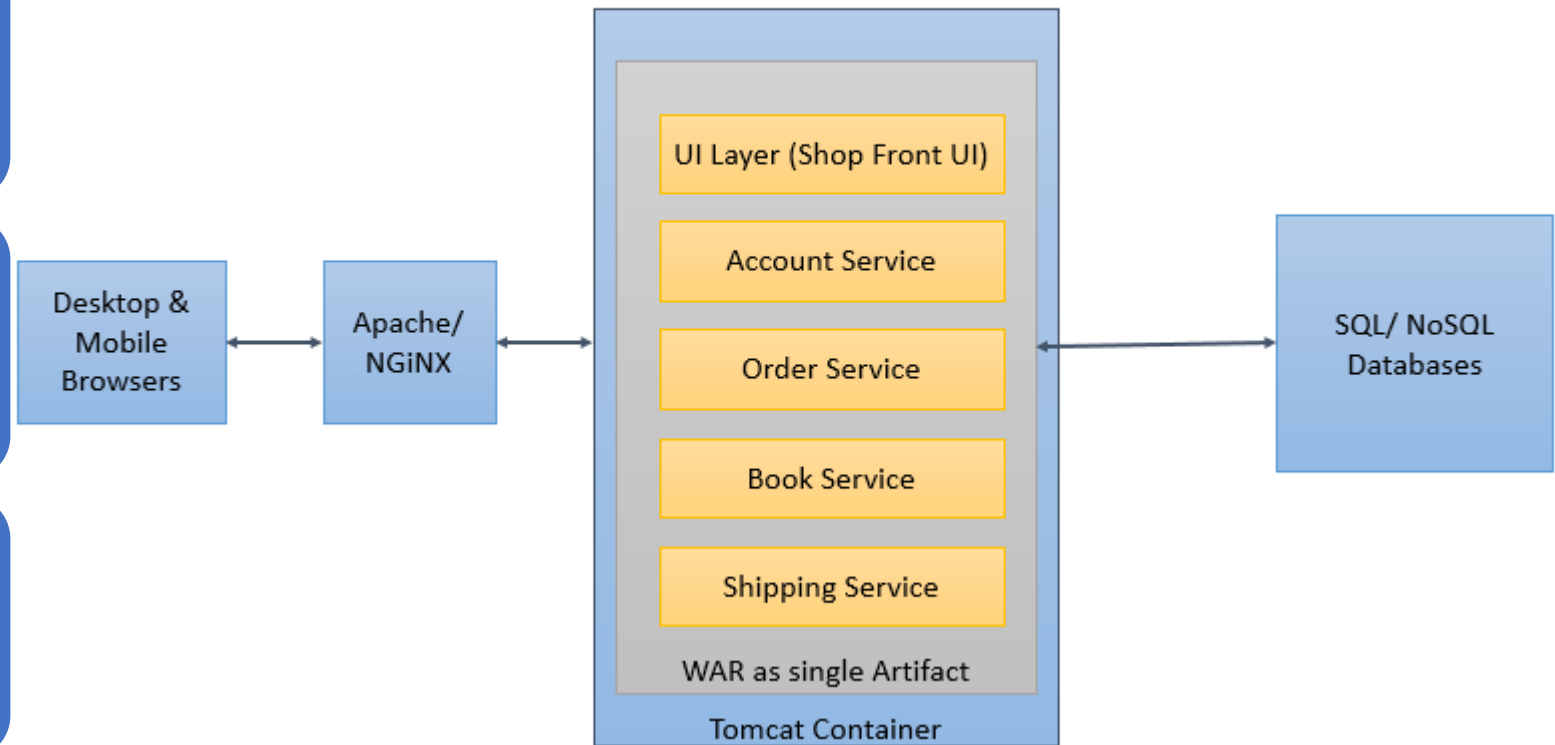


Application monolithique : rappel

Un gros code contenant toutes les fonctionnalités et les différentes couches logicielles

Une seule grosse compilation et un seul livrable (un gros fichier WAR)

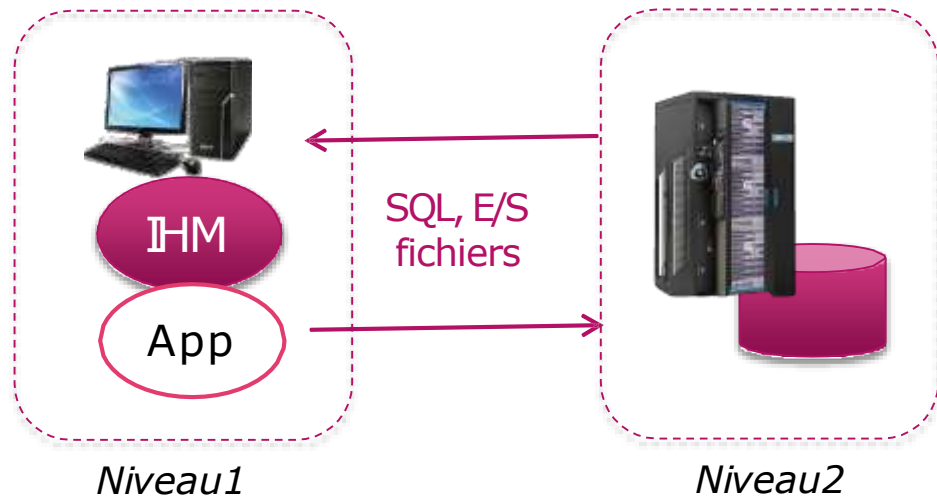
Une seule pile logicielle (Linux, JVM, Tomcat et bibliothèques tierces)



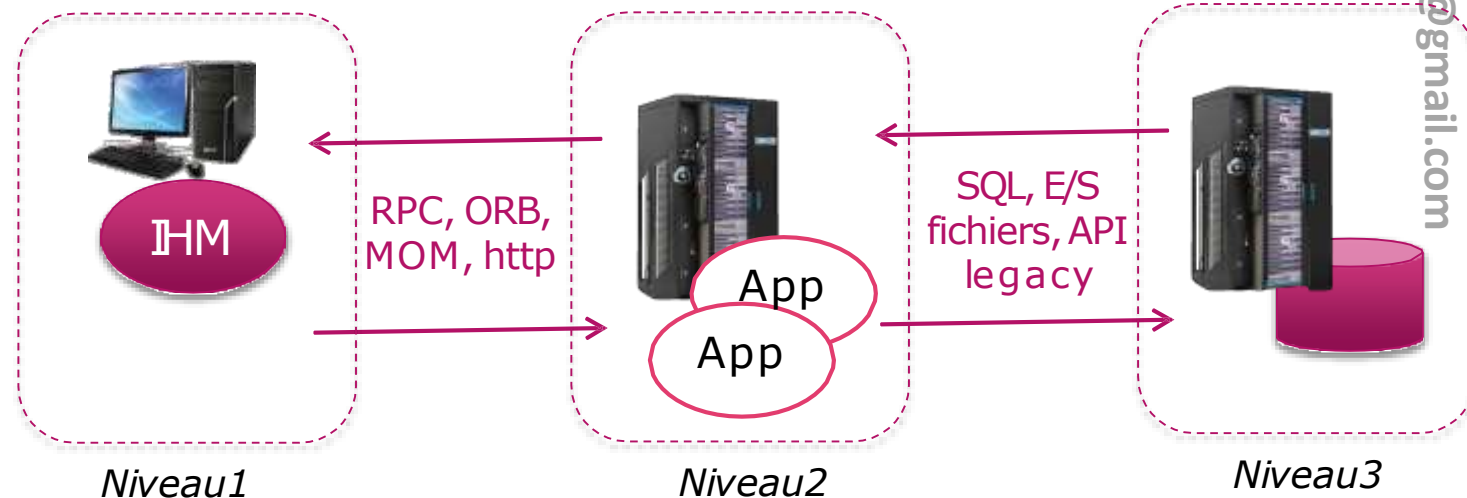
Architectures en Tier

- Principe : séparation des niveaux fonctionnels
 - IHM : interface Homme-Machine
 - Application métier
 - Gestion des données
- Le lien entre les niveaux est défini et limité à des interfaces
- Les applications peuvent être déployées et administrées de manière indépendante des IHM
- Placement des serveurs logiciels sur un ou plusieurs serveurs physiques
- Nombreuses variantes architecturales possibles

De 2 à 3 Niveaux

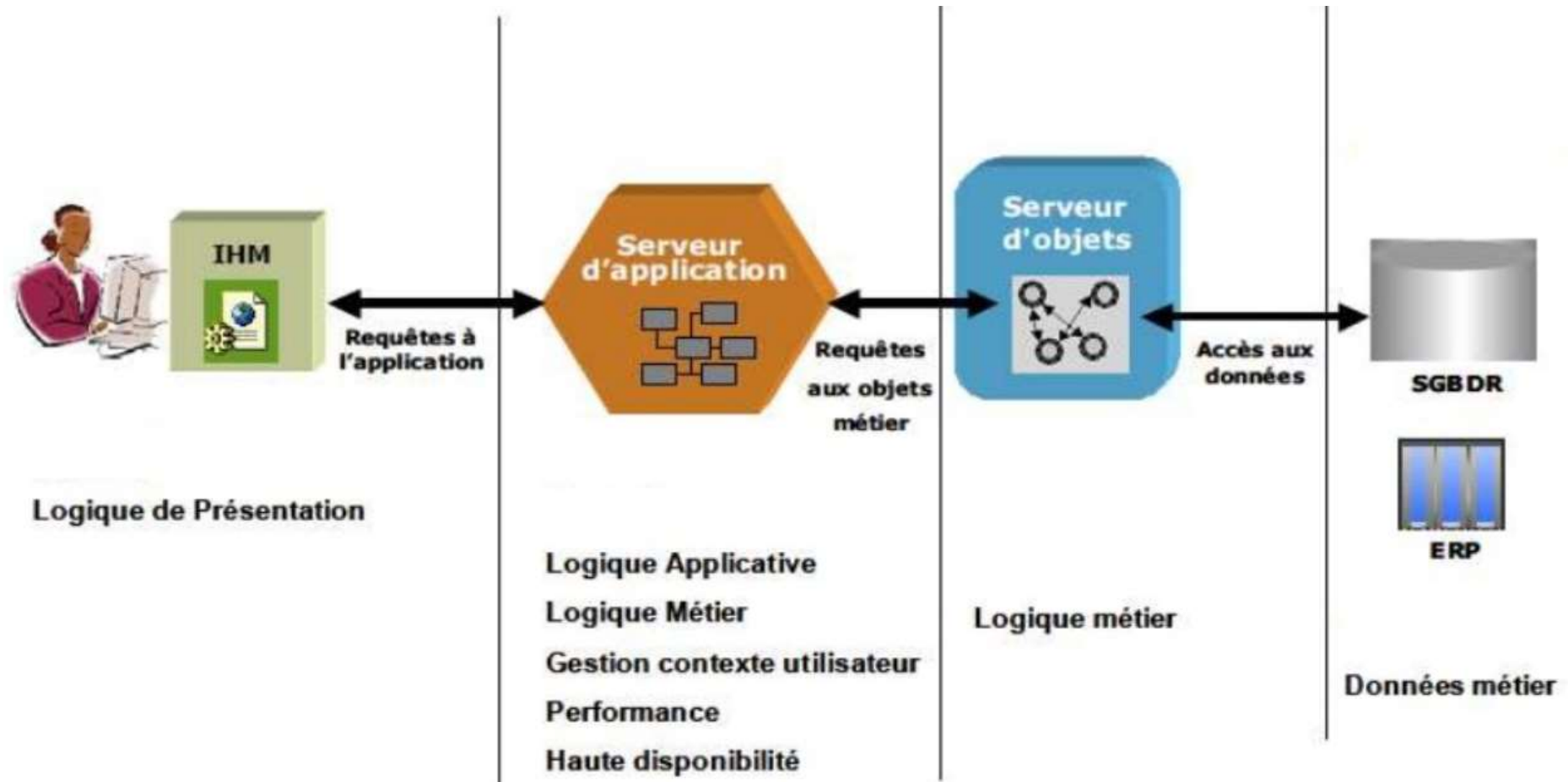


Client-Serveur à deux Niveaux

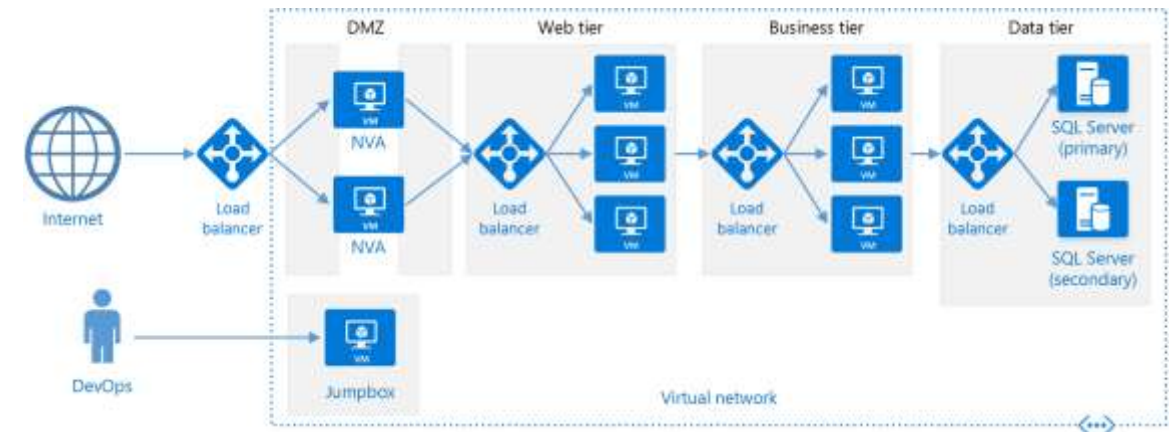
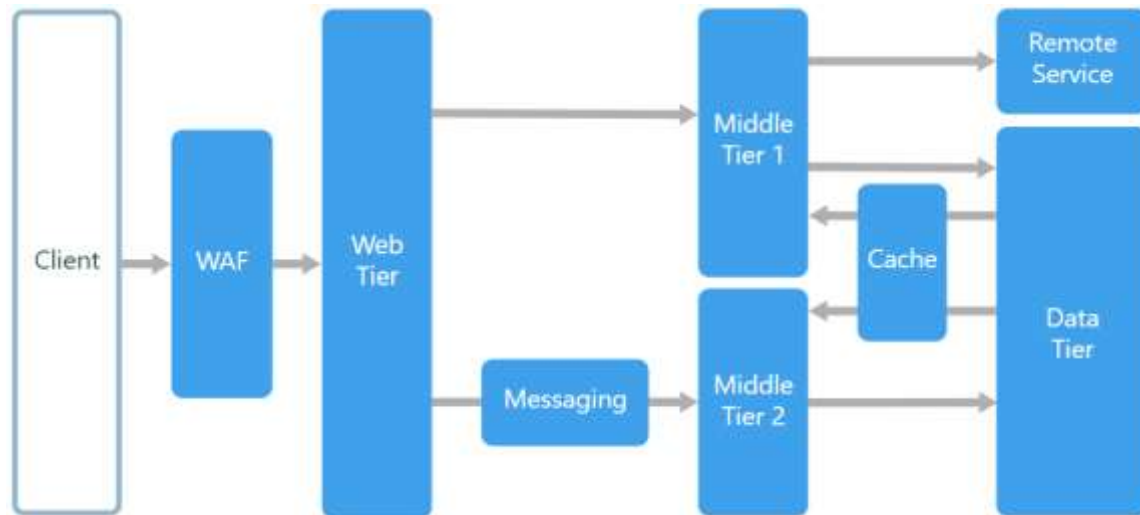


Client-Serveur à trois Niveaux

Architecture Multi-Tier



Exemples d'architectures Multi-Tier

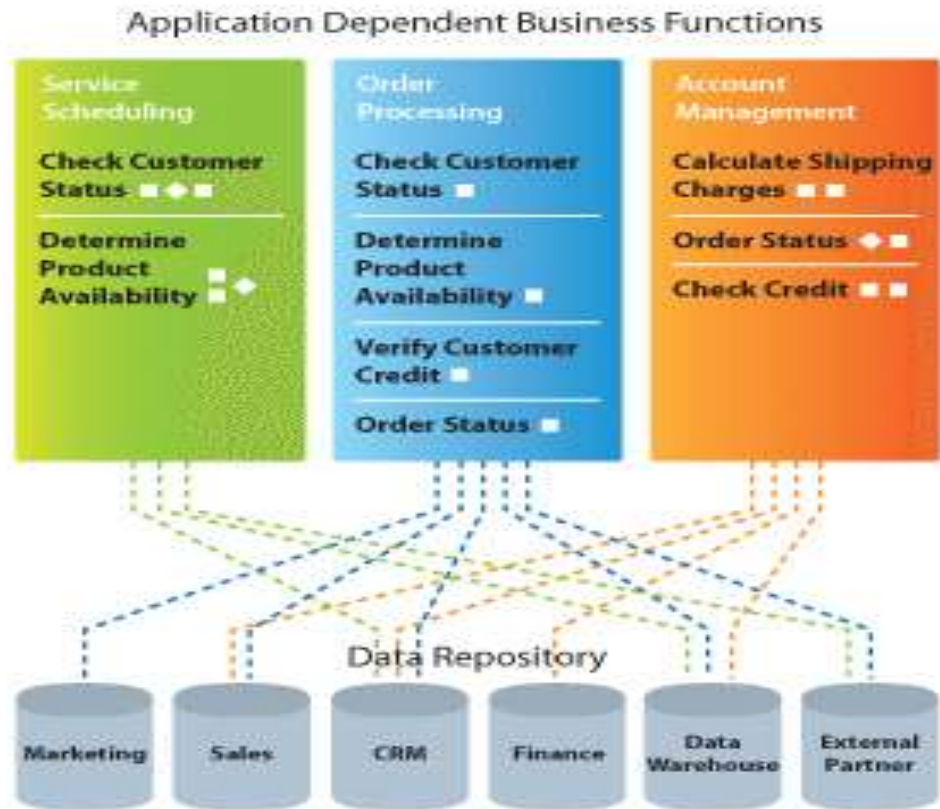


Les Architectures Orientées Services SOA

- Tout Devient «Service»...
- Service :
 - Fonctionnalité réutilisable dont le comportement est défini de façon contractuelle
- 3 Acteurs
 - **Consommateur de service** décrit le service à consommer
 - **Fournisseur de services** découvert en temps d'exécution à l'aide d'un intermédiaire (Registre de service)
 - **Registre de service** contient l'ensemble des descripteurs de services et les références vers les fournisseurs de services

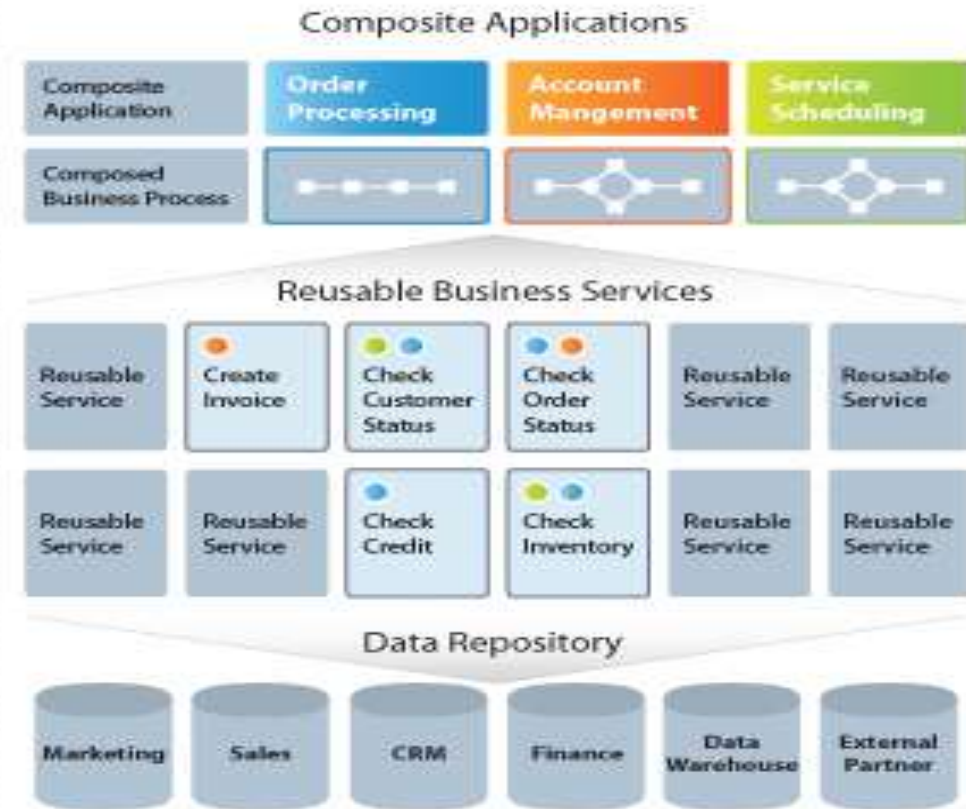
Before SOA

Closed - Monolithic - Brittle




After SOA

Shared services - Collaborative - Interoperable - Integrated



En résumé



Architecture microservices

Architecture

Naissance des MSA

Il faut donc que les entreprises conçoivent leurs applications dès le départ à partir d'architectures parfaitement adaptées au **cloud**.

L'architecture **Microservices** apporte une réponse concrète à ces préoccupations et permet d'obtenir, des applications dites "**Cloud-native**".

Que sont les microservices?

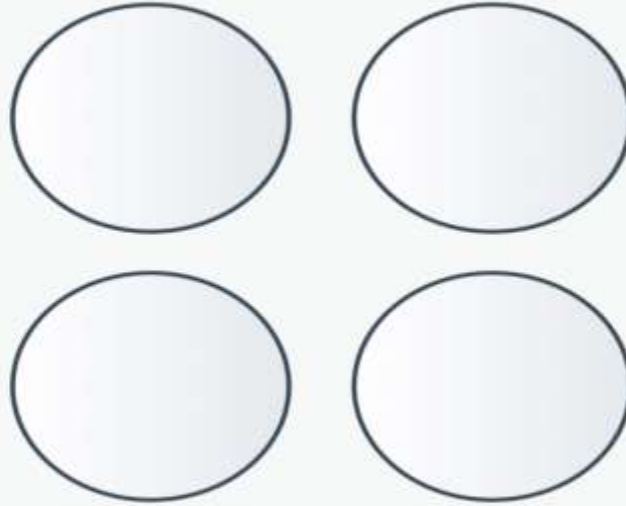
- Les microservices sont un modèle d'architecture **orienté service** dans lequel les applications sont construites comme un ensemble de diverses unités de service indépendantes les plus petites.
- Il s'agit d'une approche d'ingénierie logicielle qui se concentre sur la décomposition d'une application en modules à **fonction unique** avec des interfaces bien définies et **faiblement couplés**.
- Ces modules peuvent être déployés et exploités indépendamment par de petites équipes qui possèdent l'intégralité du cycle de vie du service.

Monolithic vs. SOA vs. Microservices



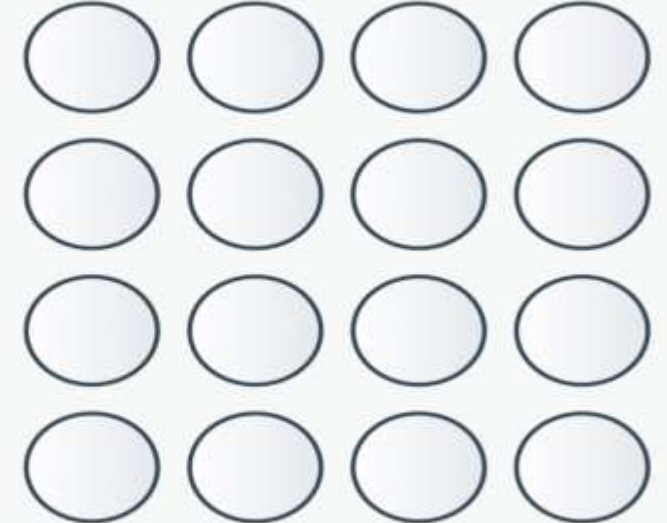
Monolithic

Single Unit



SOA

Coarse-grained

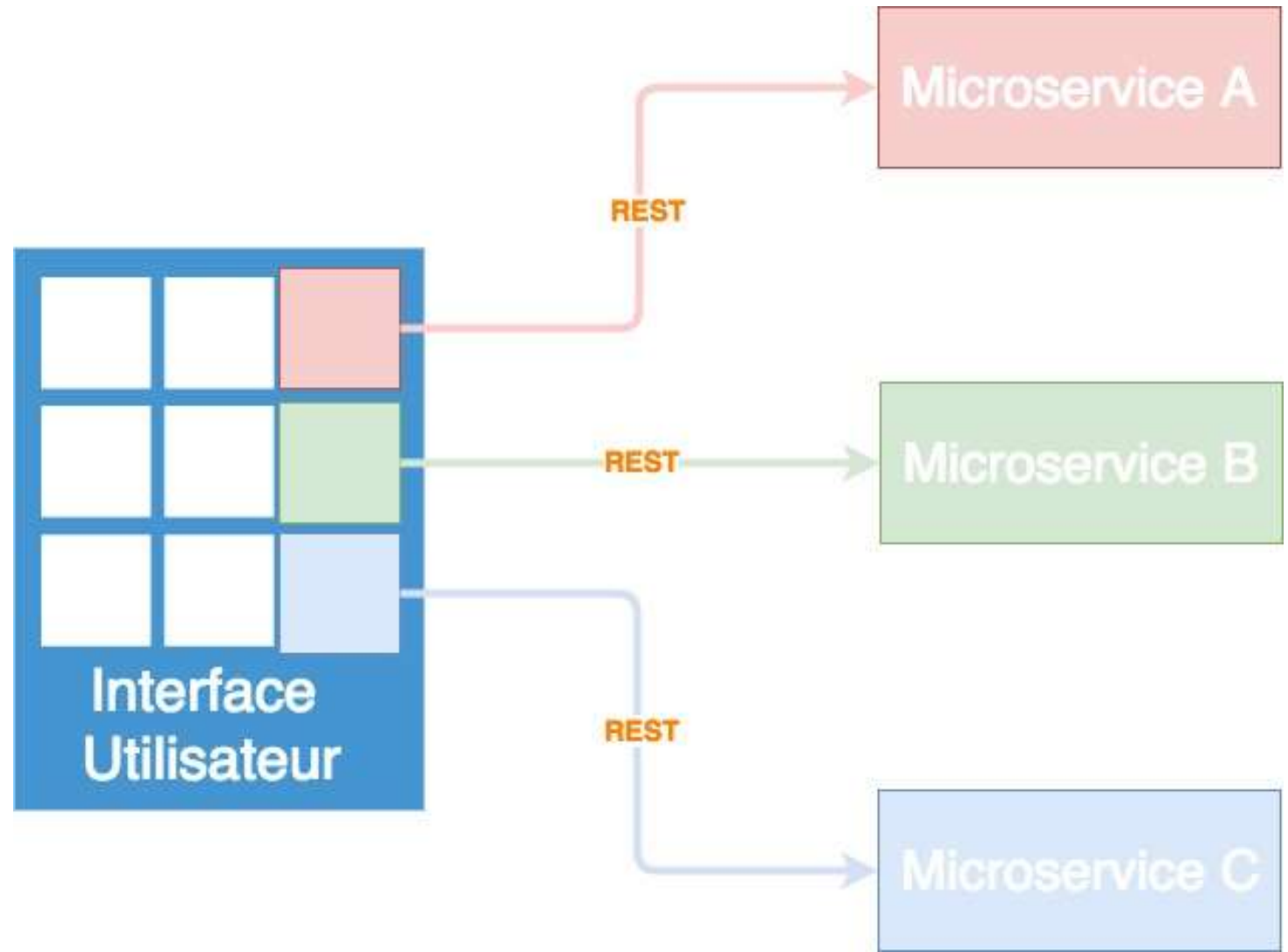


Microservices

Fine-grained

Principe de l'architecture Microservices

- Découper une application en **petits services**, appelés Microservices, **parfaitement autonomes** qui exposent une **API (REST, ...)** que les autres Microservices pourront consommer.



Les API REST

Qu'est-ce qu'une API ?

- API signifie Application Programming Interface.
- Les sites web et les applications ont besoin de communiquer et échanger des données.
- Une API est une interface pour les applications, un framework

REST

- REST signifie “Representational State Transfer”
- Style d’architecture
- Les API REST sont basées sur HTTP
- L’échange est basé sur des requêtes client et serveur
- Ce sont des méthodes qui définissent les requêtes que le client peut effectuer, dont GET, PUT, POST, PATCH, DELETE

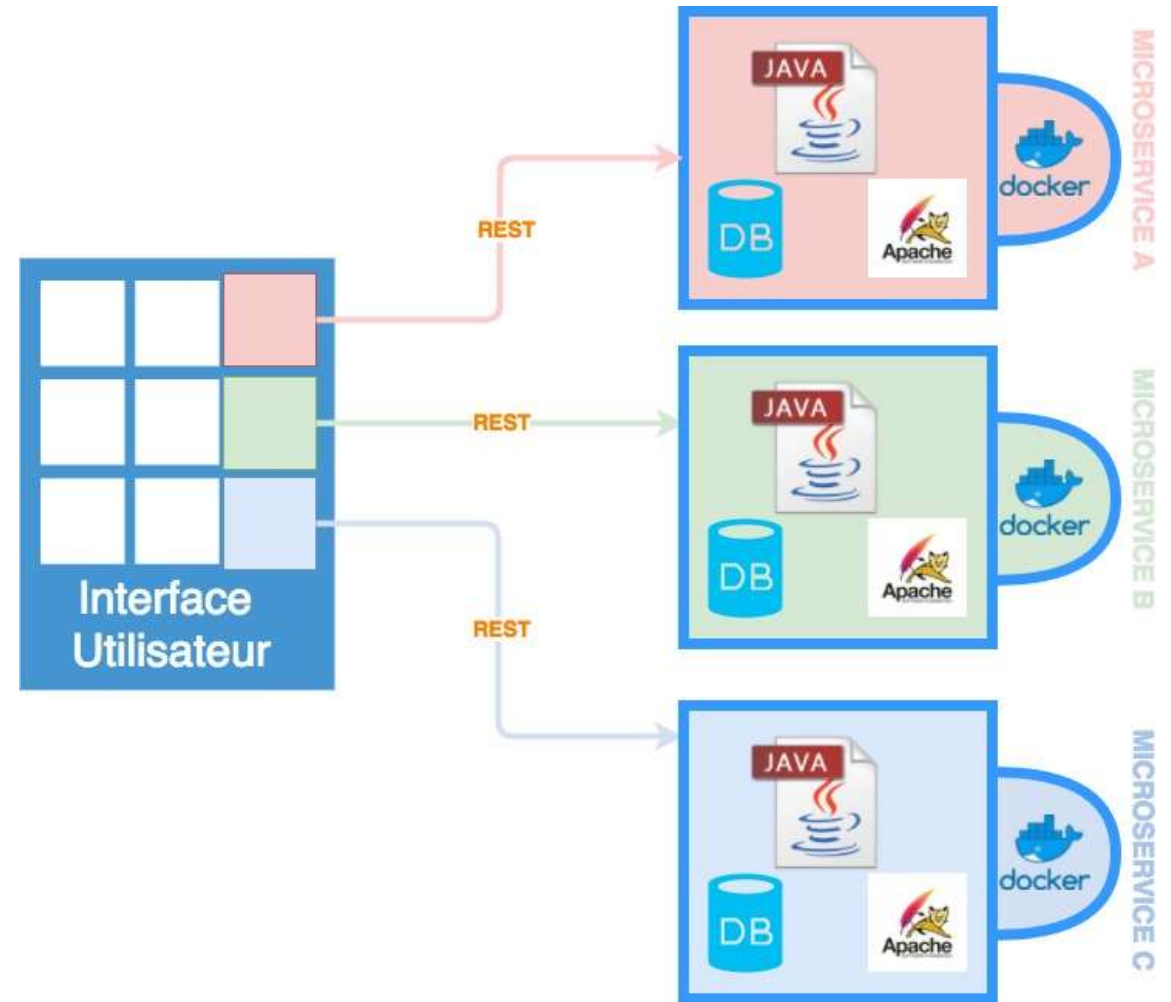
Les critères REST

- Les API REST imitent la façon dont le web lui-même marche dans les échanges entre un client et un serveur. Une API REST est :
 - Sans état
 - Cacheable (avec cache = mémoire)
 - Orientée client-serveur
 - Avec une interface uniforme
 - Avec un système de couche
 - Un code à la demande (optionnel)



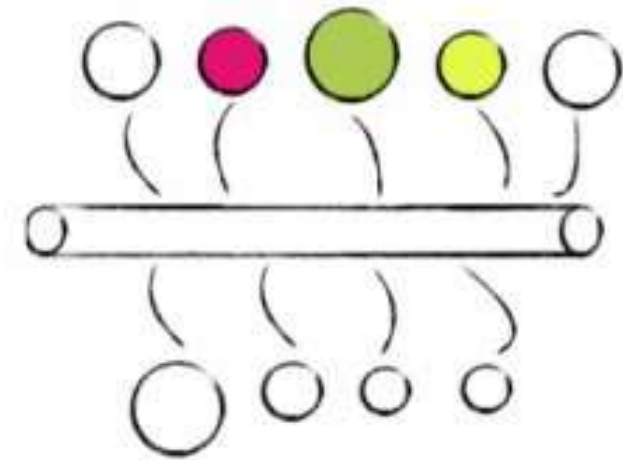
Principe

- Chaque Microservice est parfaitement autonome : il a sa propre base de données, son propre serveur d'application (tomcat, jetty, etc.), ses propres librairies et ainsi de suite.
- La plupart du temps ces Microservices sont chacun dans un container **Docker**, ils sont donc totalement indépendants y compris vis-à-vis de la machine sur laquelle ils tournent.



Microservice : Concepts

- Des concepts présentés dans les slides précédents restent valables (réutilisabilité, autonomie, sans état...)
- Huit nouveaux aspects caractérisent un microservice
 - Fonctionnalité unique
 - Flexibilité technologique
 - Equipe réduite ≤ 10
 - Déploiement ciblé
 - Montée en charge « scalabilité »
 - Tests facilités





Une seule fonctionnalité

- Un microservice doit réaliser une seule fonctionnalité de l'application globale
- Un microservice peut contenir toutes les couches logicielles (IHM, middleware et base de données)
- Un microservice possède un contexte d'exécution séparé des autres (exemple : machine virtuelle ou conteneur)

Flexibilité technologique

- Utiliser les bons langages et frameworks selon la fonctionnalité à réaliser
 - Play/Scala  
 - Java EE (JSF/EJB/Java), Spring   
 - JQuery/AngularJS/NodeJS   
 - Django/Python  
 - Symfony/PHP  

Equipe réduite – two pizza team

- Chaque microservice à sa propre équipe de développement
- L'équipe de développement orientée fonctionnalité est réduite et pluridisciplinaire
 - Développeurs backend
 - Développeurs web
 - Administrateurs bases de données

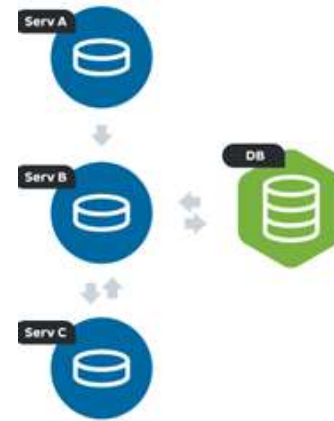


Déploiement ciblé

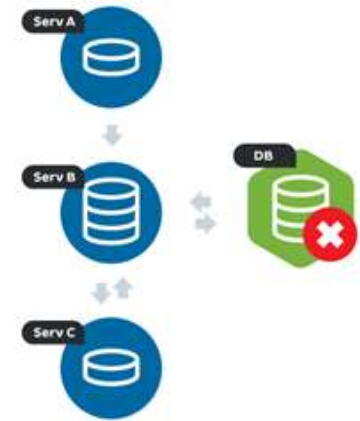
- Evolution d'une certaine partie sans tout redéployer
- Un seul livrable à partir d'un seul code source
- Moins de coordination entre équipes quand il y a un seul déploiement
 - Plus souvent
 - Moins de risque
 - Plus rapide

Montée en charge / scalabilité

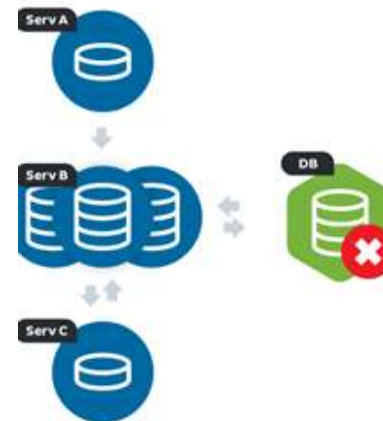
- Forte sollicitation sur un microservice ?
 - page web : Amazon en période de «Black Friday»
 - batch : compression de vidéos
- Comme chaque fonctionnalité est isolée possibilité de multiplier le nombre d'instances d'un microservice.



1. Service B: Thread Pool Exhaustion



2. Add more threads → DB connections exhausted



3. Add more replicas/traffic → Expanded DB is unable to meet demand

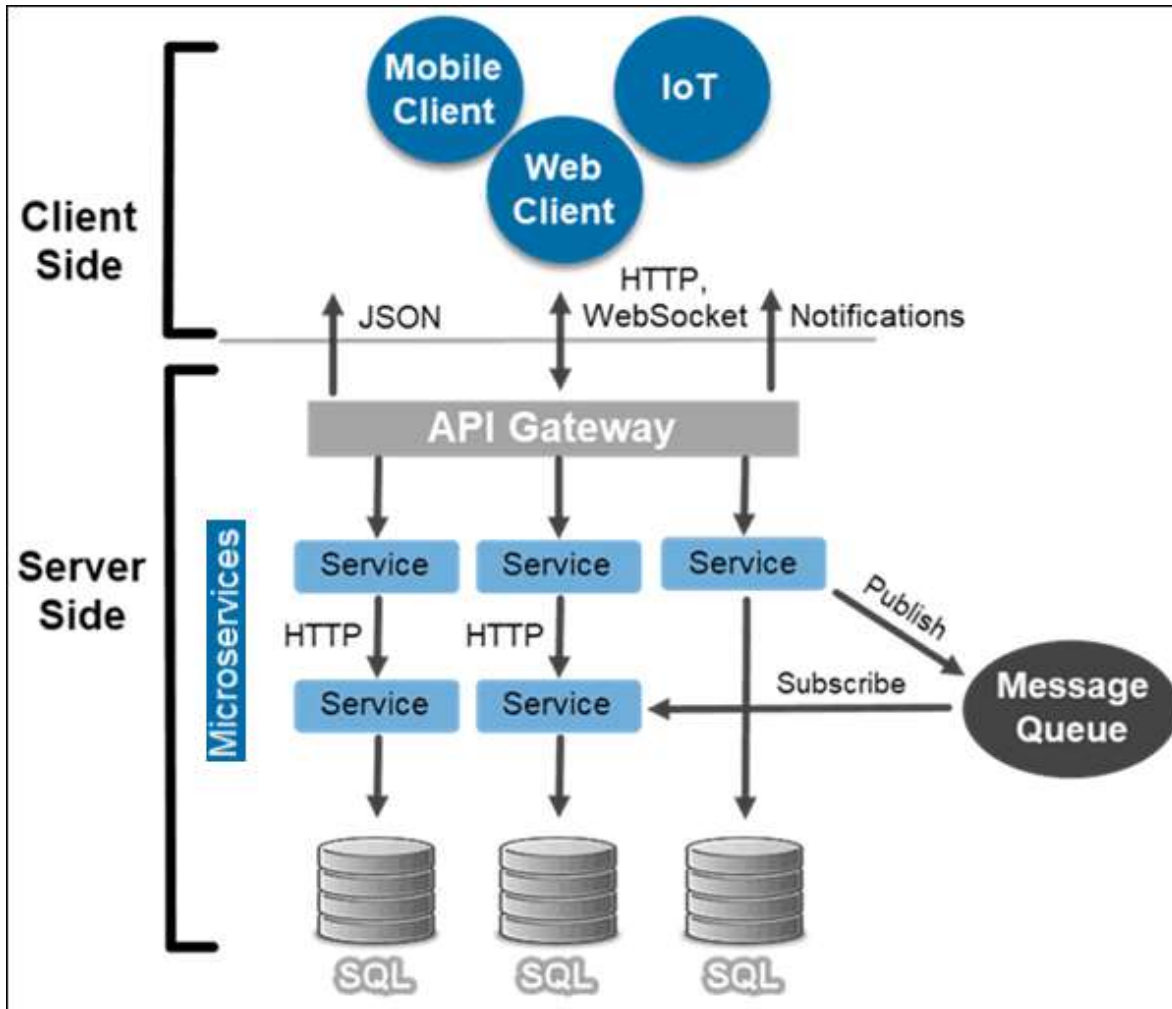


4. Sharding/Distributed strategy in place permitting significant horizontal capacity

Facilite les tests

- Comme chaque fonctionnalité est isolée dans un microservice il est plus facile de les tester,
- Les tests unitaires concernent la partie intrinsèque d'un microservice => on sait déjà faire
- Quels types de tests restants ?
 - Service : stresser l'API exposée du microservice
 - Bout-en-bout : intégrer les autres microservice
- Si c'est facile à tester alors c'est facile à déployer (principe du déploiement continu)

Architecture microservice



- Une architecture microservice est donc un ensemble de microservices autonomes
- Pas de bus d'intégration (ESB) car il y a de la logique et donc cela force le couplage
- Utilisation d'un Bus d'événements pour le faible couplage
- Utilisation d'un Load Balancer pour équilibrer la charge

Composer

- La composition va permettre de résoudre le problème d'orchestration des conteneurs
- Exemple simple pour comprendre cette problématique
 - A = un conteneur pour la base de données (MySQL)
 - B = un conteneur applicatif
 - A doit être démarré avant B et B a besoin d'accéder à A (ouverture de port)
- Solutions du marché
 - Docker Compose (Docker inc.) => le plus simple
 - Docker Swarm (Docker inc.)
 - Kubernetes (Google)
 - Mesos (Apache)
 - Fleet (CoreOS)
 - Crane (<https://github.com/michaelsauter/crane>)
 - Consul

Répartir la charge : reverse proxy / LB

- Problématique du pourquoi utiliser un reverse proxy
 - Plusieurs conteneurs Docker vont être créés
 - Plusieurs conteneurs d'une même image peuvent être créées
- Reverse Proxy ou « Proxy inversé »
 - Donne accès depuis un réseau externe aux conteurs d'un réseau interne
 - Distribuer en fonction de la charge les requêtes web aux conteneurs les moins occupés (d'une même image par exemple)
- Solutions libres du marché
 - Nginx (Nginx inc.) => largement utilisé et très simple
 - Haproxy
 - Hipache (Docker inc.)
 - Træfik



Défis des microservices

- Les MicroServices dépendent les uns des autres et ils devront communiquer entre eux.
- Par rapport aux systèmes monolithiques, il existe davantage de services à surveiller qui sont développés à l'aide de différents langages de programmation .
- Comme il s'agit d'un système distribué, il s'agit d'un modèle intrinsèquement complexe.
- Différents services auront leur mécanisme séparé, ce qui entraînera une grande quantité de mémoire pour des données non structurées.
- Gestion efficace et travail d'équipe requis pour éviter les problèmes en cascade
- Reproduire un problème sera une tâche difficile lorsqu'il est parti dans une version et revient dans la dernière version.

Défis des microservices

- Le déploiement indépendant est compliqué avec les microservices.
- L'architecture des microservices entraîne de nombreux frais généraux d'opérations.
- Il est difficile de gérer l'application lorsque de nouveaux services sont ajoutés au système
- Un large éventail de professionnels qualifiés est nécessaire pour prendre en charge les microservices distribués de manière hétérogène

Défis des microservices

La pile technologique des microservices pourrait être très volumineuse

Par nature les applications microservice sont Full Stack

La logique métier ne réside plus dans l'application

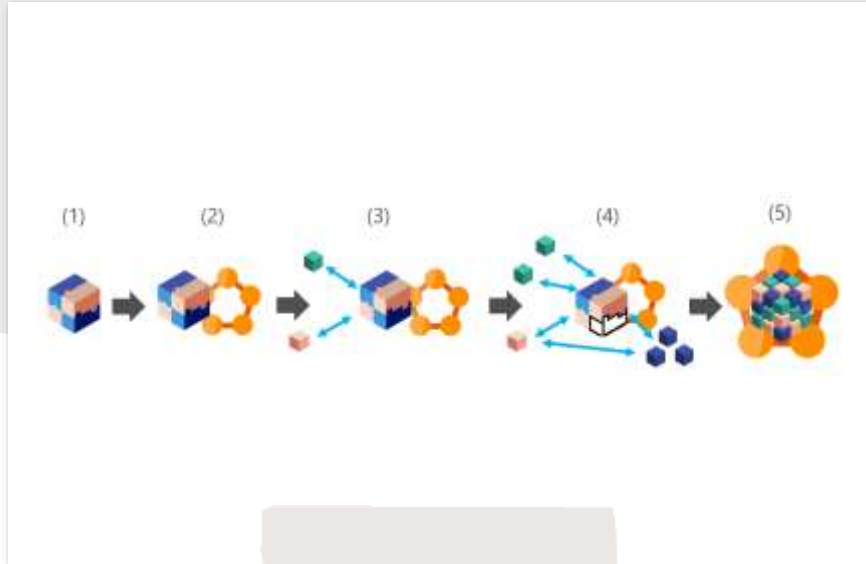
Bilan

- Le 'Time to market'
 - Création de valeur
 - Réactivité aux évolutions du marché
 - Satisfaction des utilisateurs
- L'agilité technologique
- Modernisation facilitée
- Évolutivité
- Fiabilité

Bilan

- L'architecture microservices n'est pas la solution à tous les problèmes de performances d'un applicatif
- Problèmes courants
 - Modélisation
 - Persistance des données
 - Transaction
 - Front-end non adapté
- Les microservices adressent le problème d'architecture et sont là pour faciliter la montée en charge
- Attention : il n'a jamais été précisé que les architectures monolithiques étaient mortes

Pour aller plus loin



- Les bonnes pratiques pour passer à une architecture

microservices => à partir d'une application monolithique

- Déploiement vers des clusters
- Monitoring
- Transaction
- Sécurité

The Twelve-Factors App



Introduction

12 factor Application est une méthodologie pour concevoir des logiciels en tant que service qui :

- Utilisent des formats déclaratifs pour mettre en œuvre l'automatisation, pour minimiser le temps et les coûts pour que de nouveaux développeurs rejoignent le projet;
- Ont un contrat propre avec le système d'exploitation sous-jacent, offrant une portabilité maximum entre les environnements d'exécution;
- Sont adaptés à des déploiements sur des plateformes cloud modernes, rendant inutile le besoin de serveurs et de l'administration de systèmes;
- Minimisent la divergence entre le développement et la production, ce qui permet le déploiement continu pour une agilité maximum;
- et peuvent grossir verticalement sans changement significatif dans les outils, l'architecture ou les pratiques de développement;

12 Factors



Les 12 facteurs

I. Base de code

- Une base de code suivie avec un système de contrôle de version, plusieurs déploiements

II. Dépendances

- Déclarez explicitement et isolez les dépendances

III. Configuration

- Stockez la configuration dans l'environnement

Les 12 facteurs

IV. Services externes

- Traitez les services externes comme des ressources attachées

V. Assemblez, publiez, exécutez

- Séparez strictement les étapes d'assemblage et d'exécution

VI. Processus

- Exécutez l'application comme un ou plusieurs processus sans état

VII. Associations de ports

- Exportez les services via des associations de ports

Les 12 facteurs

VIII. Concurrency

- Grossissez à l'aide du modèle de processus

IX. Jetable

- Maximisez la robustesse avec des démarrages rapides et des arrêts gracieux

X. Parité dev/prod

- Gardez le développement, la validation et la production aussi proches que possible

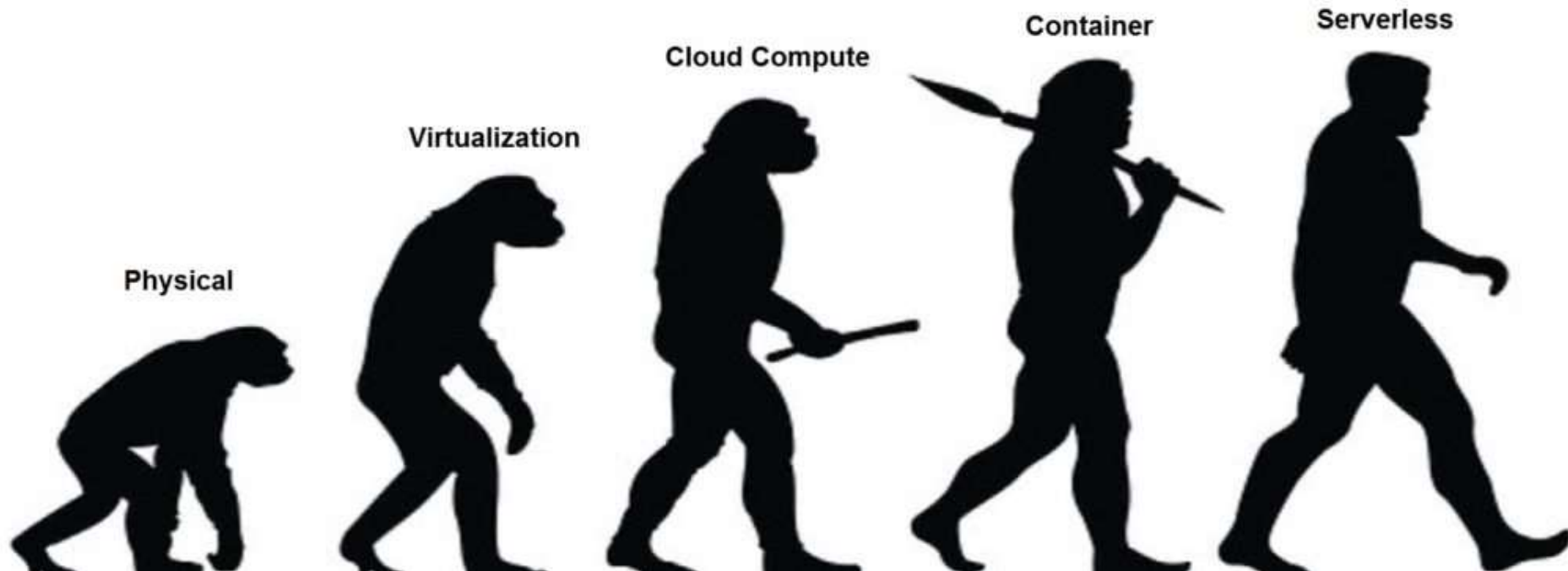
XI. Logs

- Traitez les logs comme des flux d'évènements

XII. Processus d'administration

- Lancez les processus d'administration et de maintenance comme des one-off-processes

Et du côté des Ops !!!

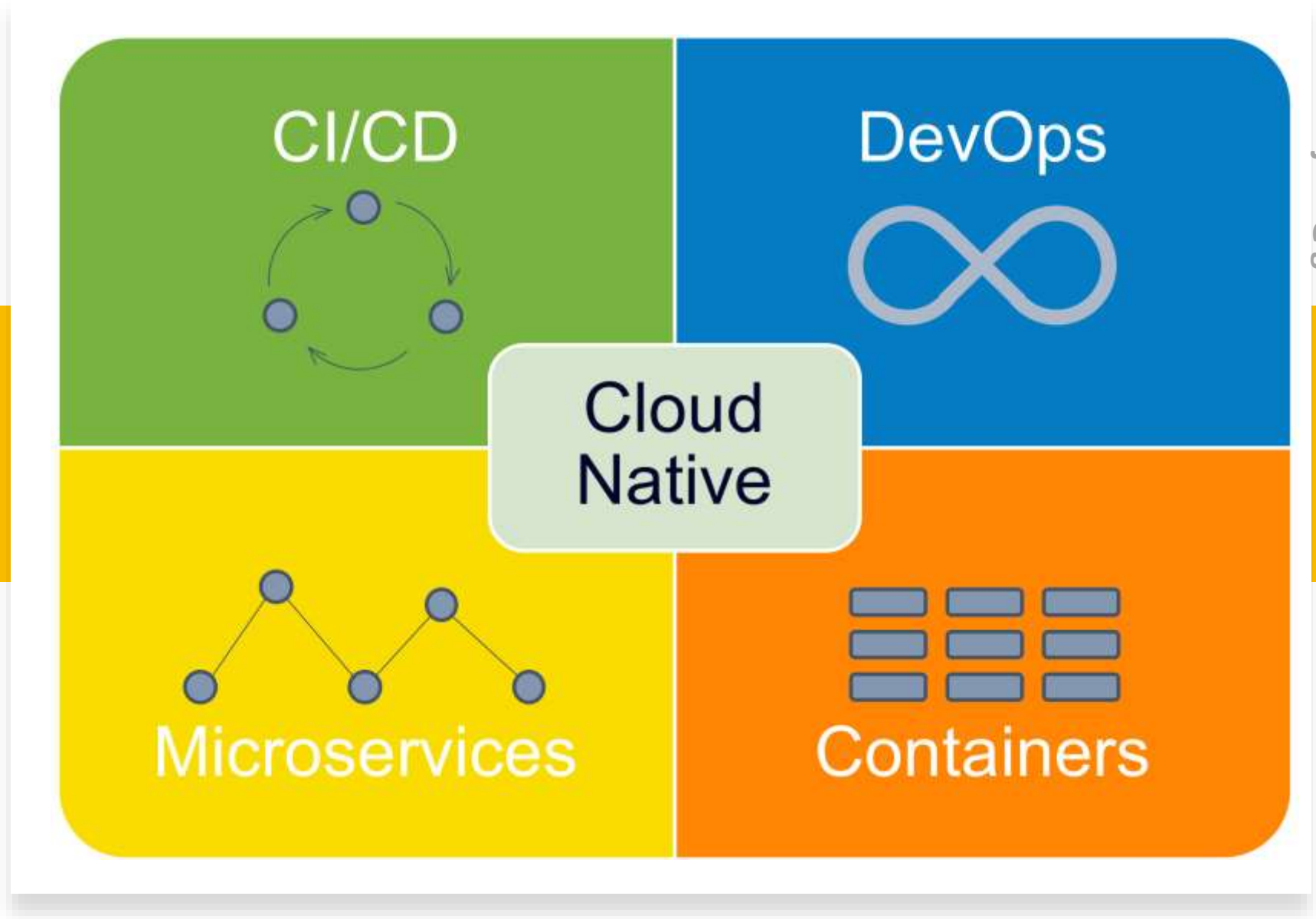



DevOps et le cloud

L'adoption du cloud a profondément transformé la manière dont les équipes créent, déploient et exploitent des applications. Parallèlement à l'adoption de DevOps, les équipes ont pu désormais améliorer leurs pratiques et mieux servir leurs clients.

Briques de base du Cloud Native

V2 03-22





Agilité cloud

- La possibilité d'approvisionner et de configurer rapidement des environnements cloud multirégions dotés de ressources illimitées renforce l'agilité avec laquelle les équipes déploient leurs applications.
- Désormais, au lieu d'acheter, de configurer et d'assurer la maintenance de serveurs physiques, les équipes créent des environnements cloud complexes en quelques minutes, puis les arrêtent dès qu'ils deviennent inutiles.



Kubernetes

- Alors que de plus en plus d'applications utilisent la technologie de conteneur, Kubernetes s'impose pour orchestrer les conteneurs à grande échelle.
- L'automatisation des processus de création et de déploiement de conteneurs via des pipelines CI/CD et la supervision de ces conteneurs en production relèvent de pratiques incontournables à l'ère de Kubernetes.



Informatique serverless

- L'essentiel des tâches de gestion de l'infrastructure ayant été transféré au fournisseur de cloud, les équipes peuvent se concentrer sur leurs applications plutôt que sur l'infrastructure sous-jacente.
- L'informatique Serverless offre la possibilité d'exécuter des applications sans configuration ni maintenance de serveurs.
- Certaines options réduisent la complexité et les risques liés au déploiement et aux opérations.

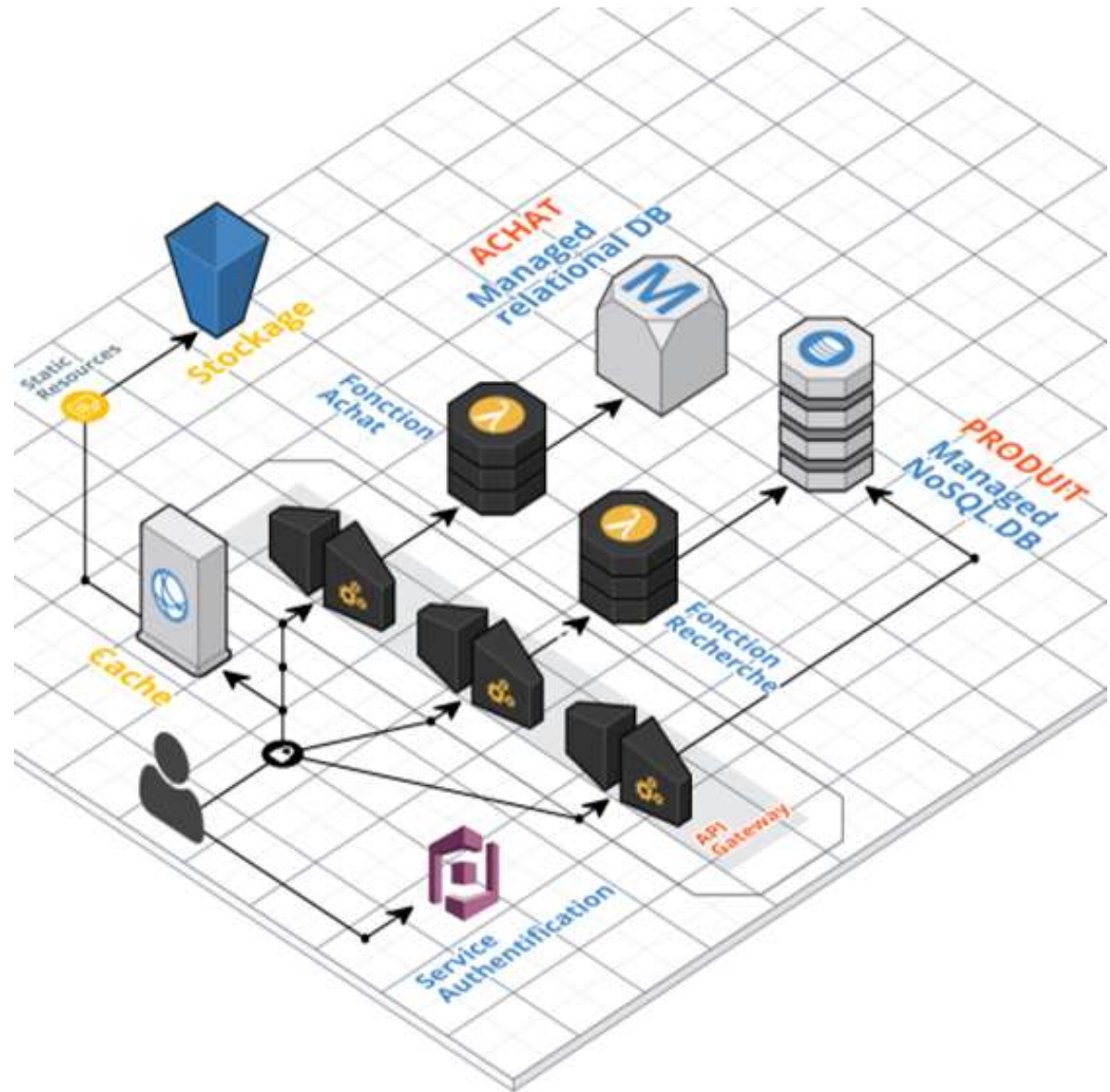
Qu'est-ce que le Serverless

- Applications qui dépendent de services tiers (Backend en tant que service ou «BaaS»).
- De nos jours la migration de nombreuses fonctionnalités côté FrontEnd nous a permis de supprimer nos besoins de serveurs “Always On”.
- Réduire le coût et la complexité opérationnels et ainsi payer uniquement les frais d'utilisations (bande passante, volume de stockage).
- Aussi connu comme **pay-as-you-go**.

Qu'est-ce que le Serverless

- Serverless peut également symboliser des applications où une certaine quantité de logique serveur est toujours écrite par le développeur, mais contrairement aux architectures traditionnelles (exemple: Monolith), elle est exécutée dans des conteneurs stateless qui sont déclenchés par le biais d'événements, éphémère (uniquement une invocation) et entièrement gérée par une 3rd party.
- Une façon de penser à ceci est le terme Function as a service ou FaaS.
- AWS Lambda est l'une des implémentations les plus populaires de FaaS à l'heure actuelle, mais il y en a d'autres (Google Functions).

Exemple site e-commerce



Use Case

Introduction

- Nous allons commencer par examiner une société de commerce électronique fictive, Mike's Electronics, et voir comment l'introduction de l'intégration continue et la livraison continue apportent des améliorations considérables à leurs systèmes informatiques.
- Ces améliorations comprennent un cycle de développement logiciel plus court, une mise sur le marché plus rapide et une augmentation des revenus de l'entreprise.
- À la fin de ce module, vous comprendrez les concepts de base de l'intégration continue, de la livraison continue et de DevOps.



Intégration

La manière Old School

Etude de cas



Public Website



Software Development Team



Product Management

Notre société fictive s'appelle Mike's Electronics.

Ils sont un détaillant en ligne pour toutes sortes de produits électroniques.

Ils vendent en ligne sur mikeselectronics.com.

Ce site Web est essentiellement une application Web développée en interne par leur équipe informatique.

Il y a beaucoup de travail à faire pour créer et maintenir cette application Web.

Une équipe de gestion des produits définit la feuille de route fonctionnelle du produit en effectuant des études de marché et des enquêtes auprès des clients.

Nouvelles exigences du marché



Public Website



Software Development Team

Ils interagissent étroitement avec l'équipe de développement logiciel.

Cette équipe prend les contributions de la gestion des produits et travaille sur la mise en œuvre des fonctionnalités demandées.



Product Management





Tout le code est conservé dans un référentiel de code source.

Git est un système de versioning de code source qui a acquis une acceptation universelle au cours de la dernière décennie. Il a diverses fonctionnalités comme le versioning, la création de branches, le balisage, etc.

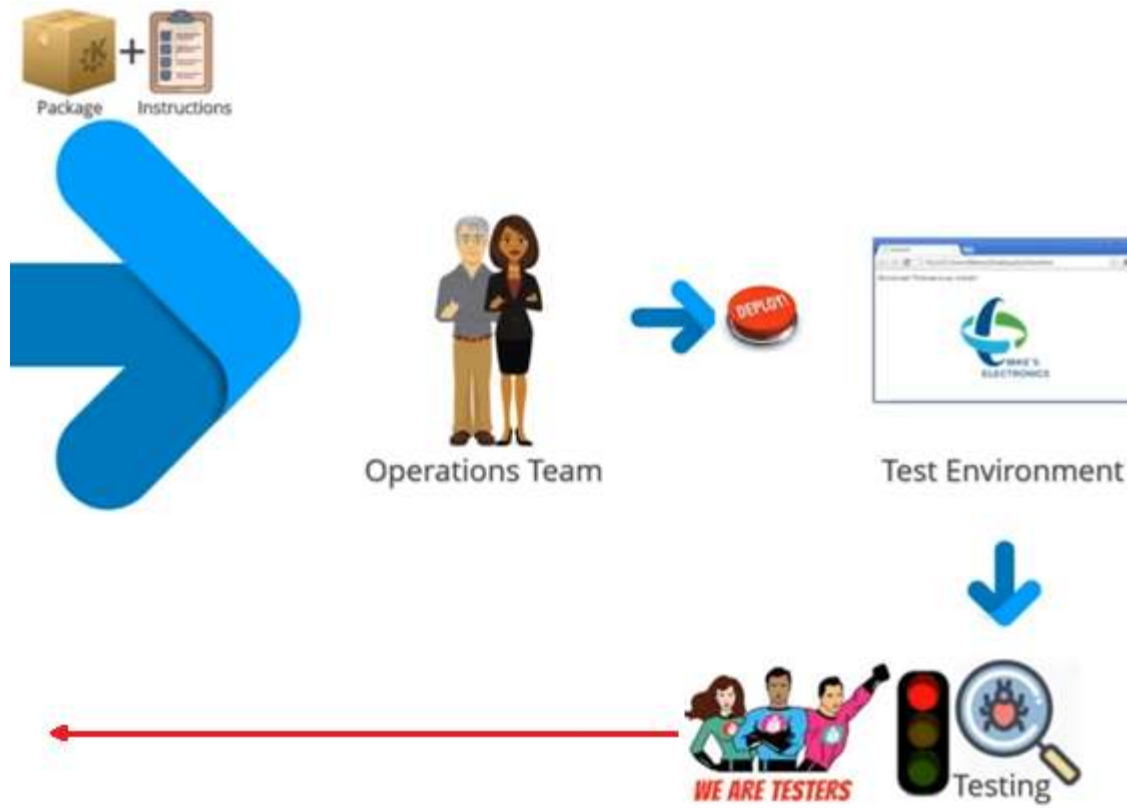
Il existe également une équipe Build & Integration qui intègre le code de tous les programmeurs. Ensuite, ils le compilent et en construisent un progiciel.

Dev Team / Build & Integration Team



Ce package est ensuite transmis, accompagné de quelques instructions, à l'équipe Opérations qui le déploie dans un environnement de test.

Operations Team



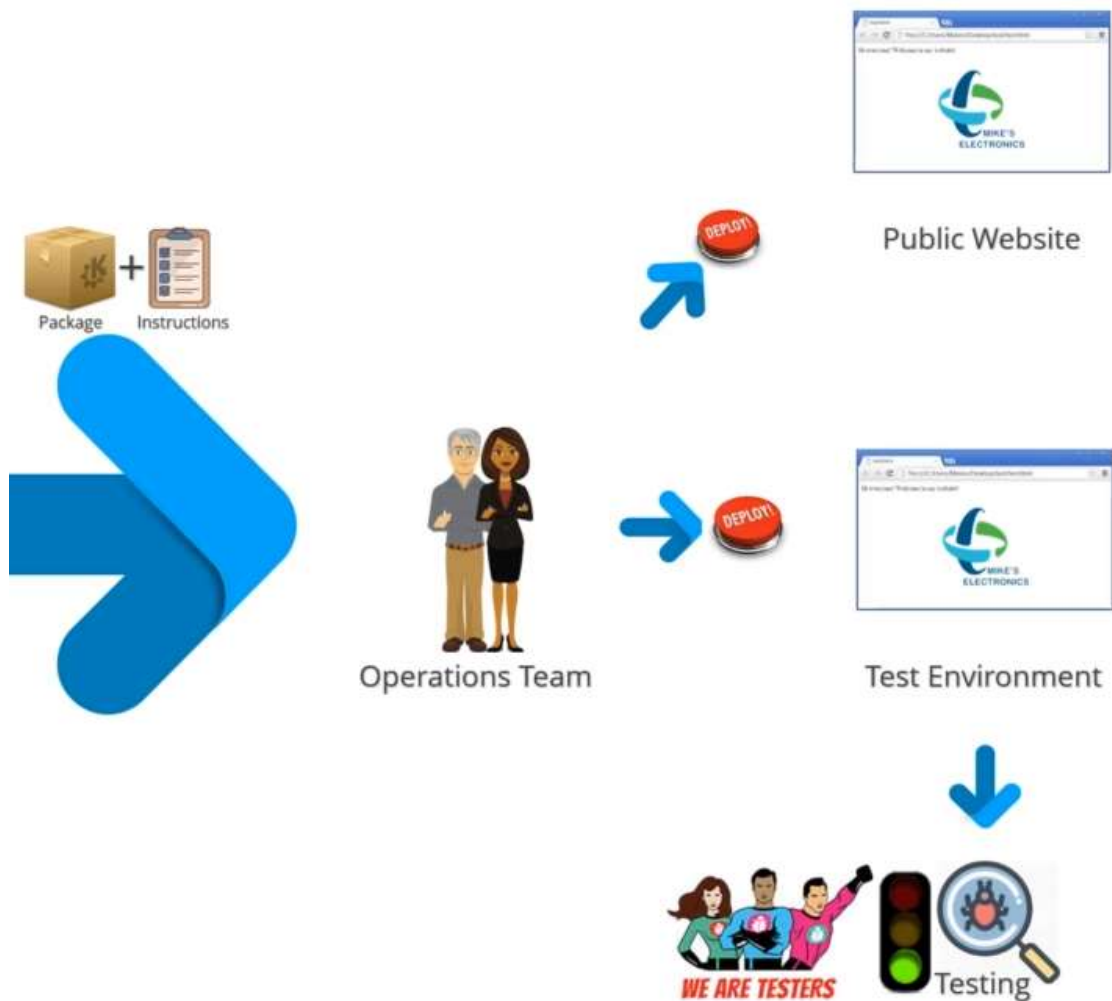
Maintenant vient l'équipe d'assurance qualité. Ils testent le logiciel pour les défauts et s'assurent que les fonctionnalités fonctionnent comme prévu.

Tout défaut trouvé est signalé aux programmeurs, puis l'ensemble du processus est répété

D'abord en réparant le code, puis en l'intégrant, en le construisant et en créant un nouveau package, puis en le redéployant.

L'assurance qualité travaille en étroite collaboration avec la gestion des produits pour s'assurer que la gestion des produits obtient ce qu'ils ont demandé.

Quality Assurance Team



Une fois que l'assurance qualité donne son feu vert, l'équipe des opérations prend la dernière version en date, suit les instructions qui leur sont données par les développeurs et la déploie en production.

À ce stade, les nouvelles fonctionnalités sont disponibles sur le site Web, mikeselectronics.com.

go-ahead



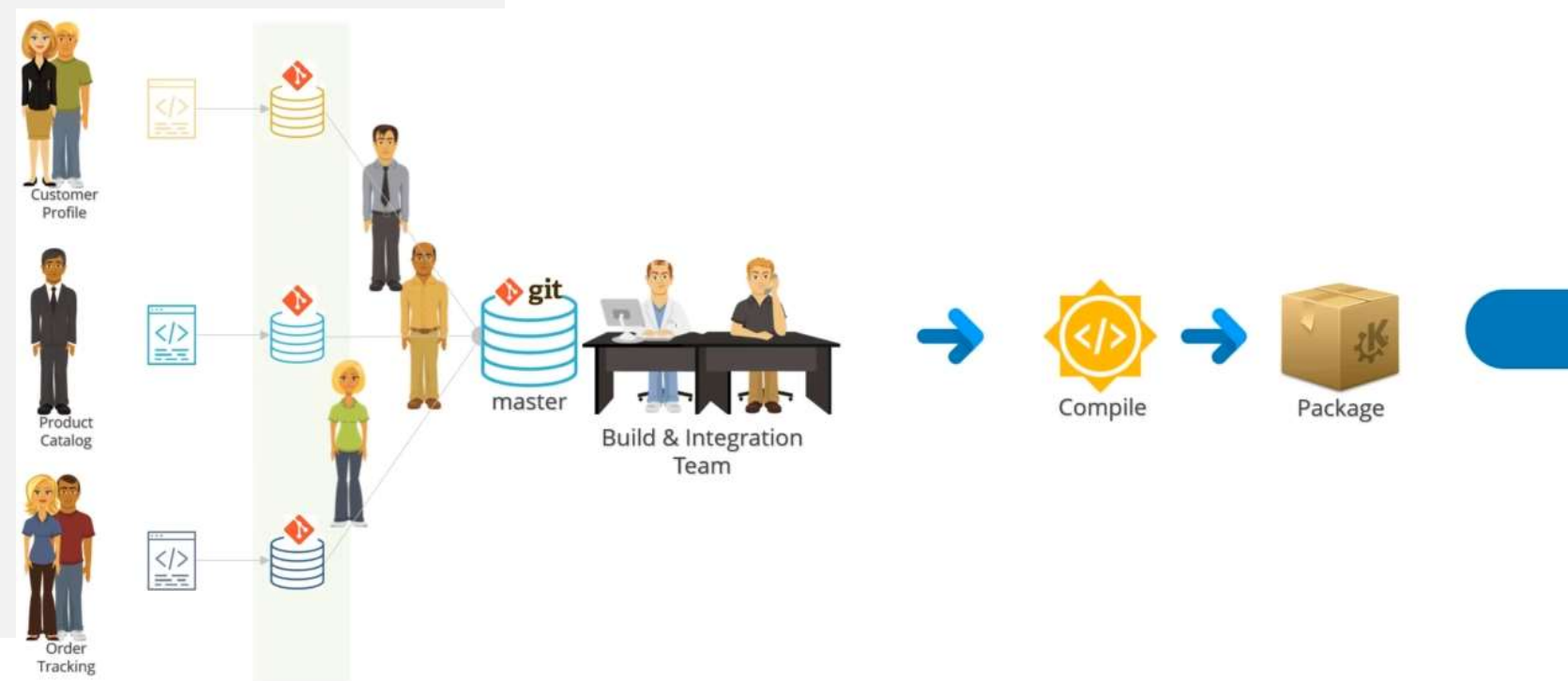
Tout ce cycle peut durer de quelques semaines à quelques mois. Un tel cycle est appelé une **itération**.
Nous allons diviser l'équipe informatique en deux départements, l'un pour le développement et l'autre pour les opérations.

Itération

Inconvénients de l'ancien modèle



Problème n ° 1.



- La fusion du code des branches de source sur la branche principale et son intégration ne se produit qu'une seule fois dans une itération.
- L'équipe d'intégration est souvent assistée par au moins un développeur de chacune des équipes du module pour comprendre comment intégrer le code en une seule unité de travail.
- Ils consomment beaucoup d'efforts et le processus est sujet aux erreurs car il s'agit principalement d'un processus manuel.

Problème n ° 2.

- Nous avons vu que l'intégration du code entre les modules se produit vers la fin de l'itération.
- C'est la seule fois dans l'itération où les équipes peuvent voir si leur code fonctionne avec le code des autres modules.
- Si ce n'est pas le cas, cela peut entraîner beaucoup de retouches.
- De plus, comme les défauts sont détectés tardivement, les développeurs sont probablement passés à travailler sur d'autres fonctionnalités et auront du mal à se rappeler les moindres détails liés au défaut.
- Cela leur prendra beaucoup de temps et d'efforts pour revenir à ces morceaux de code et déboguer.
- Faites-moi confiance quand je dis, la plupart des programmeurs ont une mémoire à très court terme en ce qui concerne le code qu'ils ont écrit.

Problème n ° 3.

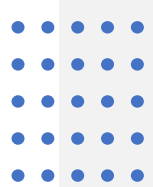
- Si un développeur introduit un défaut fonctionnel dans le code, il y a une chance qu'il reste là inaperçu pendant longtemps.
- Les défauts fonctionnels n'introduisent pas nécessairement des erreurs de compilation, ils peuvent donc ne pas être découverts tant que les tests appropriés ne sont pas effectués sur l'application déployée.
- Le développeur aurait fait ses propres tests unitaires, mais il y a une limite à ce que les tests unitaires peuvent détecter.
- Le code du développeur, ainsi que le code de tous les autres développeurs, doit d'abord être fusionné, intégré et compilé afin d'obtenir une construction stable.

Problème n ° 3. (suite)

- Ensuite, le package généré doit être transmis à l'équipe des opérations, qui le déploie sur un serveur de test.
- Ce n'est qu'à ce stade que le développeur peut voir son code fonctionner en totalité.
- Et probablement remarquer le défaut fonctionnel lui-même ou être averti par l'équipe QA.
- Il s'agit d'un cycle de rétroaction très long.
- Idéalement, le développeur devrait pouvoir voir son code en action dès que possible.
- Les longs cycles de rétroaction pour les développeurs ne sont donc pas recommandés.

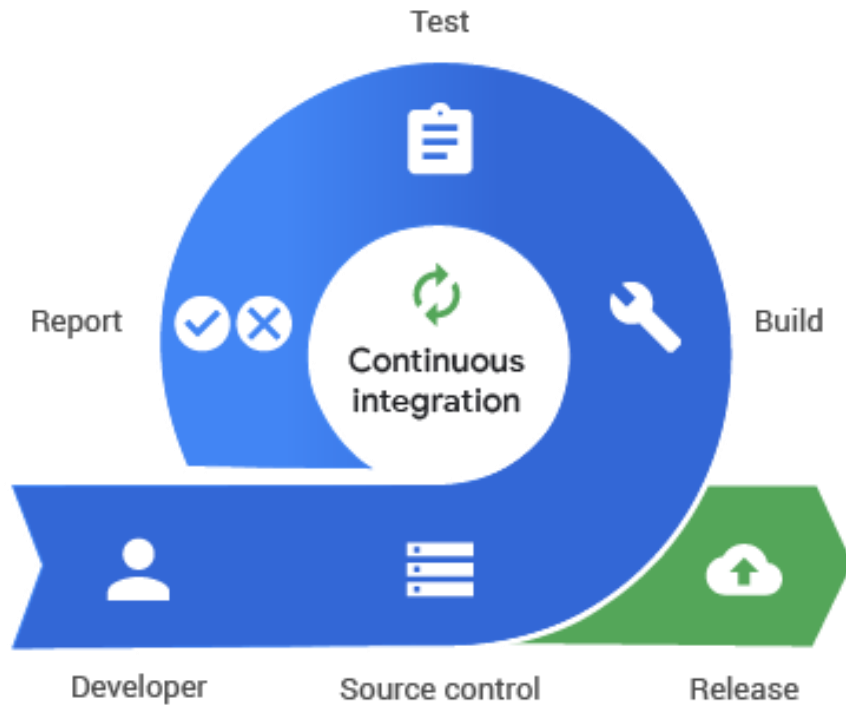
Problème n ° 4.

- En raison de l'énorme quantité de temps et d'efforts impliqués dans la fusion et l'intégration manuelle du code, ainsi qu'en raison de la détection tardive des défauts et des longs cycles de rétroaction, une seule itération peut généralement durer de quelques semaines à quelques mois.
- À l'ère numérique dans laquelle nous nous trouvons, c'est beaucoup de temps.
- Si la direction du produit souhaite déployer une fonctionnalité mineure en production, vous ne pouvez pas toujours lui demander d'attendre aussi longtemps.
- Aujourd'hui, lorsque les marchés sont perturbés en un clin d'œil, cette longue période d'attente est totalement injustifiée.



Résumons
maintenant tous
les points
douloureux.

- Beaucoup de temps et d'efforts impliqués dans l'intégration manuelle.
- Correction des problèmes à la fin des itérations en raison de la détection tardive.
- Les problèmes de fusion intermédiaires dans le code conduisent souvent à des builds cassés, bloquant ainsi les développeurs.
- Longs cycles de rétroaction pour les défauts.
- Des itérations plus longues conduisant à un délai de commercialisation plus long pour les entreprises.




Continuous Integration



Définition

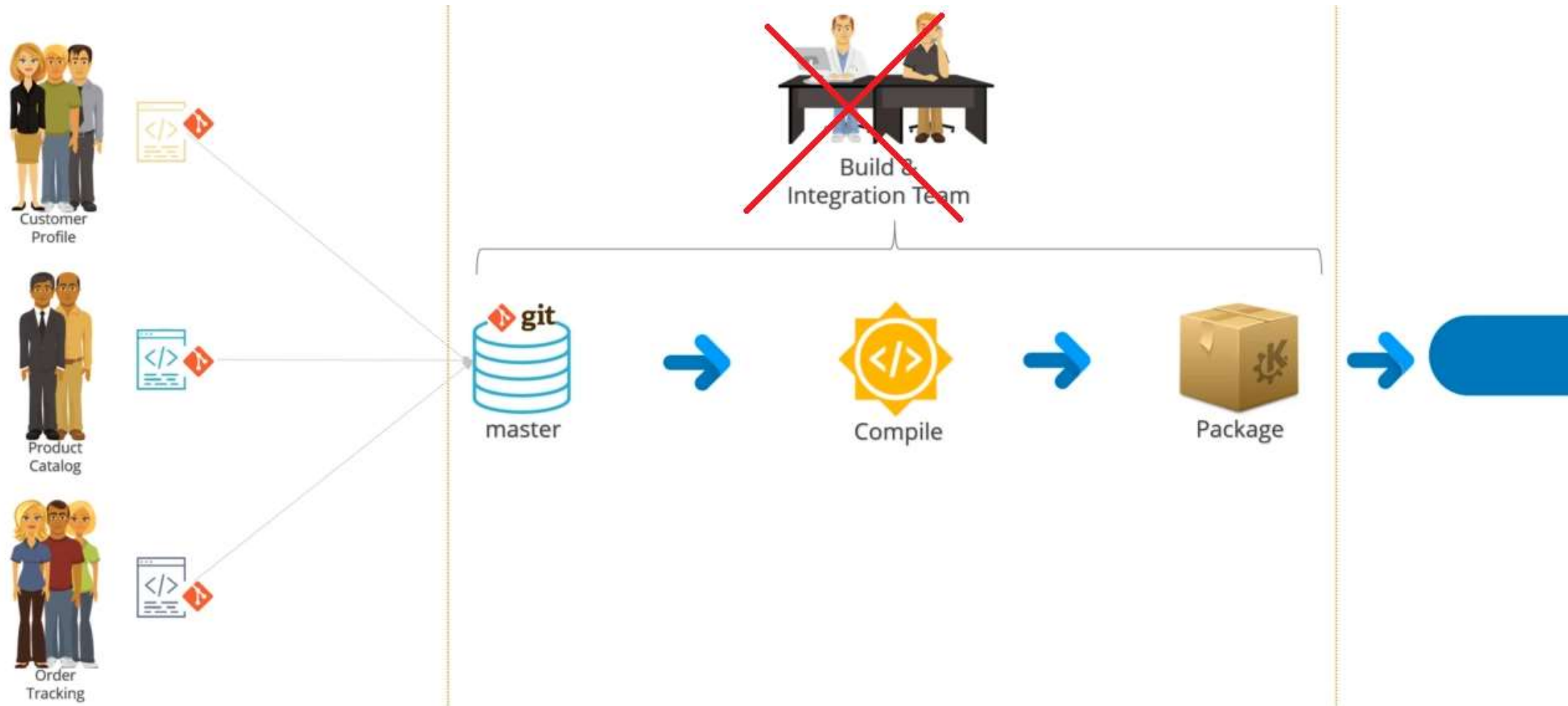
- L'intégration continue est une pratique de développement qui oblige les développeurs à intégrer du code dans un dépôt partagé plusieurs fois par jour.



Supprimer l'équipe Build & Integration

- Dans le développement traditionnel, les développeurs archivent leur code dans des branches de contrôle de source spécifiques au module.
- L'intégration dans la branche principale se produit en fin d'itération.
- Désormais tous les développeurs devraient utiliser le même référentiel de contrôle de source.
- Ils peuvent toujours utiliser différentes branches, mais ils doivent souvent fusionner avec la branche principale, plusieurs fois par jour.
- Cela les obligera à intégrer le code eux-mêmes.
- Donc, nous pouvons faire à moins de l'équipe Build & Integration et distribuer cette tâche sur tous les développeurs à la place.
- Nous pouvons maintenant complètement supprimer l'équipe Build & Integration.

Supprimer l'équipe Build & Integration



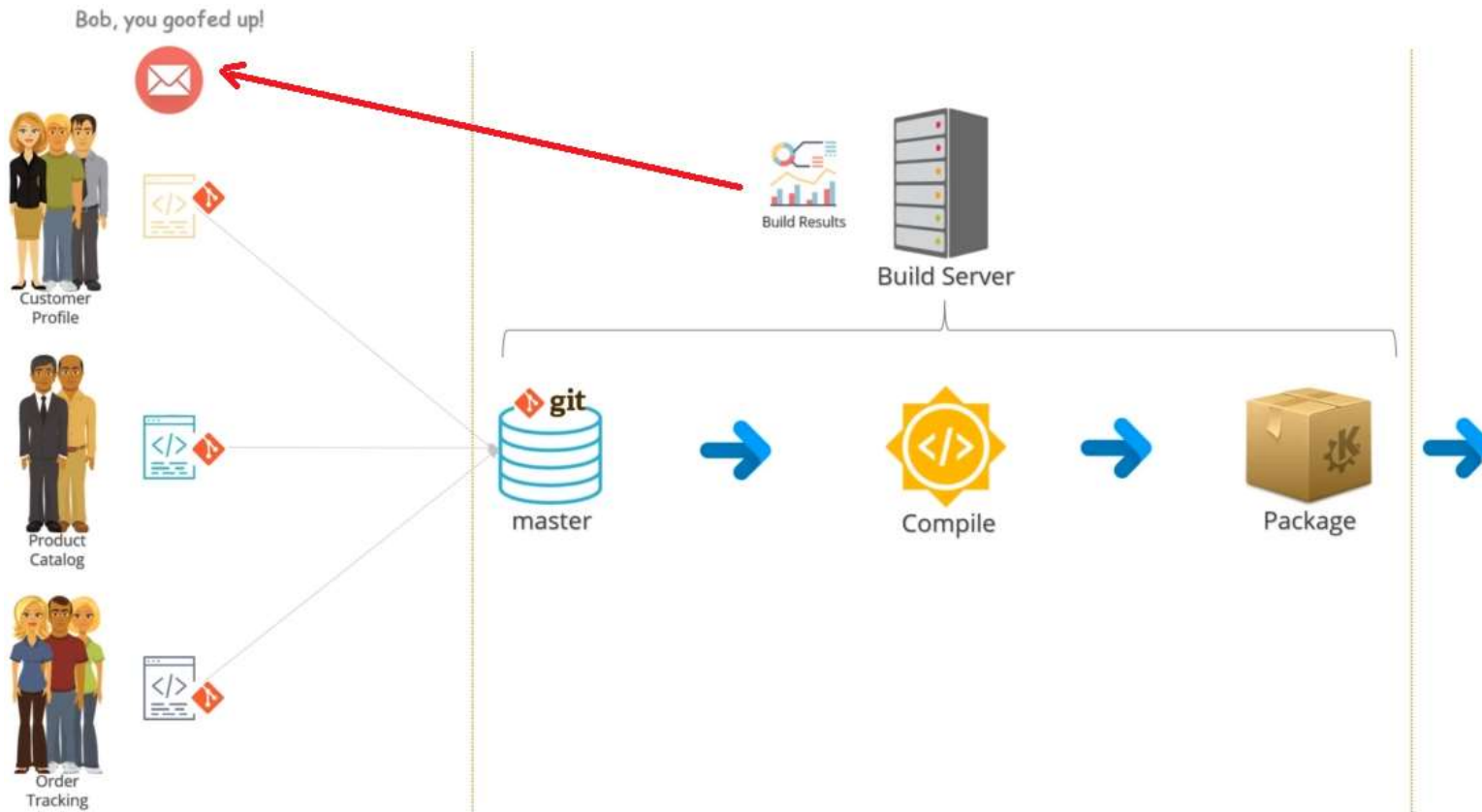
Serveur de build

- Les développeurs connaissent bien leur code, ils peuvent alors passer un peu plus de temps et l'intégrer, afin qu'ils puissent s'assurer que «leur» code fonctionne avec le reste du code.

Le prochain changement que nous devons apporter est le suivant:

- Pour vous assurer que le code s'assemble et fonctionne réellement, il doit être compilé régulièrement, c'est-à-dire après chaque validation de code par n'importe quel développeur à tout moment.
- Ainsi, nous allons maintenant utiliser un serveur de build.

Serveur de build (automatisation)

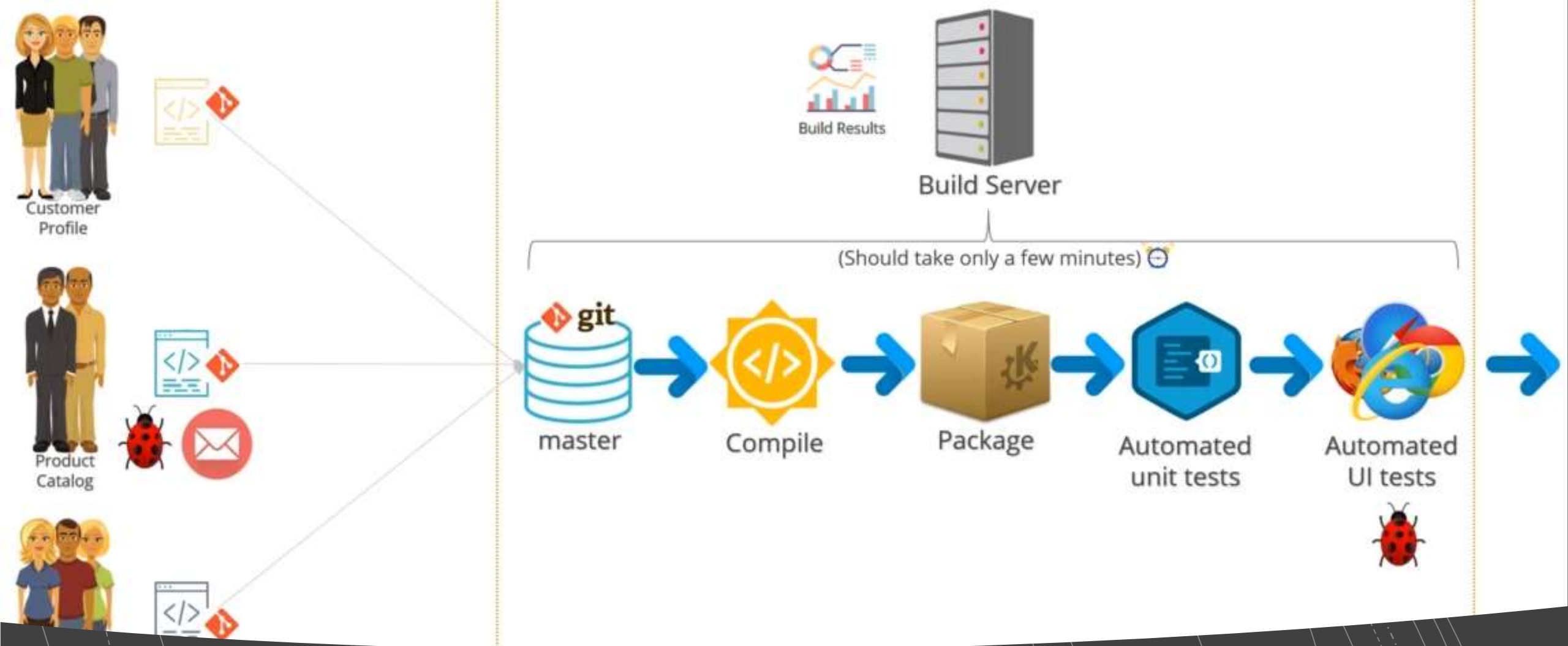


Chaque fois qu'un développeur « check in » du code, il doit être compilé sur ce serveur de build dédié.

L'ensemble du processus de génération doit être déclenché **automatiquement** lorsqu'un développeur effectue une validation.

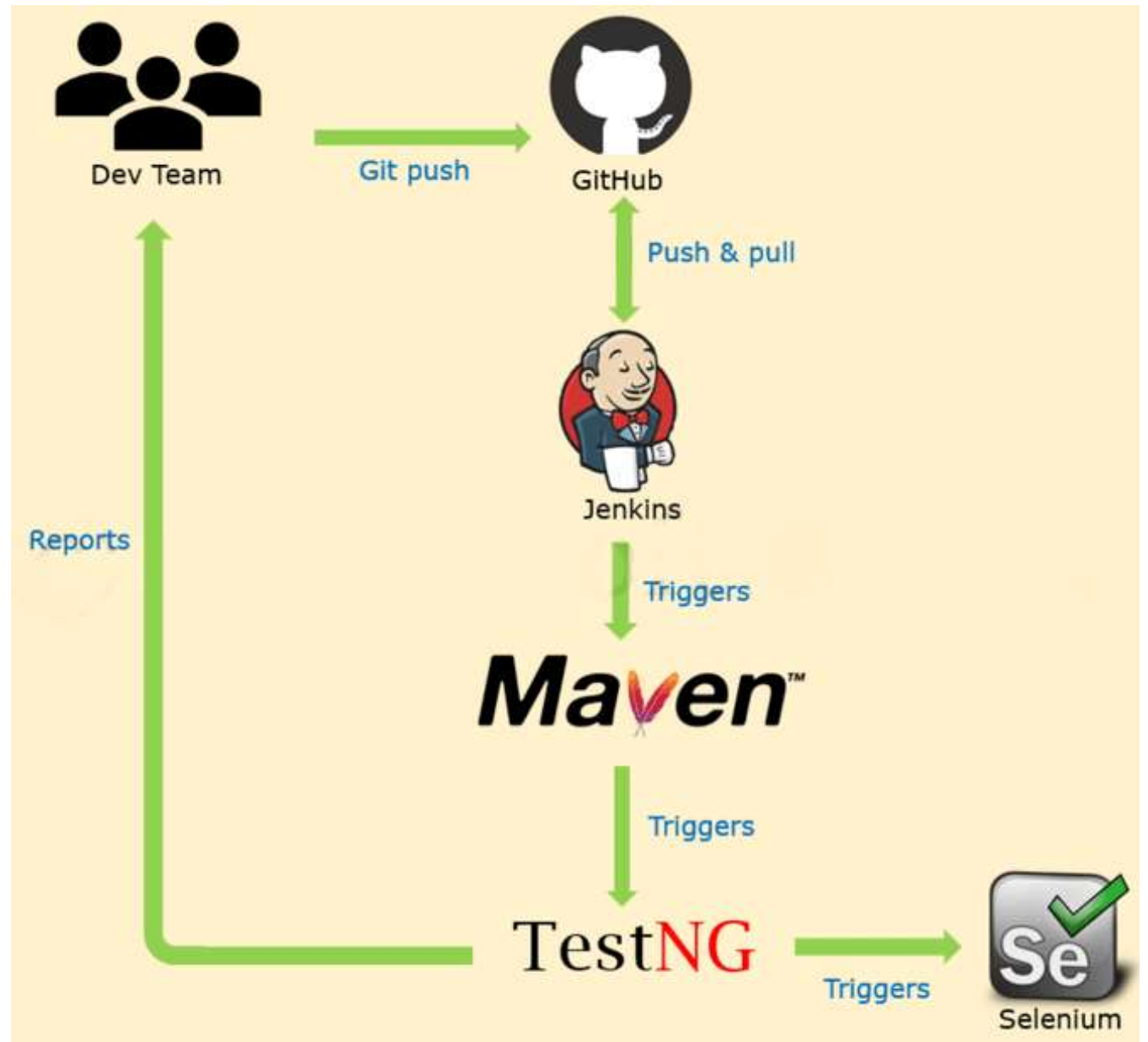
Serveur de Build

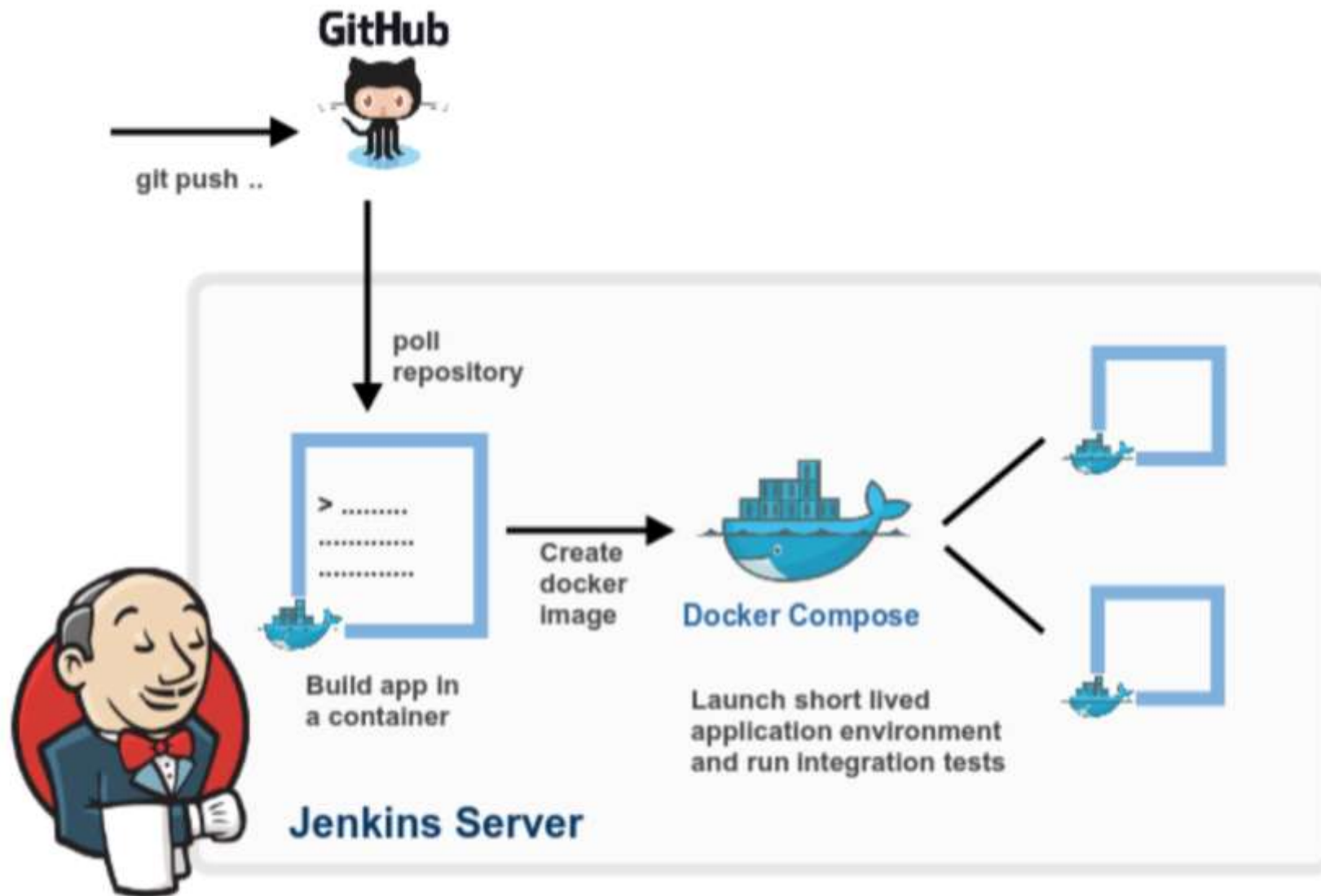
- Le processus de génération est donc désormais centralisé et automatisé et se déroule sur un serveur de build dédié.
- Tous les développeurs devraient avoir accès aux résultats de build, afin qu'ils puissent voir si leur code a bien été compilé, ou s'ils ont introduit une erreur de compilation qu'ils doivent corriger.
- Après chaque build, les résultats seront automatiquement publiés sur un portail par le serveur de build.
- Donc, si les développeurs de l'équipe font en moyenne, 20 commits de code en une journée, il y aura 20 builds qui seront déclenchés.
- Le tout géré automatiquement par le serveur de build.
- Dans le cas où une erreur de compilation a été introduite et la construction a échoué, un e-mail est immédiatement déclenché pour le développeur.
- Il s'agit du développeur qui vient de valider le code qui a provoqué l'erreur de compilation.



Tests automatisés

Continuous Build





Environnement
automatisé de
test de l'UI

CI attitude

Si une organisation adhère à ces principes l'organisation aurait adopté l'intégration continue.

Il y a une fausse idée selon laquelle l'intégration continue signifie simplement automatiser le processus de construction.

Ce n'est pas vrai.

C'est une nouvelle façon de travailler qui doit être adoptée par les développeurs.

L'automatisation est une étape habilitante de l'intégration continue.

Les principes à adopter pour réaliser la CI

Un référentiel central unique (SCM) où réside le code.

Les développeurs archivent leur code fréquemment, au moins une fois par jour.

A chaque « check in » le build doit être déclenché.

Le build doit être automatisée.

Le build doit être rapide et exécuté plusieurs fois et les développeurs obtiennent une rétroaction immédiate.

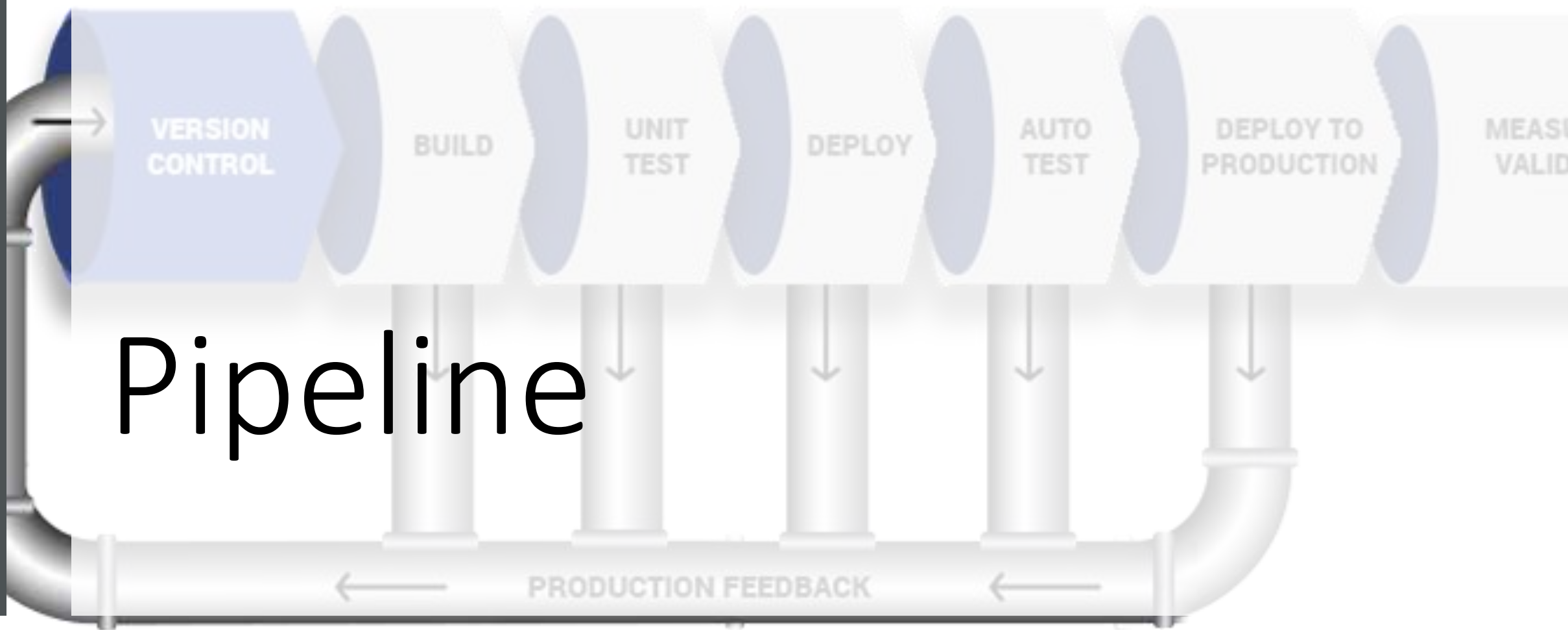
Le build doit compiler le code et exécuter des tests automatisés sur le code compilé.

La correction d'un build échoué devrait être la priorité absolue des développeurs concernés.

Les résultats du build et des tests automatisés, doivent toujours être visibles pour tous les développeurs.

Intégration Old School vs CI

- L'intégration consomme temps et d'efforts et est sujette aux erreurs.
- Dans CI, l'intégration est automatisée et rapide.
- Les problèmes apparaissent tard dans l'itération dans le modèle de la vieille école
- Dans CI, les problèmes apparaissent tôt, car nous intégrons fréquemment.
- Si une fusion échoue les développeurs sont bloqués par des versions cassées
- Dans CI, les builds cassés ont la priorité immédiate des développeurs.
- Long cycle de rétroaction pour connaître les défauts fonctionnels
- Dans CI, le cycle de rétroaction est plus court, , le développeur en est immédiatement informé
- Itérations longues.
- Dans CI, les itérations peuvent être très courtes, mise sur le marché plus rapide.



Pipeline



Chaine de montage auto sans pipeline



sans pipeline



sans pipeline



No of cars in 24 hours = $24 / 6 = 4$ cars



sans pipeline



Avec Pipeline



Avec Pipeline



No of cars in 24 hours = 10 cars

Build Pipeline



CODE INTEGRATION



COMPILATION



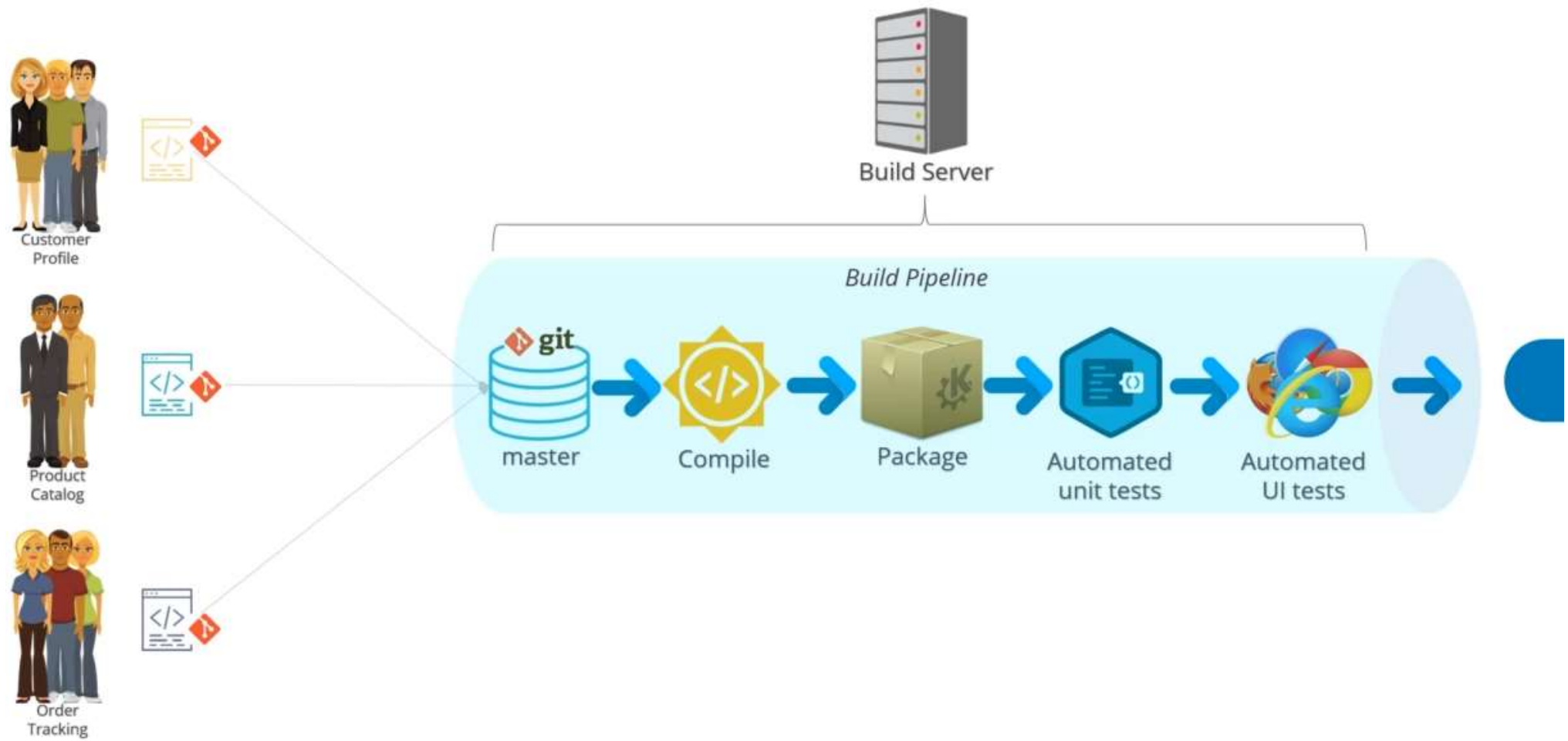
PACKAGING



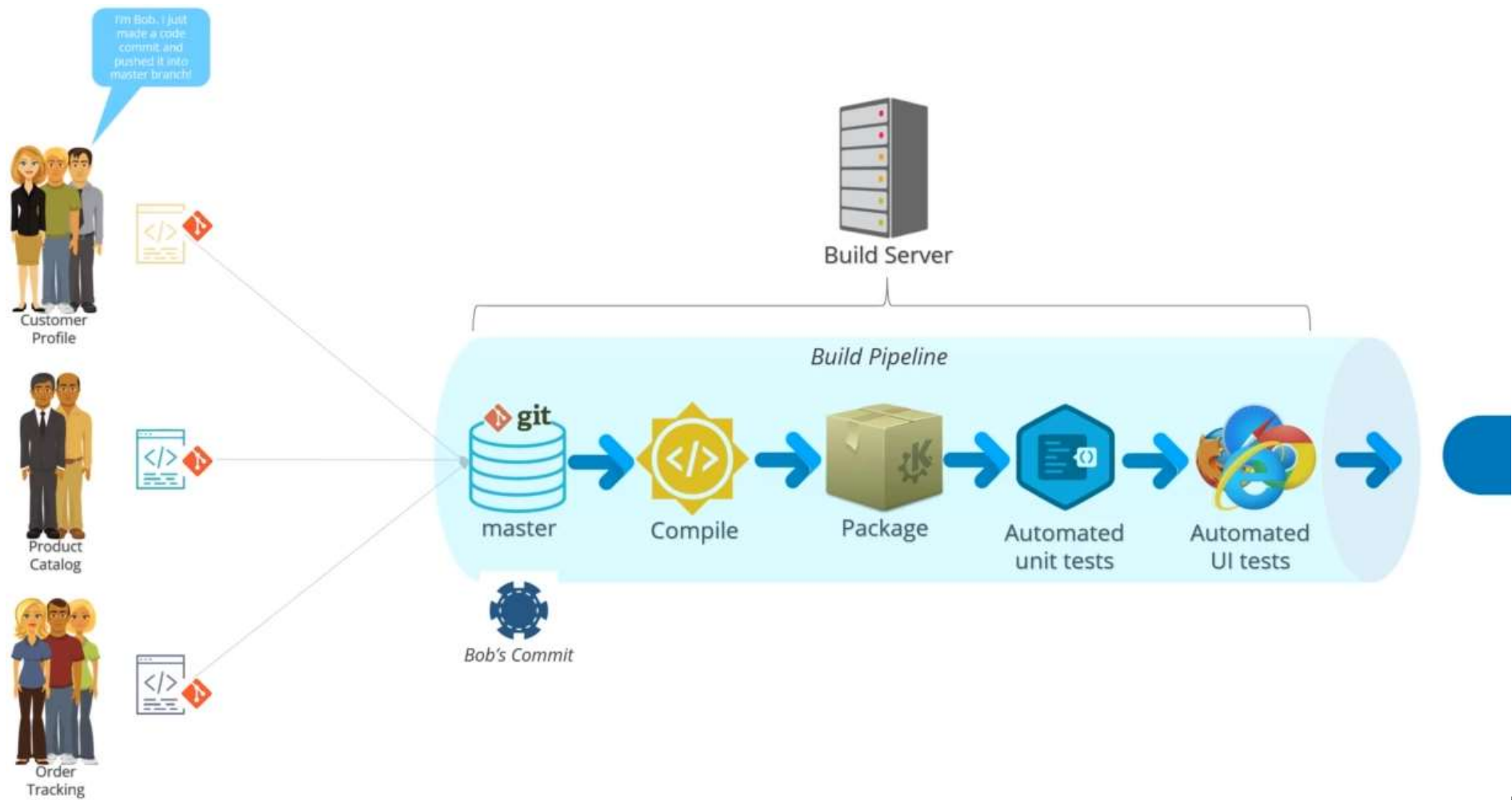
RUNNING
AUTOMATED UNIT
TESTS



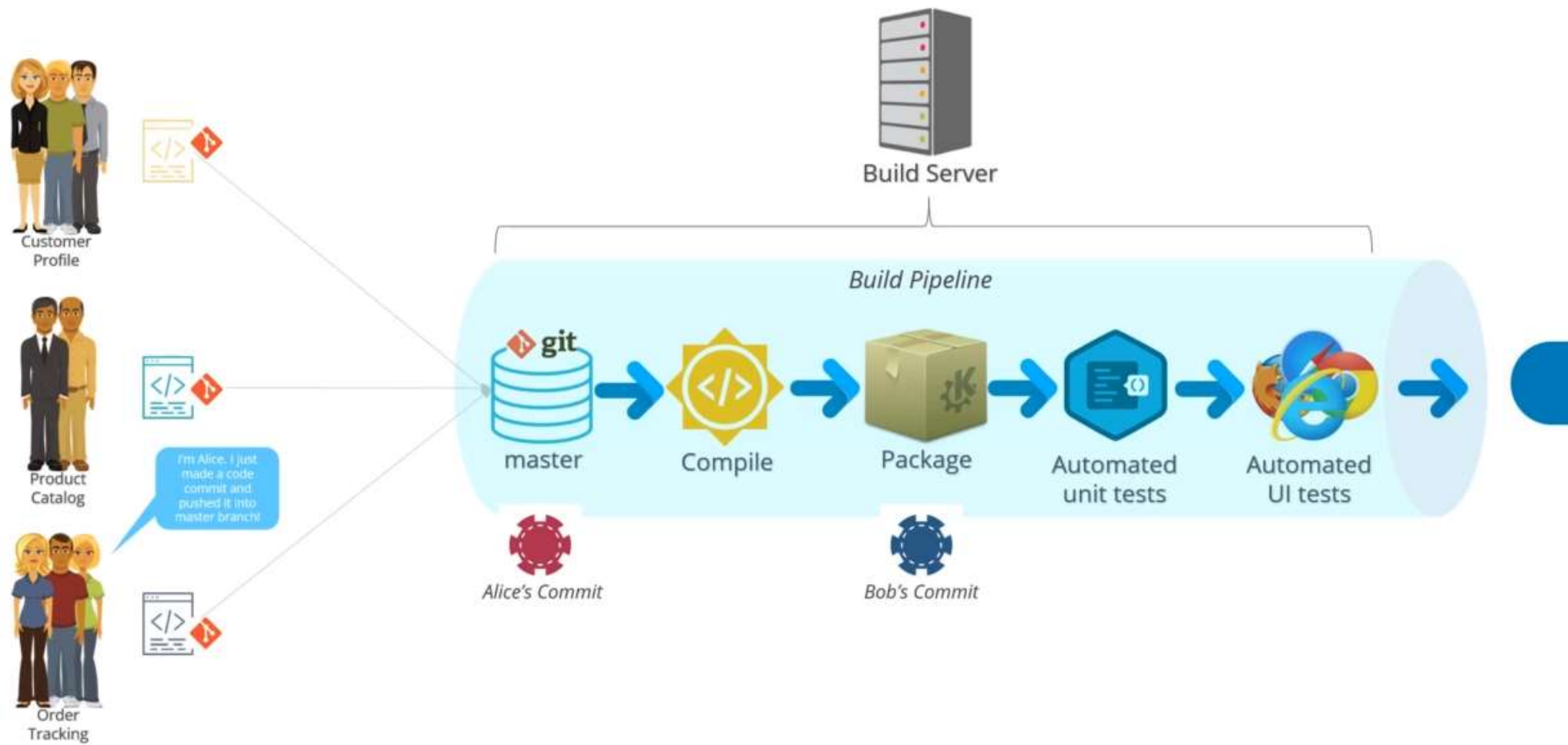
RUNNING
AUTOMATED UI
TESTS



Build Pipeline



Build Pipeline

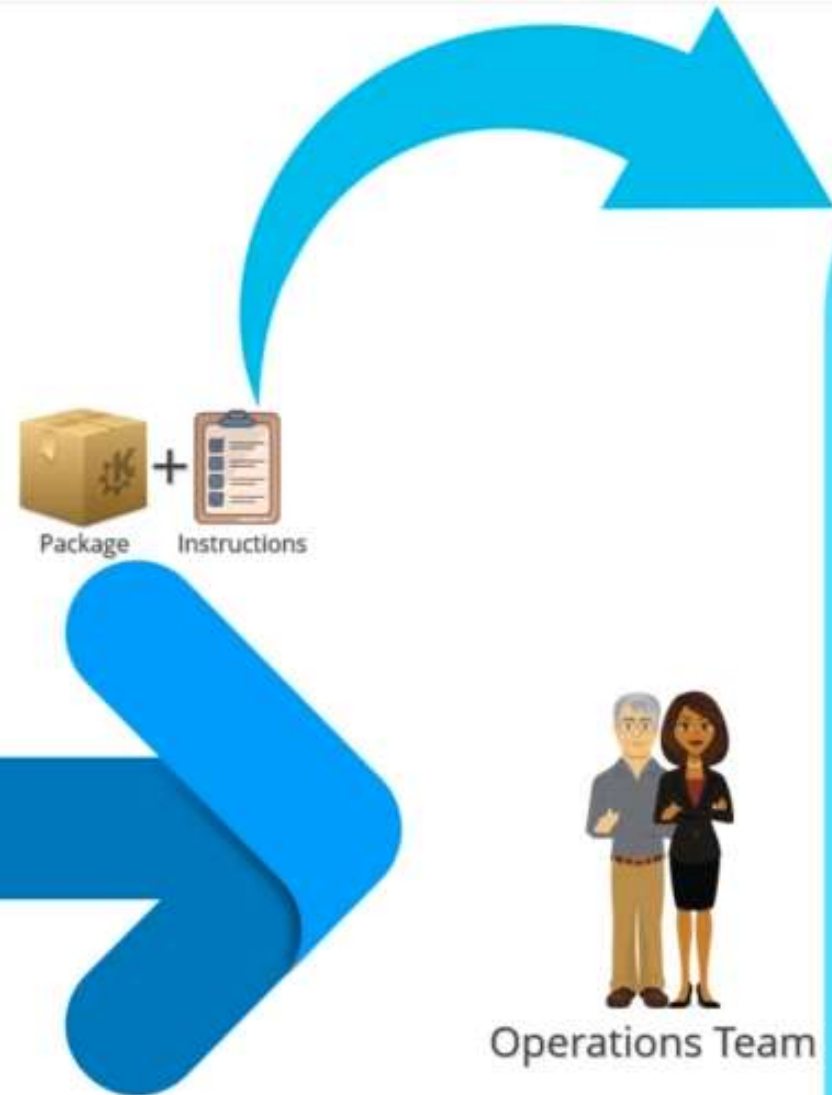


En résumé

- Le pipelining est efficace pour maximiser le débit lors de plusieurs builds consécutifs.
- Pour 10 ou 100 builds quotidiens, le pipelining est incontournable, plutôt qu'intéressante.
- Dans un pipeline de build CI, toutes les étapes doivent être automatisées.
- Il ne devrait y avoir aucune intervention humaine.
- Et toutes ces étapes devraient s'exécuter sur un serveur de build dédié.
- Il en sera de même pour le CD (livraison continue), où également on a un pipeline.
- Nous appellerons cela un pipeline de release.



Old School Operations |



Operations Team

Instructions for Operations Team

1. Copy a set of config files(files ending in .properties) from the root of the package, to the folder /etc/config on the server.
2. Download and install the latest software patches from the website <http://xxx/yyy/patches> on the Operating System.
3. Setup environment variables on the OS. If you are deploying to the test environment, use the first set of values attached. If you are deploying to production, use the second set of values attached.

Test

env=test
db=mockdb
url=test.mikeselectronics.com

Production

env=prod
db=productdb
url=mikeselectronics.com

Regards,
The Development Team

Elies Jebri

Instructions pour
la release

Environnement de Test



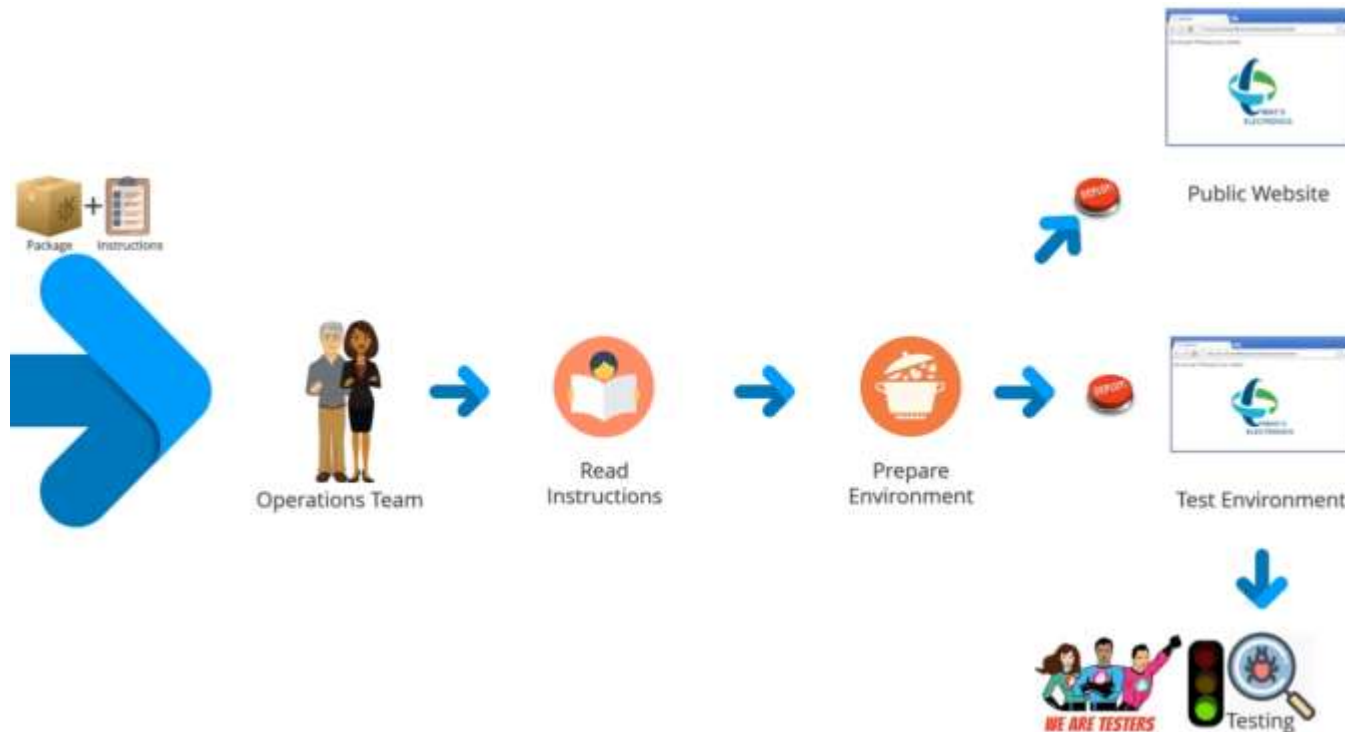
Après avoir lu les instructions, préparer l'environnement de test selon les instructions.

Lorsque l'environnement TEST est prêt, déployer le package sur TEST.

Maintenant que l'équipe d'assurance qualité entre en jeu, elle effectue une série de tests sur l'application déployée sur l'environnement TEST.

S'ils découvrent des défauts, la rétroaction revient à l'équipe de développement et le cycle doit se répéter.

Environnement de déploiement



S'ils ne trouvent aucun défaut, ils certifient la version, puis l'équipe des opérations préparera maintenant l'environnement de production conformément aux instructions.

Maintenant que l'environnement PROD est prêt, déployez le package dans Production.

À ce stade, les nouvelles fonctionnalités sont disponibles pour tous les utilisateurs du site Web. Et cela marque la fin de notre cycle de développement logiciel.

Problèmes de l'ancienne méthode



L'(in)exactitude des instructions.



La différence d'instructions en fonction des env (test/prod).



Les instructions exécutées manuellement par un opérateur.



Un seul déploiement peut prendre de quelques heures à une demi-journée.

Continuous Delivery

Définition

La livraison continue est une pratique de développement de logiciels où les logiciels peuvent être mis en production à tout moment.

Une condition préalable importante pour la livraison continue est l'intégration continue.



Instructions for Operations Team

1. Copy a set of config files(files ending in .properties) from the root of the package, to the folder /etc/config on the server.
2. Download and install the latest software patches from the website <http://xxx/yyy/patches> on the Operating System.
3. Setup environment variables on the OS. If you are deploying to the test environment, use the first set of values attached. If you are deploying to production, use the second set of values attached.

Test

env=test
db=mockdb
url=test.mikeselectronics.com

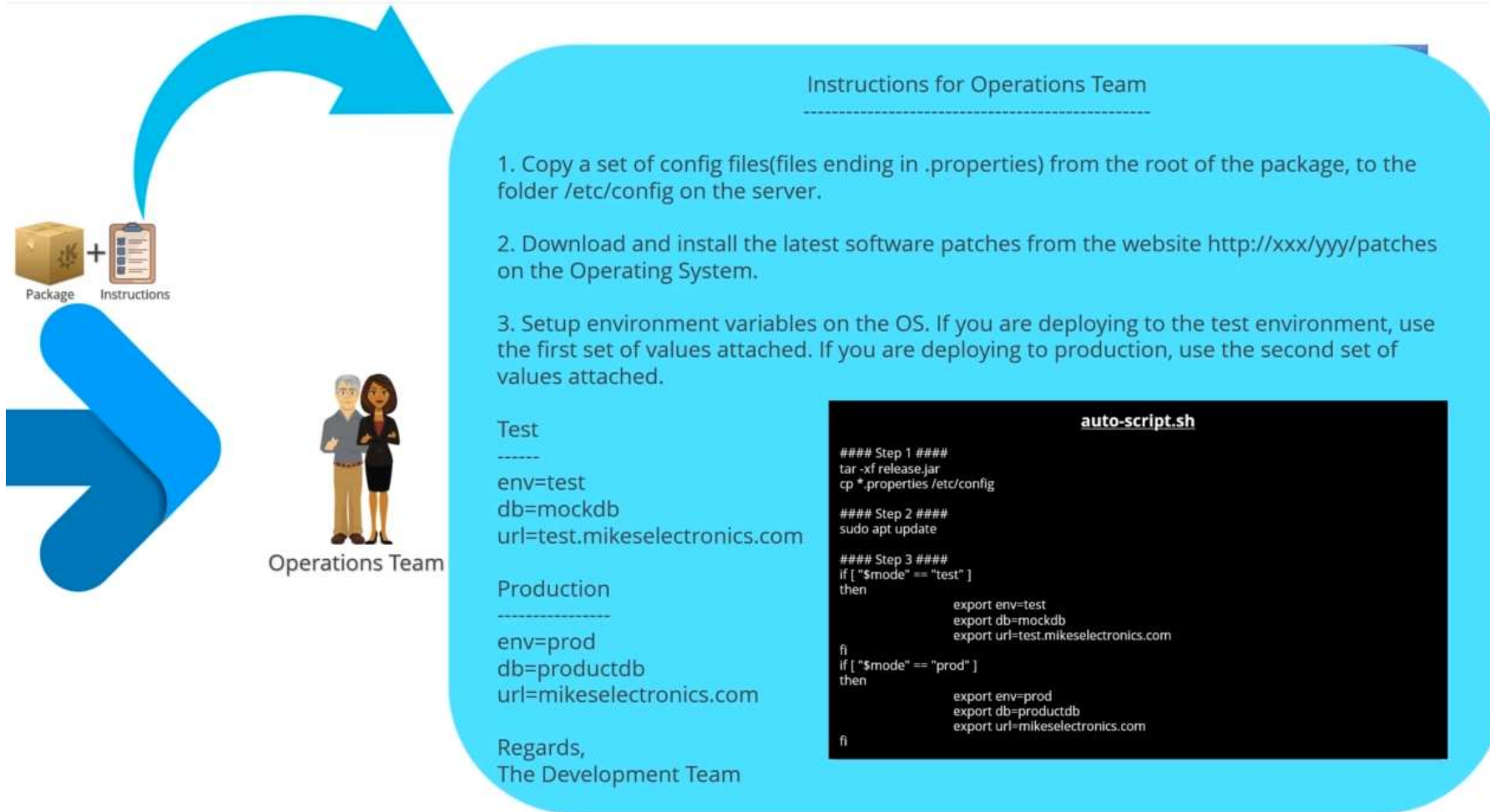
Production

env=prod
db=productdb
url=mikeselectronics.com

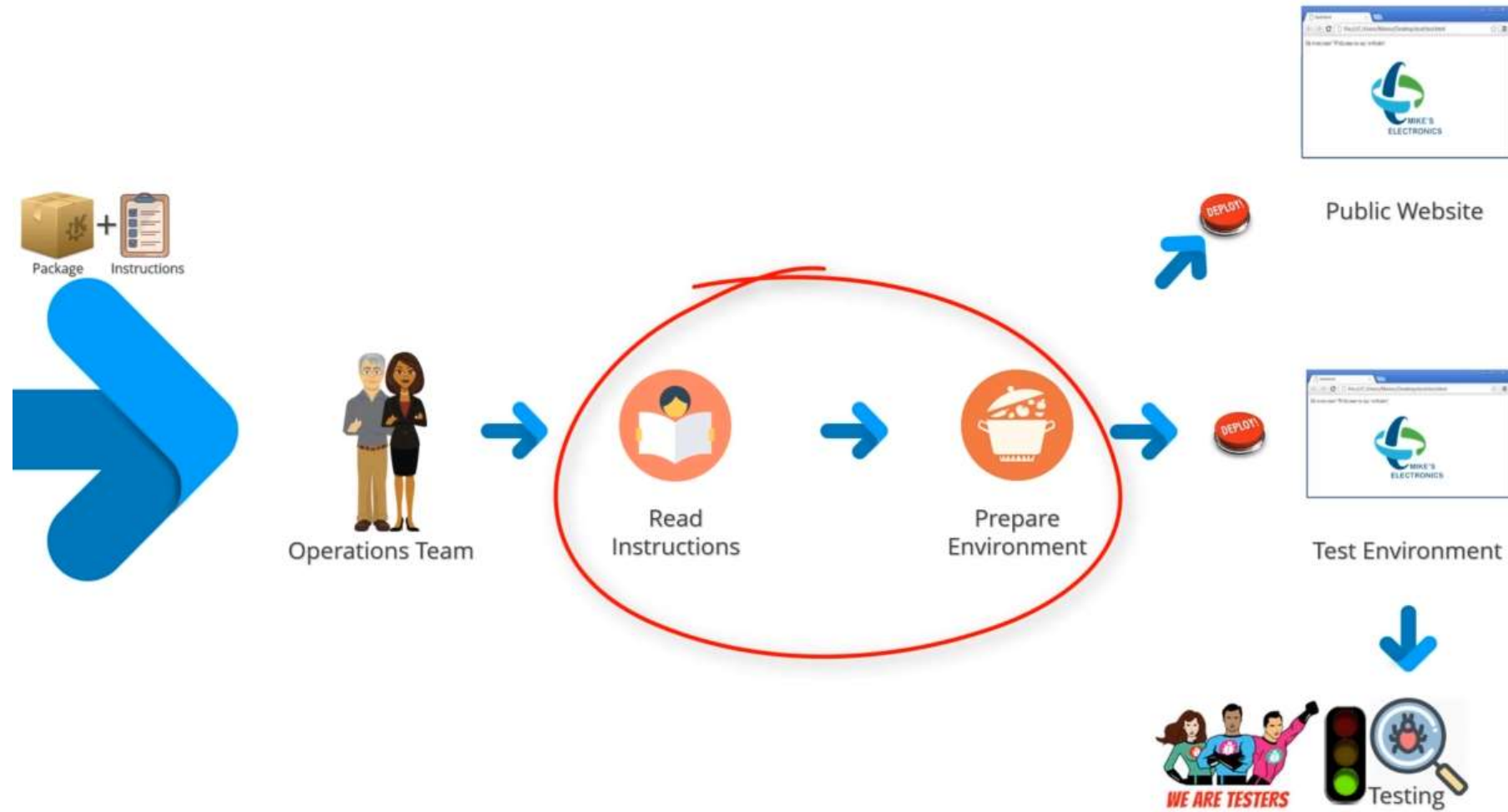
Regards,
The Development Team

Instructions pour
la release

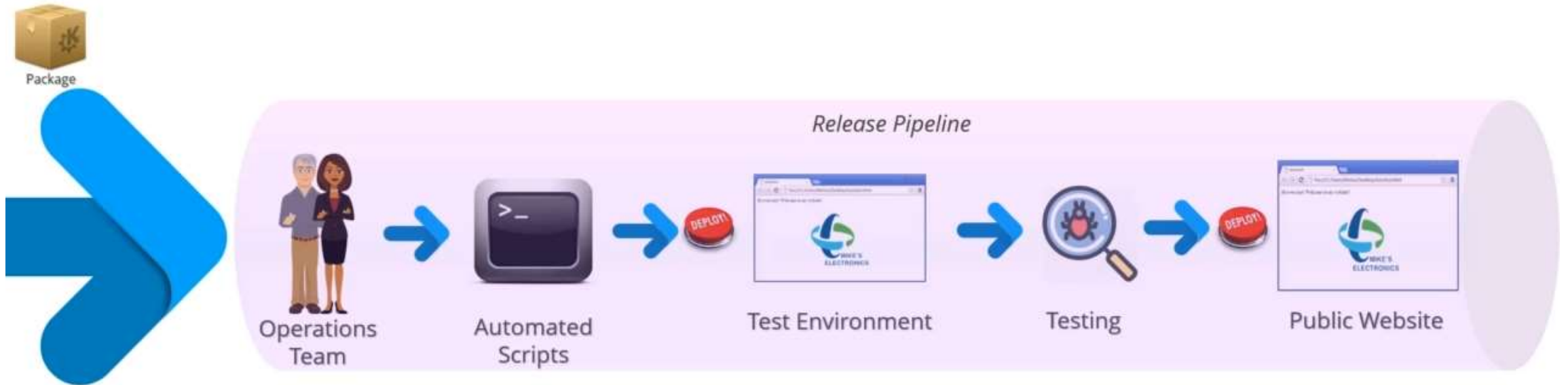
Automatisation des instructions



Éléments à automatiser



Release Pipeline



La plupart des tâches à l'intérieur de ce pipeline sont automatisées.

Mais notez que nous avons encore **une étape manuelle** où l'équipe d'assurance qualité effectue des vérifications sur l'application Web déployée sur l'environnement TEST.

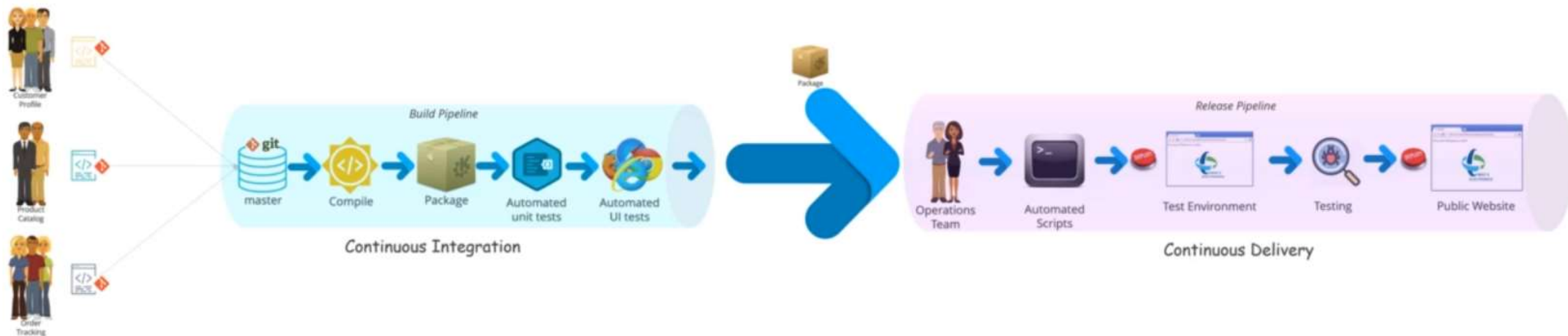
Continuous Delivery

- La livraison continue est une pratique de développement de logiciels où les logiciels **peuvent** être mis en production à tout moment.
- Insistons sur l'expression "le logiciel **PEUT** être publié".
- On ne dit pas "le logiciel est automatiquement publié".
- Donc, en livraison continue, nous sommes dans un état à partir duquel nous pouvons toujours déployer en production, **si nous le souhaitons**.
- L '«intégration» continue, que nous avons vue, se produit de manière complètement automatisée, dans laquelle si quelqu'un fait un commit dans une branche principale, la construction sera déclenchée automatiquement et le package de version sera généré en quelques minutes.
- Mais contrairement à l'intégration continue, la livraison continue n'a pas besoin d'être entièrement automatisée.

En résumé

- Dans le vieux modèle, le processus de release n'était pas rapide.
- Cela impliquait de nombreuses étapes manuelles et des temps d'arrêt des applications.
- Mais maintenant, avec la livraison continue, même si nous avons besoin d'une intervention manuelle, la plupart des étapes intermédiaires sont automatisées.
- Il est donc rapide et efficace et également reproductible, et les temps d'arrêt sont très réduits.

Maturation en déploiement continu



Cycle de vie complet du logiciel compatible CD CI

Derniers obstacles

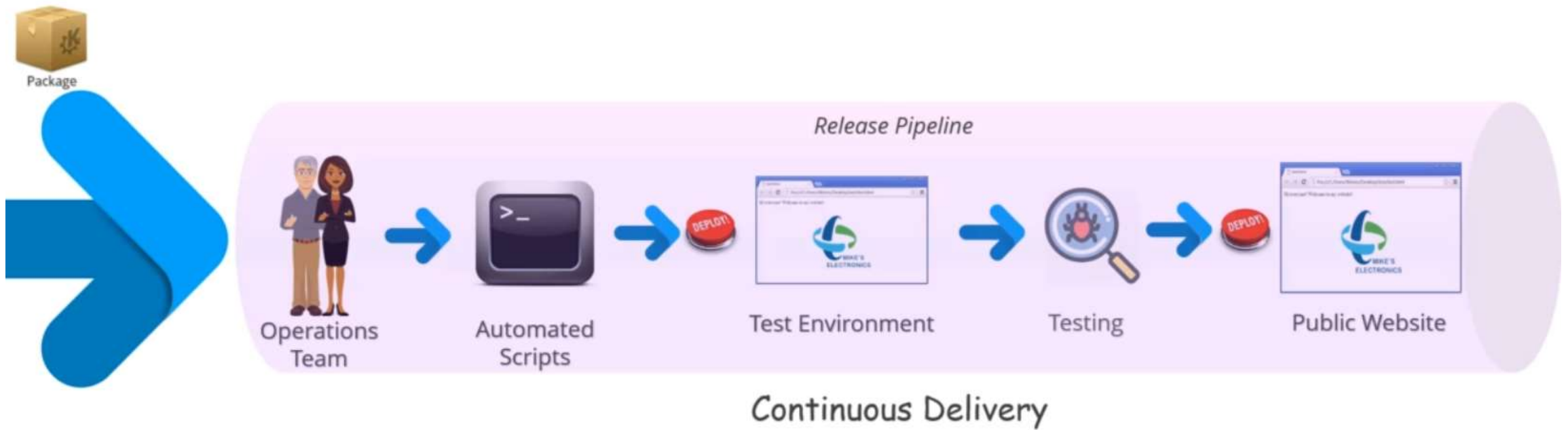
Certaines étapes intermédiaires sont toujours manuelles.

Par exemple en livraison continue, les tests manuels et la certification par l'équipe d'assurance qualité.

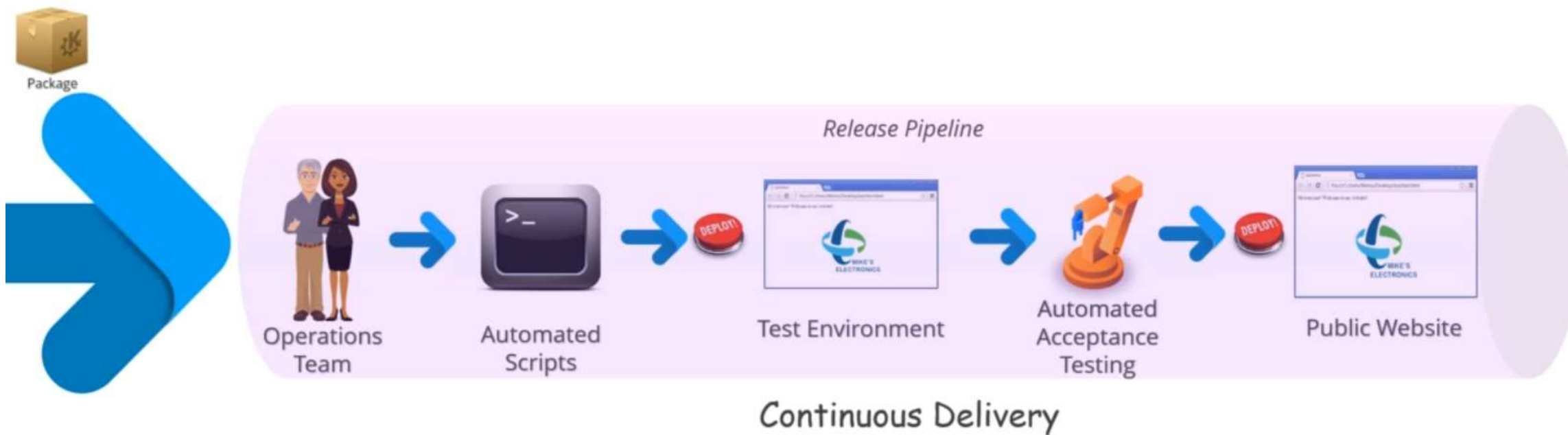
L'équipe d'assurance qualité effectue toujours des tests d'acceptation sur l'application manuellement avant de la déployer en production.

Et si on automatisait tout?

Les tests d'acceptation peuvent-ils également être automatisés?

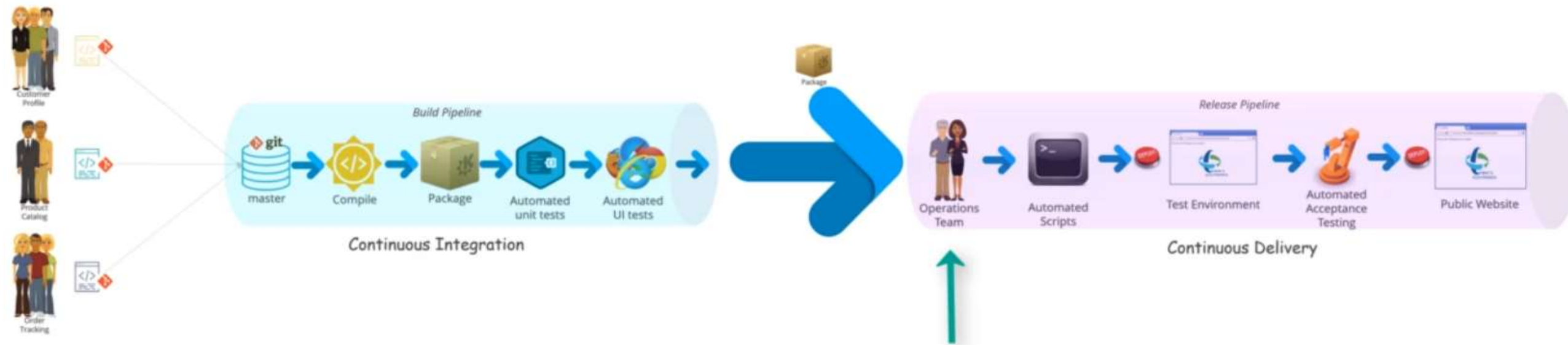


Tests d'acceptance manuels

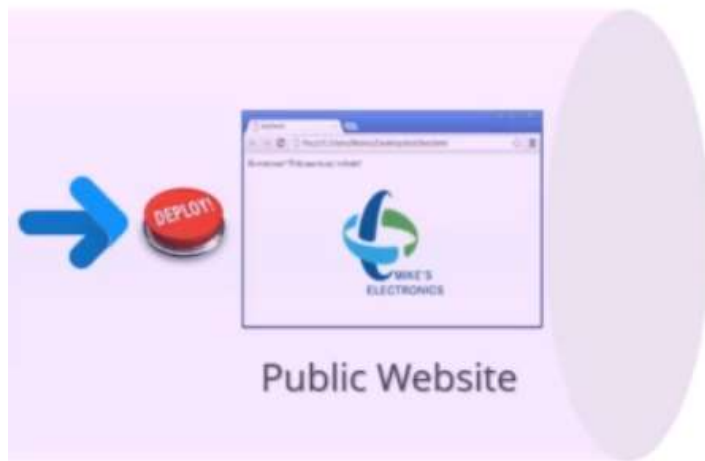


Tests d'acceptance automatisés

Plus d'intervention manuelle



L'équipe opérations maintient simplement les scripts d'automatisation. Ils n'interviennent pas dans le processus de déploiement.



Déploiement Continu



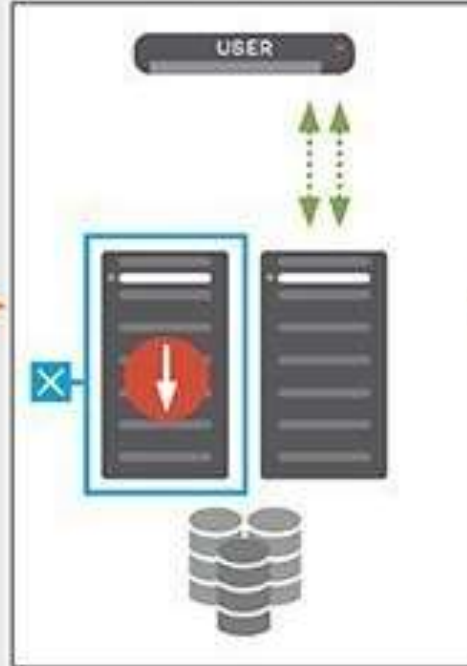
Techniques de déploiement

ROLLING DEPLOYMENT

BOTH NODES RUNNING ...



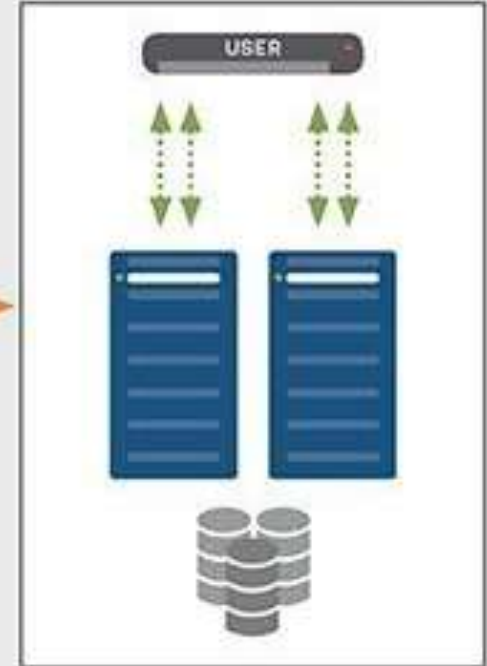
PATCHING 1ST NODE ...



PATCHING 2ND NODE ...



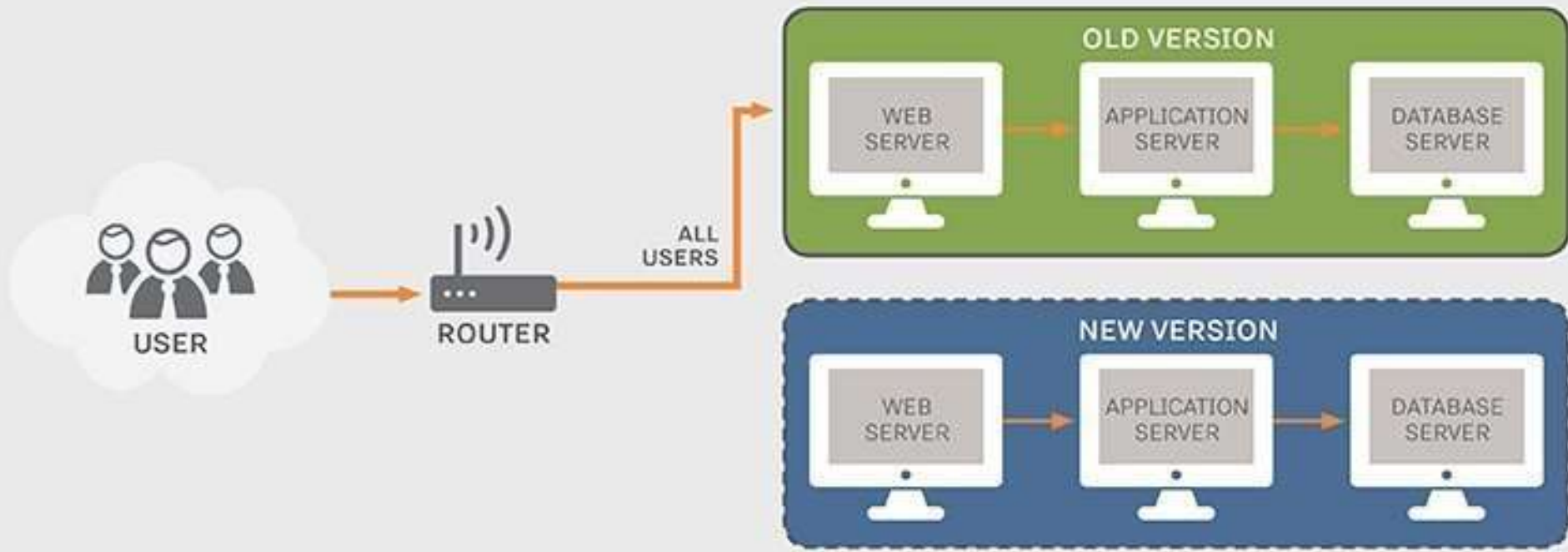
BOTH NODES RUNNING ...



Déploiement progressif

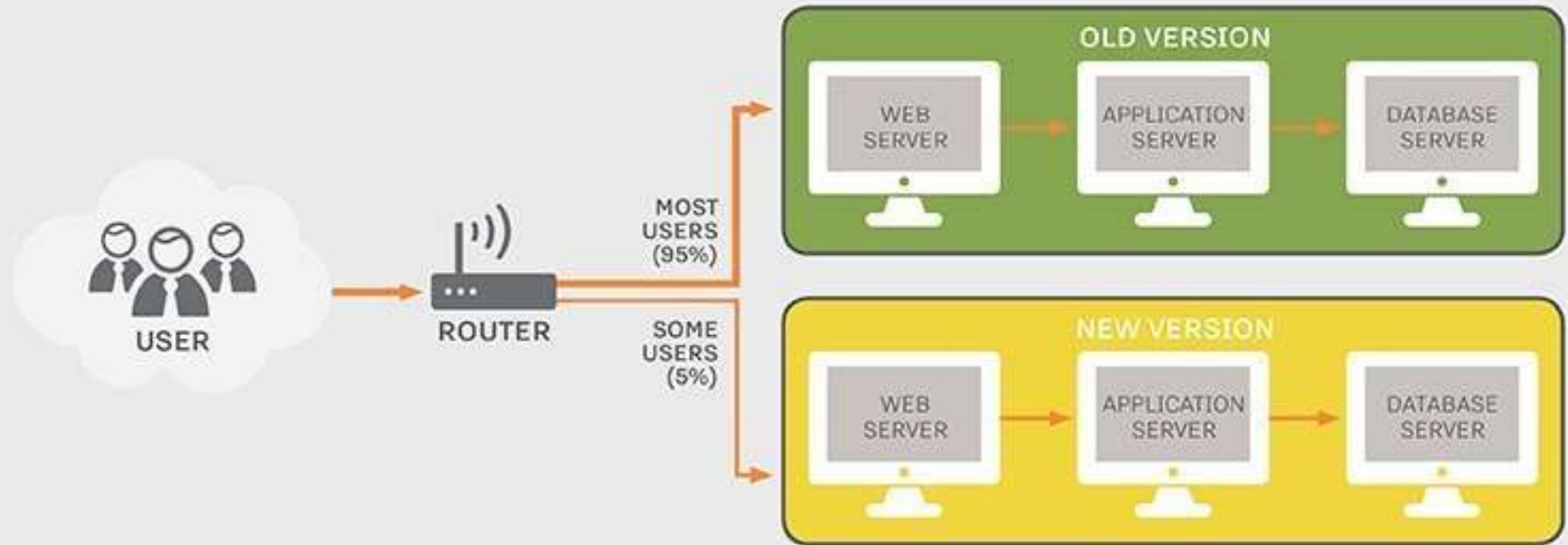
Déploiement blue/green

BLUE/GREEN DEPLOYMENT



Déploiement canary

150



Continuous Delivery & Continuous Deployment



La livraison continue est une pratique de développement de logiciels où les logiciels peuvent être mis en production à tout moment.



C'est un choix.



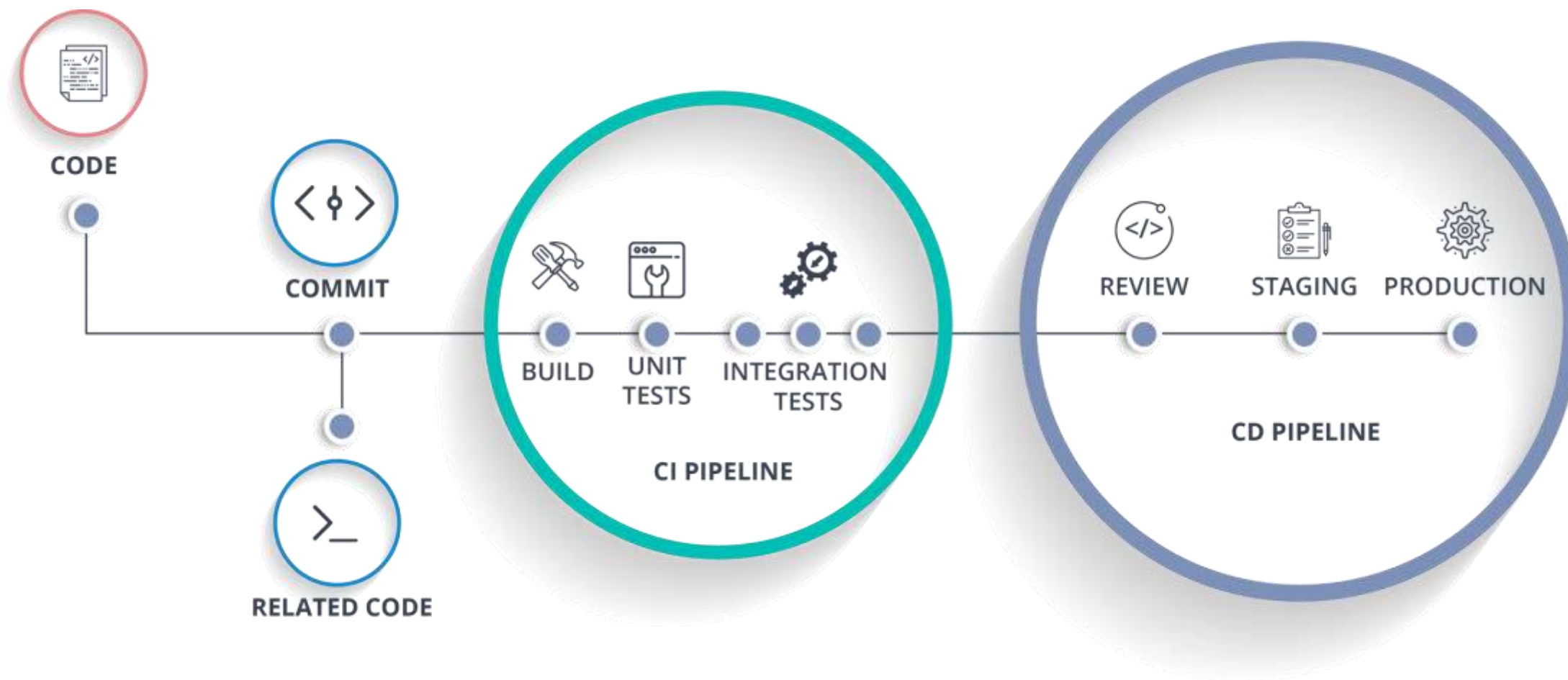
Le «déploiement» continu est une pratique de développement logiciel où le code est automatiquement déployé en production tout le temps, en continu.

Continuous Deployment est rare

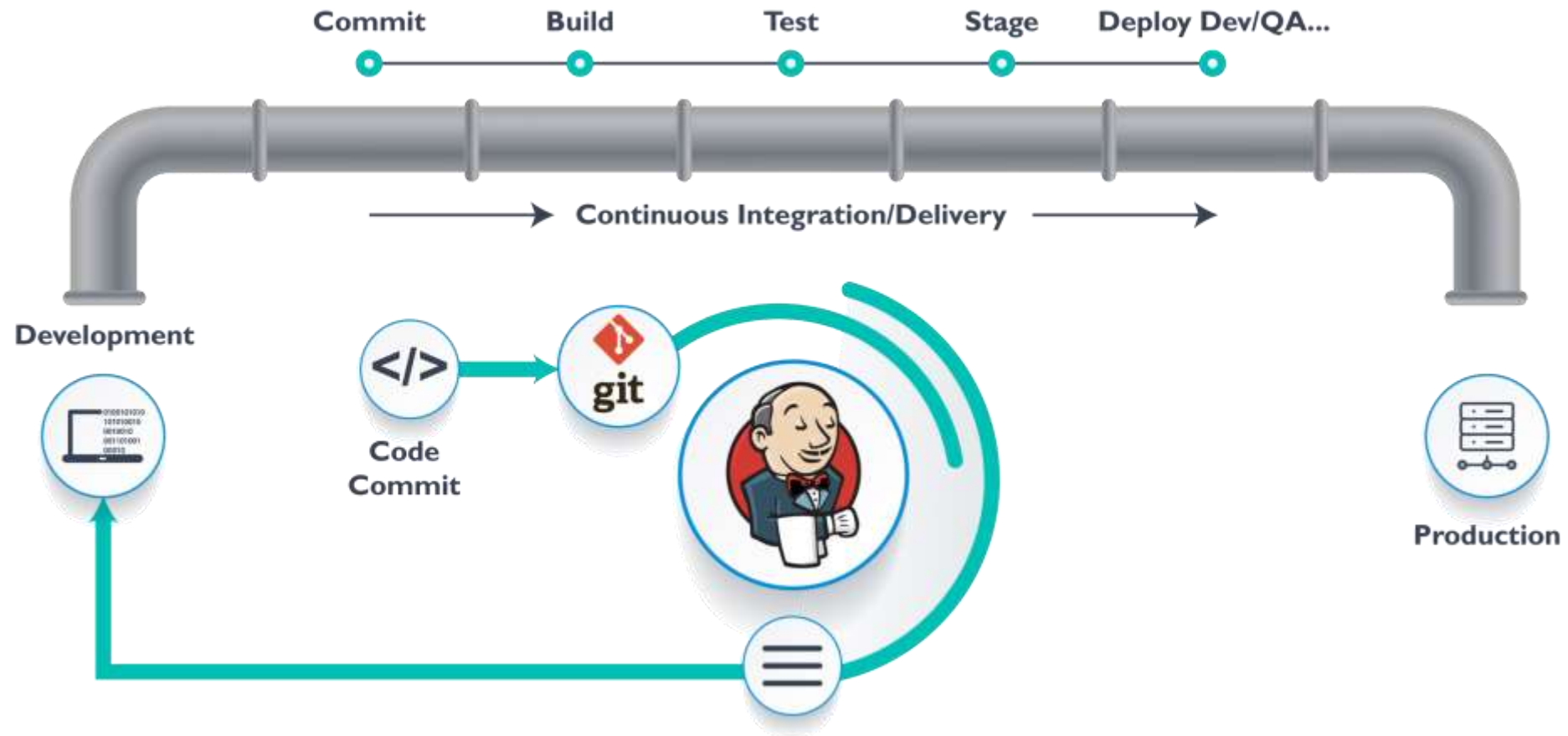
- Seules les entreprises dont le département informatique a atteint un niveau de maturité très élevé, opteront pour le déploiement continu, en raison des risques encourus.



CI/CD



Automatisation CI/CD



La pile Technologique du DevOps

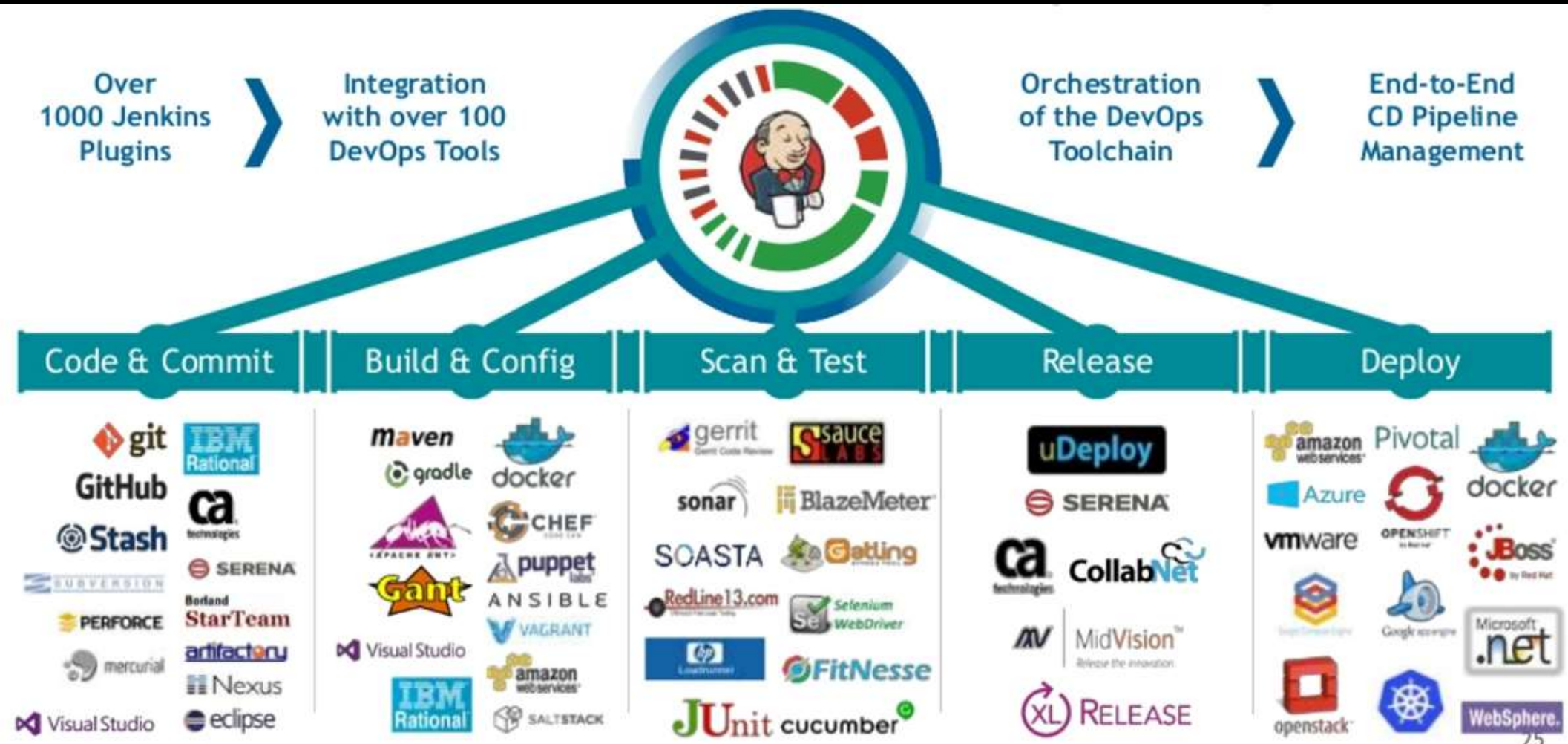


Tableau périodique du DevOps



| | | | | | | | | | |
|-----------|-------------|-----------|------|-----------|----------|-----------|------|-----------|------------|
| Os | Open-source | Fr | Free | Fm | Freemium | Pd | Paid | En | Enterprise |
|-----------|-------------|-----------|------|-----------|----------|-----------|------|-----------|------------|

