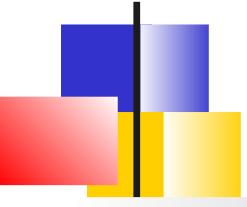


DevOps

Comprendre les outils utilisés par tous les acteurs d'un projet Agile

David THIBAU - 2021

david.thibau@gmail.com



Agenda

Introduction

- Les contraintes de l'agilité
- Le constat DevOps
- CI et CD
- Cycle de vie du code et outils DevOps
- DevOps et Architecture des systèmes

Méthodologies et outils de pilotage

- Rappels sur l'agilité
- Les outils de gestion d'issues

Gestion des sources

- L'unique source de vérité
- Workflows de collaboration

Outils de build

- Caractéristiques d'un outil de build
- Tests et analyse qualité
- Relase et dépôts d'artefacts

Plateforme CI/CD

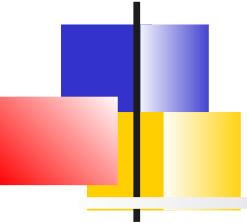
- Concepts communs
- Pipelines typiques
- Gitlab CI / Jenkins

Déploiement

- Considérations
- Virtualisation
- Gestion de configuration. Le cas Ansible
- Containerisation. Le cas docker
- Orchestrateur de conteneur : Kubernetes
- Kubernetes dans la pipeline CD

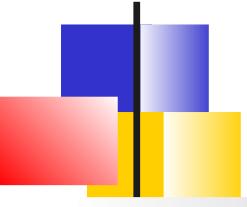


Contraintes de l'agilité



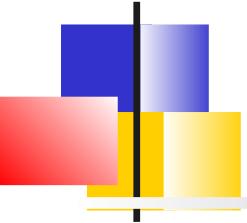
L'agilité

- Le terme agile (regroupant de nombreuses méthodes) est consacré par le manifeste Agile : <http://agilemanifesto.org/> 2001
 - Personnes et interactions plutôt que processus et outils
 - Logiciel fonctionnel plutôt que documentation complète
 - Collaboration avec le client plutôt que négociation de contrat
 - Réagir au changement plutôt que suivre un plan



Les 12 principes

1. Satisfaction du client **en livrant rapidement et régulièrement** des fonctionnalités à grande valeur ajoutée.
2. Accepter les changements de besoins, même tard dans le projet.
3. **Livrez fréquemment** un logiciel opérationnel.
4. Utilisateurs et développeurs travaillent **ensemble** quotidiennement.
5. Les personnes sont motivées et **bénéficient d'un environnement et d'un soutien** à la hauteur
6. Transmission de l'information via le dialogue en face à face.
7. L'avancement est mesuré via le **logiciel opérationnel**
8. Rythme de développement soutenable.
9. Excellence technique et bonne conception renforce l'agilité.
10. La simplicité - c'est-à-dire l'art de minimiser la quantité de travail inutile - est essentielle.
11. Les équipes sont auto-organisées.
12. À intervalles réguliers, l'équipe cherche à s'améliorer.



Méthodes agiles

2 facteurs communs à toutes les méthodes agiles :

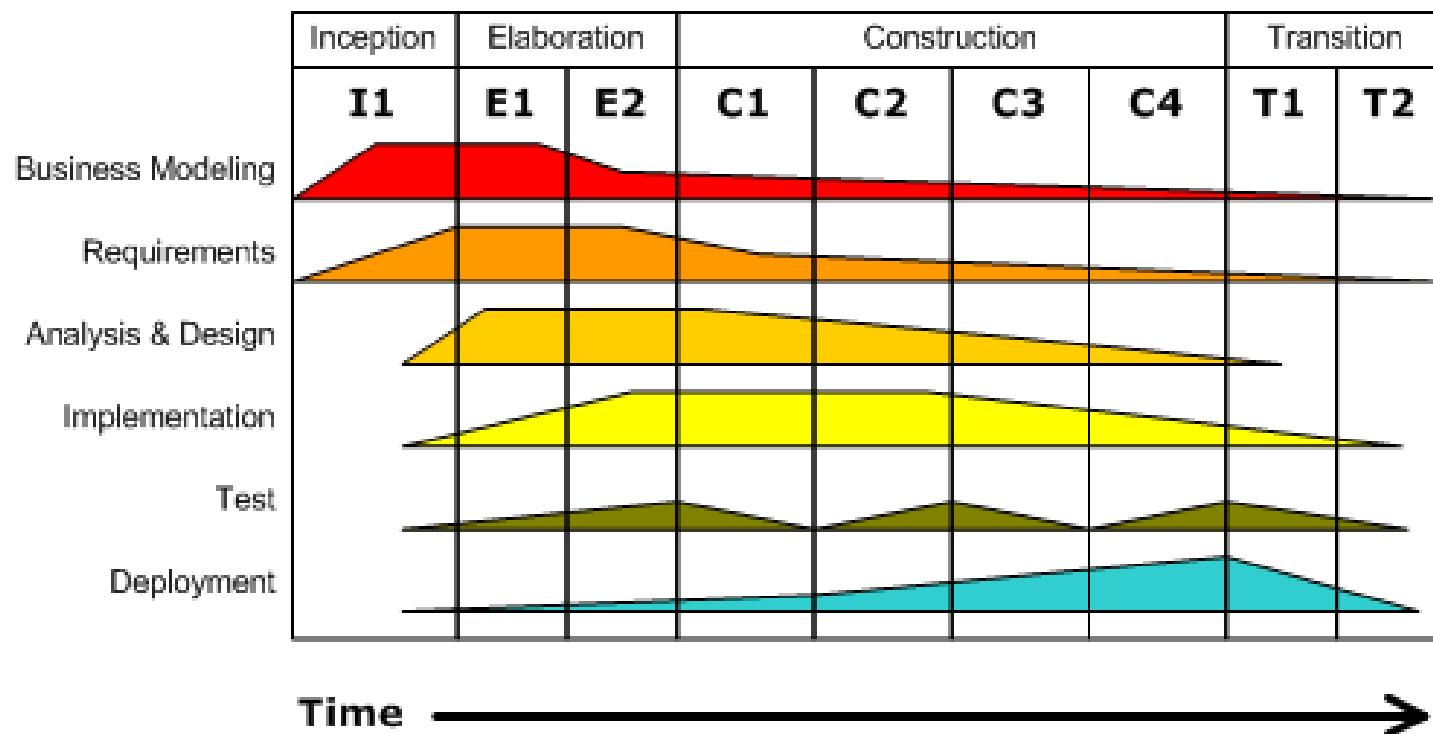
- la mise en place de pratiques **itératives**,
- des projets plus **petits**
=> des équipes de dév. de plus en plus petites, des périmètres fonctionnels limités

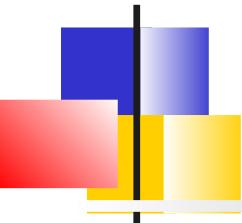
→ C'est ce l'on retrouve dans *RUP, XP Programming, Scrum*

Unified Process

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.

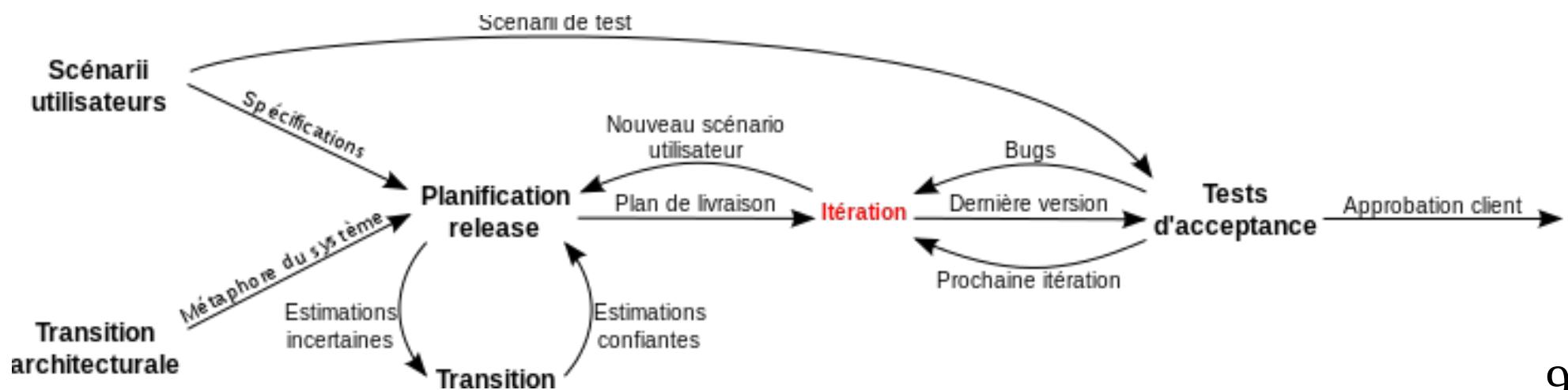




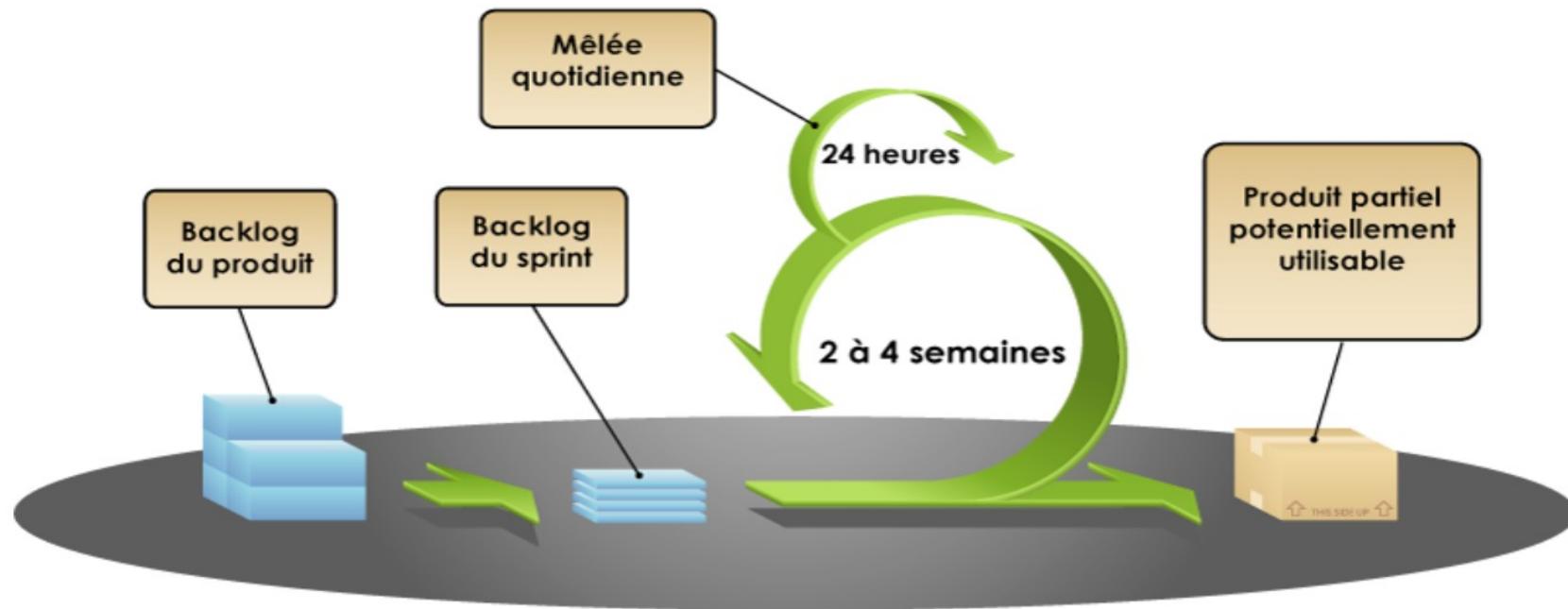
XP: eXtreme Programming

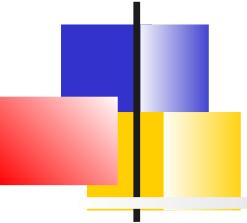
Agilité, Petites équipes, Tout à l'extrême :

- Revue de code en permanence (par un binôme)
- Tests systématiques
- Refactoring en continu
- Toujours la solution la plus simple ;
- Evolution des métaphores ;
- Intégration continue ;
- Cycles de développement très rapides pour s'adapter au changement.



Scrum : rythme





Contraintes sur les déploiements

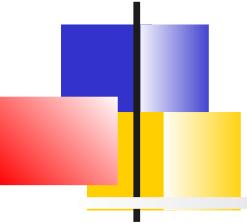
L'agilité suppose d'augmenter la fréquence des déploiements dans les différents environnements : intégration, recette, production afin :

- De mieux piloter le projet
- De prendre en compte rapidement les retours utilisateurs

Problème : Avant DevOps, l'organisation des services informatiques ne facilitait pas les déploiements



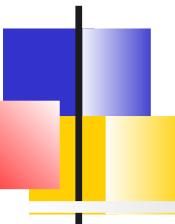
Le constat DevOps



Cycle de vie d'un logiciel

Le cycle de production d'un logiciel passe par plusieurs étapes correspondant à plusieurs environnements :

- **Développement** : Poste du développeur
- **Intégration** : Intégration des modifications de toute l'équipe
- **QA** : Environnement proche de la production permettant la qualification des releases
- **Production** : Exploitation, Support, Maintenance

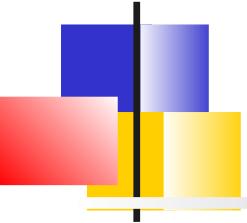


Disparité des environnements

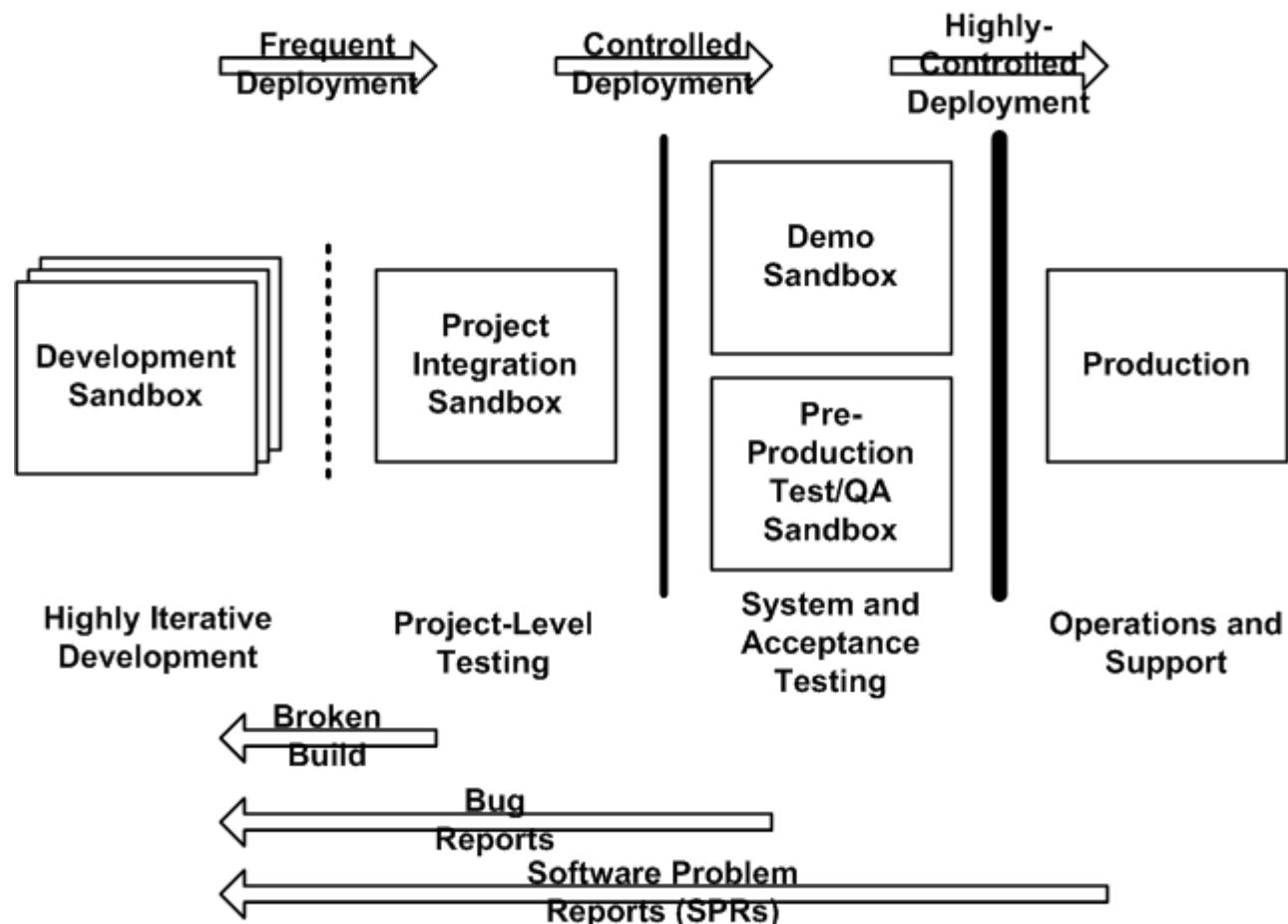
Les environnements ne peuvent que différer :

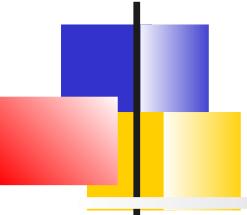
- **Développement** : IDE, Code source lisible permettant le debug, Configuration serveur pour des déploiements à chaud, base de données simplifiée, ...
- **Intégration** : Configuration pour les tests d'intégration. Sondes, Niveau de trace, Simulation des charges, des données
- **QA** : Le plus proche de la production mais pas les mêmes données, pas les mêmes charge.
- **Production** : Qualité de service, charge réelle, données de production, utilisateurs finaux

DevOps : S'entraîner continuellement à déployer dans ses différentes conditions



Avant DevOps : Fréquence de déploiement



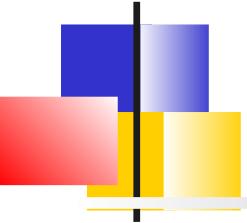


Disparité des objectifs

Ces environnements sont traditionnellement gérés et utilisés par des équipes distinctes qui ... souvent communiquent peu

Les équipes ont de plus des objectifs différents

- Développeur : Implémenter les fonctionnalités requises dans le temps imparti.
- Intégrateur : Dimensionner l'architecture pour atteindre des SLA
- QA : Valider fonctionnellement le système dans des scénarios pas toujours anticipés par les dev.
- Opérations : Garantir la stabilité du système et des infra-structures

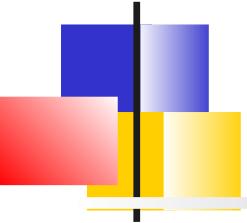


Le constat DevOps

Les différents objectifs donnés à des équipes qui se parlent peu créent des tensions et des dysfonctionnements dans le processus de mise en production d'un logiciel.

=> Pour l'équipe Ops, l'équipe de développement devient responsable des problèmes de qualité du code et des incidents survenus en production.

=> L'équipe Dev blâme son alter ego Ops pour sa lenteur, les retards et leur méconnaissance des livrables qu'elle manipule

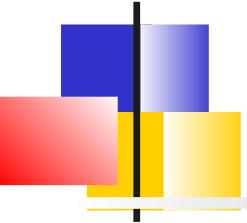


Approche *DevOps*

DevOps vise l'alignement des équipes par la réunion des "Dev engineers" et des "Ops engineers" chargés d'exploiter les applications existantes au sein d'une même équipe.

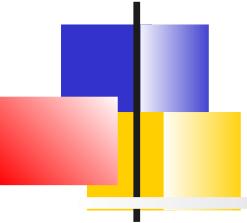
Cela impose :

- la réunion des équipes
- la montée en compétence des différents profils.



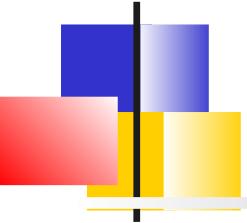
Pratiques *DevOps* (1)

- Un déploiement régulier des applications dans les différents environnements.
La seule répétition contribuant à fiabiliser le processus ;
- Un décalage des tests "vers la gauche".
Autrement dit de tester au plus tôt ;
- Une pratique des tests dans un environnement similaire à celui de production ;
- Une intégration continue incluant des "tests continus" ;



Pratiques *DevOps* (2)

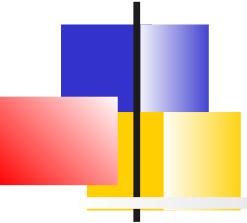
- Une boucle d'amélioration courte i.e. un feed-back rapide des utilisateurs ;
- Une surveillance étroite de l'exploitation et de la qualité de production factualisée par des métriques et indicateurs "clé".
- Les configurations des différents environnements, des builds, des tests centralisées dans le même SCM que le code source



« As Code »

Les outils *DevOps* permettent d'automatiser/exécuter la construction/fourniture des ressources à partir de l'unique point central de vérité

On parle de *Build As Code*, *Infrastructure As Code*, *Pipeline As Code*, *Load Test As Code*, ...

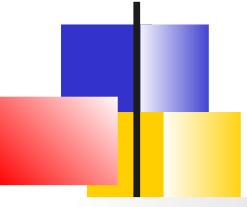


Objectif ultime

- Déployer souvent et rapidement
- Automatisation complète
- Zero-downtime
- Possibilité d'effectuer des roll-backs
- Fiabilité constante de tous les environnements
- Possibilité de scaler sans effort
- Créer des systèmes auto-correctifs, capable de se reprendre en cas de défaillance ou erreurs



CI/CD



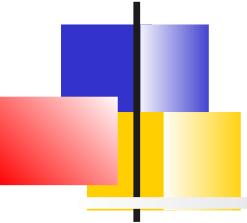
Avant l'intégration continue

Le cycle de développement classique intégrait une **phase d'intégration** avant de produire une release :

intégrer les développements des différentes équipes sur une plate forme ressemblante à la production.

Différents types de problèmes pouvaient survenir nécessitant parfois des réécritures de lignes de code et introduire des délais dans la livraison

=> L'intégration continue a pour but de lisser l'intégration **pendant** tout le cycle de développement

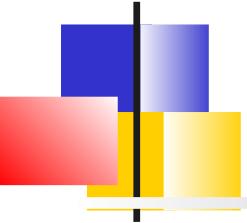


Plateforme d'intégration continue (PIC)

L'intégration continue dans sa forme la plus simple consiste en un outil surveillant les changements dans le **Source Control Management (SCM)**

Lorsqu'un changement est détecté, l'outil construit, teste automatiquement et déploie l'application dans l'environnement d'intégration

Si ce traitement échoue, l'outil notifie immédiatement les développeurs afin qu'ils **corrigeant le problème ASAP**



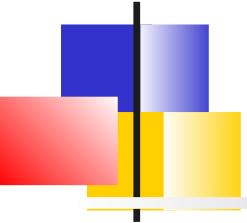
Build is tests !

La construction de l'application consiste à principalement à :

- Packager le code source dans un format exécutable qui peut être automatiquement déployé
- Exécuter toutes les vérifications automatique permettant d'avoir confiance dans l'artefact généré

L'activité de build intègre alors tous les types de tests automatisés que peut subir un logiciel (unitaires, intégration, fonctionnel, performance, analyse qualité, ...)

En fonction du résultat des tests et de la confiance qu'on leur accorde, chaque itération de création de valeur logicielle pourra être poussée dans un des environnements (intégration, QA, production)



Outil de communication et de motivation

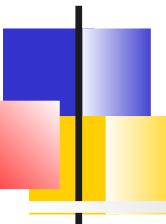
La PIC permet également de publier les résultats des builds (les résultats des tests et analyse):

- Nombre de tests exécutés, succès/échecs
- Couverture des tests
- Métriques Qualité
- Performance : Temps de réponse/débit
- Documentation produit du code source
-

=> Donne de la confiance dans la robustesse du code développé, réduction des coûts de maintenance.

=> Métriques qualité visibles aussi bien par les fonctionnels que par les développeurs

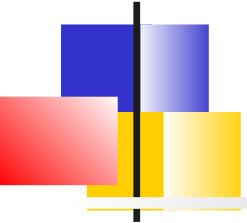
=> Cette transparence motive les équipes pour produire un code de qualité



Intégration continue et agilité

L'**intégration continue** met automatiquement à disposition sur une plateforme d'intégration l'application en cours de développement

- Dans les méthodes agiles, c'est une nécessité. Les fonctionnels et les développeurs peuvent alors arbitrer les choix fonctionnels en se basant sur du concret.



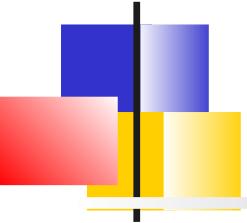
Livraison continue

Dans une philosophie DevOps, la **livraison continue (Continuous Delivery)** consiste à essayer de produire une release à chaque modification du code source

Produire une release implique :

- Des tests automatiques poussés permettant d'avoir un très haut niveau de confiance dans l'artefact produit
- Tagger l'artefact et le stocker dans un dépôt

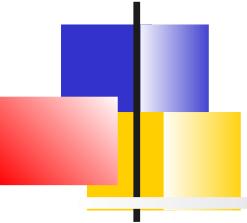
Les release peuvent alors être déployés en QA et des tests manuels peuvent être effectués afin de décider d'une mise en production éventuelle



Déploiement continu

Le **déploiement continu (Continuous Deployment)** est le stade ultime de l'intégration continue

La totale confiance dans les tests exécutés lors de la production de release permet de déployer automatiquement en production.



Phases de la mise en place

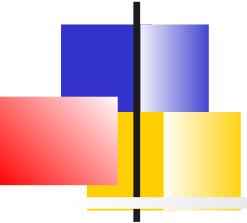
La mise en place de l'intégration/déploiement continu ne se fait pas en 1 jour.

En général, cela passe par plusieurs phases :

1. Pas de serveur de build
2. Serveur de build et Nightly Builds
3. Nightly Builds et test basiques automatisés
4. Déploiement automatisé en environnement de recette et Obtention des métriques
5. Renforcement des tests, distinction environnement d'intégration et de QA
6. Automatisation du déploiement en production,
7. Tests d'acceptance et Déploiement continu

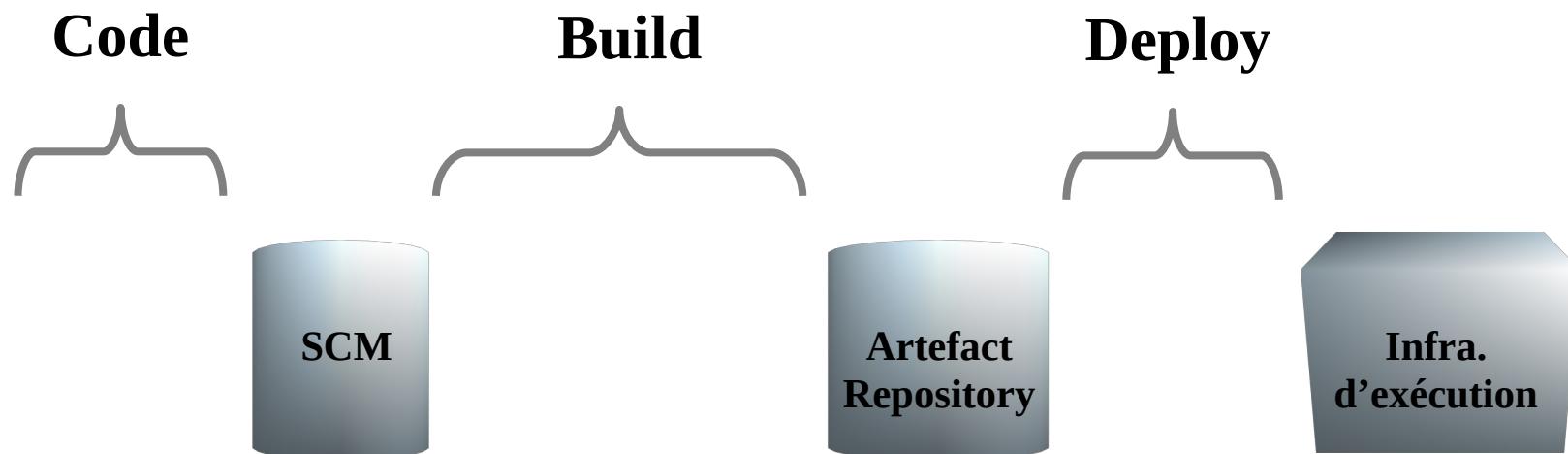


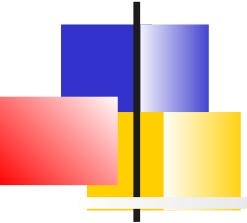
Cycle de vie du code et outils DevOps



Cycle de vie du code

- 1) Le code est testée localement puis poussé dans le dépôt
- 2) Le build construit l'artefact et le stocke dans un dépôt
- 3) L'outil de déploiement récupère l'artefact et le déploie sur l'infra d'exécution



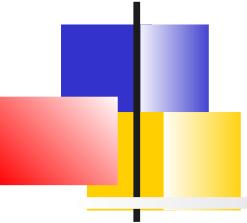


Types d'outils

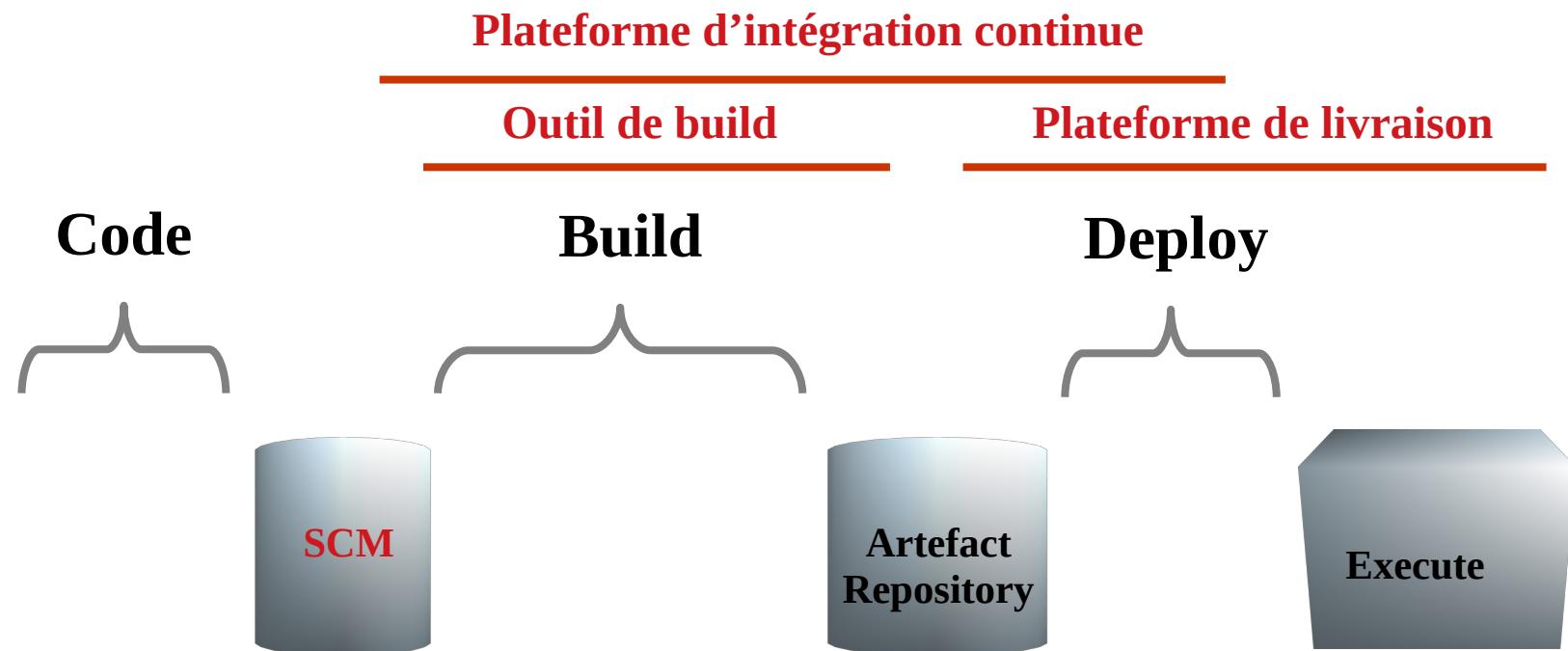
3 types d'outils s'intègrent dans le processus de CI/CD :

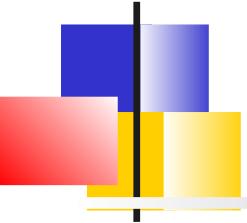
- Le **SCM** : centralise le code source (applicatif, code de build, des tests, scripts de provisionnement, ...). Il gère plusieurs branches ou versions plus ou moins stables
- **L'outil de build** : exécute les étapes nécessaires à la production de l'artefact
- La **PIC** qui à partir des branches du SCM exécute des pipeline de construction différentes et décident des déploiement dans les dépôts
- La **plate-forme de livraison** permet de contrôler une version à livrer et provisionner les cibles de production

Bien que ces trois groupes soient clairement identifiés, les outils sont parfois capables de gérer plusieurs aspects du processus



Outils et Cycle de vie



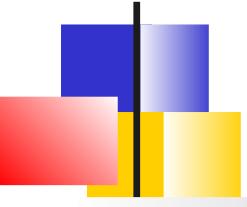


Dépôts et formats des artefacts

En fonction de la plateforme de livraison, différents types d'artefacts peuvent être générés par le build

- Code applicatif à déployer sur un serveur pré-provisionné.
Ex : *Appli JavaEE déployé sur un serveur applicatif provisionné mutualisant des applications*
- Code applicatif + serveur embarqué
Ex : *Application standalone déployé sur un serveur provisionné (OS + JVM par exemple)*
- Image d'un conteneur ou plusieurs images collaborant
Ex : *Architecture Microservices déployé sur orchestrateur de conteneurs ou Cloud*

Certaines solutions ont comme vocation de gérer tous ces formats. D'autres sont spécialisés



Infrastructure

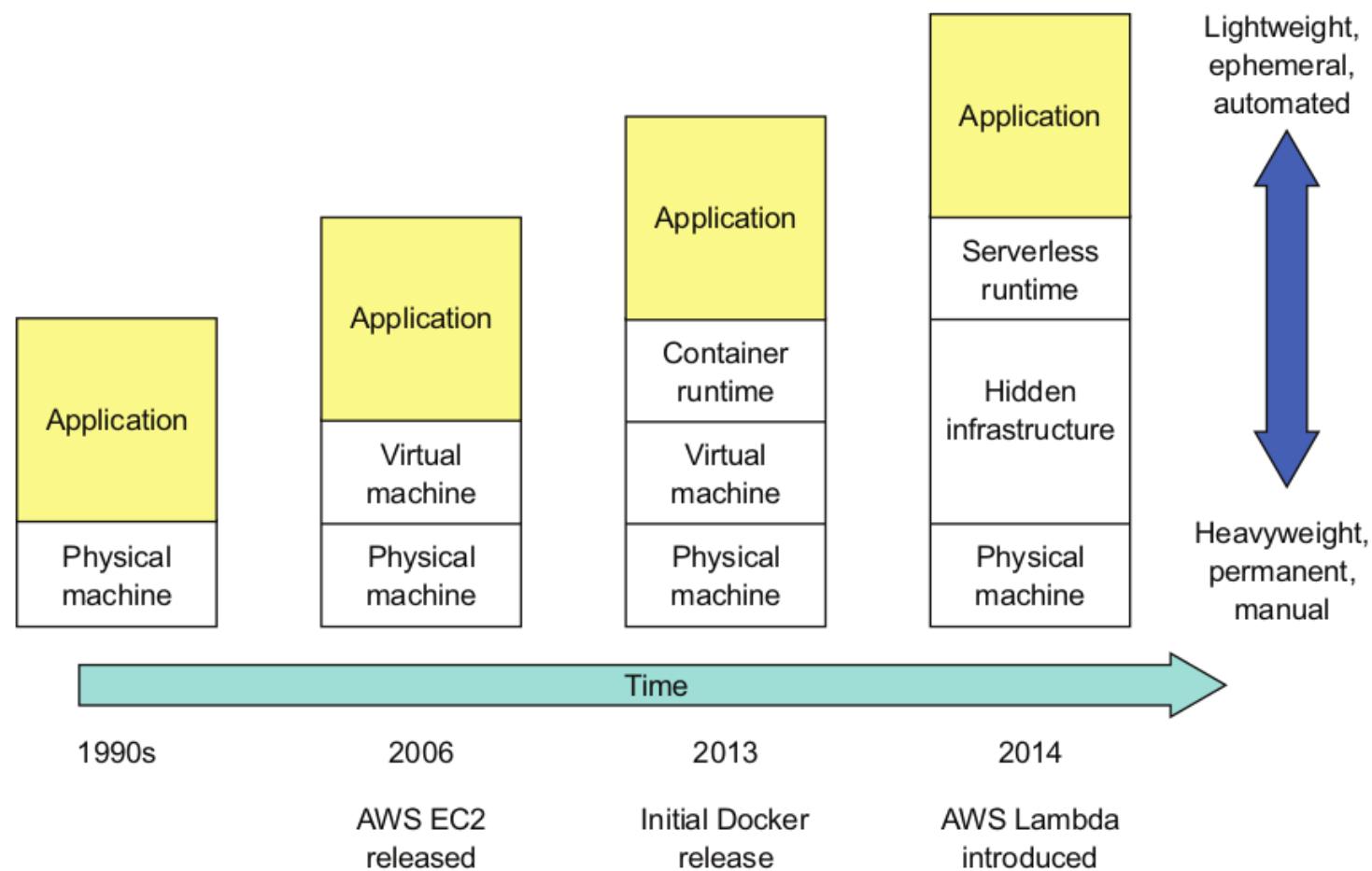
L'application produite nécessite une infrastructure d'exécution constituée de :

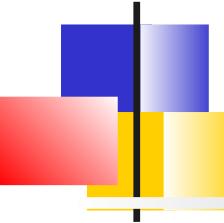
- **Matérielles** : Combien de CPU, RAM, Disque sont nécessaires pour l'application
- **Système d'exploitation** : Quelle est le système d'exploitation Cible
- **Middleware, produits, stack** : Quelles sont le middleware et les produits à installer ? Serveur applicatif, Base de données, ...

L'infrastructure est déclinée dans les différents environnements requis (intégration, QA, production)

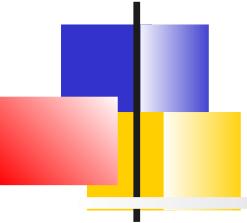
Préparer l'infrastructure et les logiciels nécessaires s'appelle le **provisionnement**. Dans un contexte de CI/CD, il doit être également automatisé.

Evolution des infrastructures





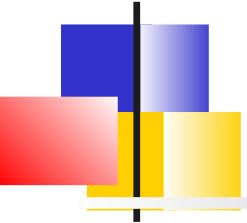
Influence de DevOps sur l'architecture des systèmes



Introduction

Avec DevOps une nouvelle architecture de systèmes visant à améliorer la rapidité des déploiements des retours utilisateur est apparu : les « **micro-services** »

C'est le même objectif que l'approche *DevOps* : « *Déployer plus souvent* »

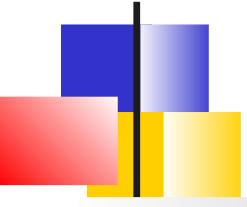


Architecture

Une architecture micro-services implique la décomposition des applications en très petits services

- faiblement couplés
- ayant une seule responsabilité
- Développés par des équipes full-stack indépendantes.

On l'appelle également *SOA 2.0*



Caractéristiques

Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

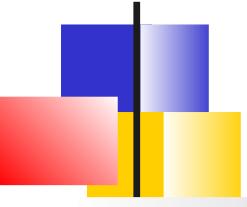
Principe de la responsabilité unique : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

Une interface explicitement publiée : Un producteur de service publie une interface qui peut être consommée

DURS (Deploy, Update, Replace, Scale)

indépendants : Chaque service métier peut être indépendamment déployé, mis à jour, remplacé, scalé

Communication légère : REST sur HTTP, STOMP sur WebSocket,



Bénéfices

Scaling indépendant : les services les plus sollicités (cadence de requête, mutualisation d'application) peuvent être scalés indépendamment (CPU/mémoire ou sharding),

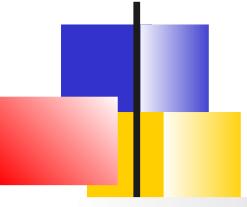
Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes

Maintenance facilitée : Le code d'un micro-service est limité à une seule fonctionnalité

Hétérogénéité des langages : Utilisation des langages les plus appropriés pour une fonctionnalité donnée

Isolation des fautes : Les fautes et dysfonctionnement sont plus faciles à identifier

Communication inter-équipe renforcée : Full-stack team



Contraintes

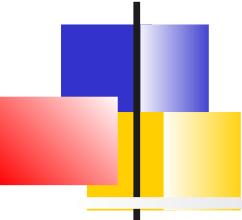
RéPLICATION : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

Découverte automatique : Les services sont typiquement distribués dans un environnement PaaS. Le scaling peut être automatisé selon certains métriques. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

Monitoring : Les services sont surveillés en permanence. Des traces sont générées et éventuellement agrégées

Résilience : Les services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

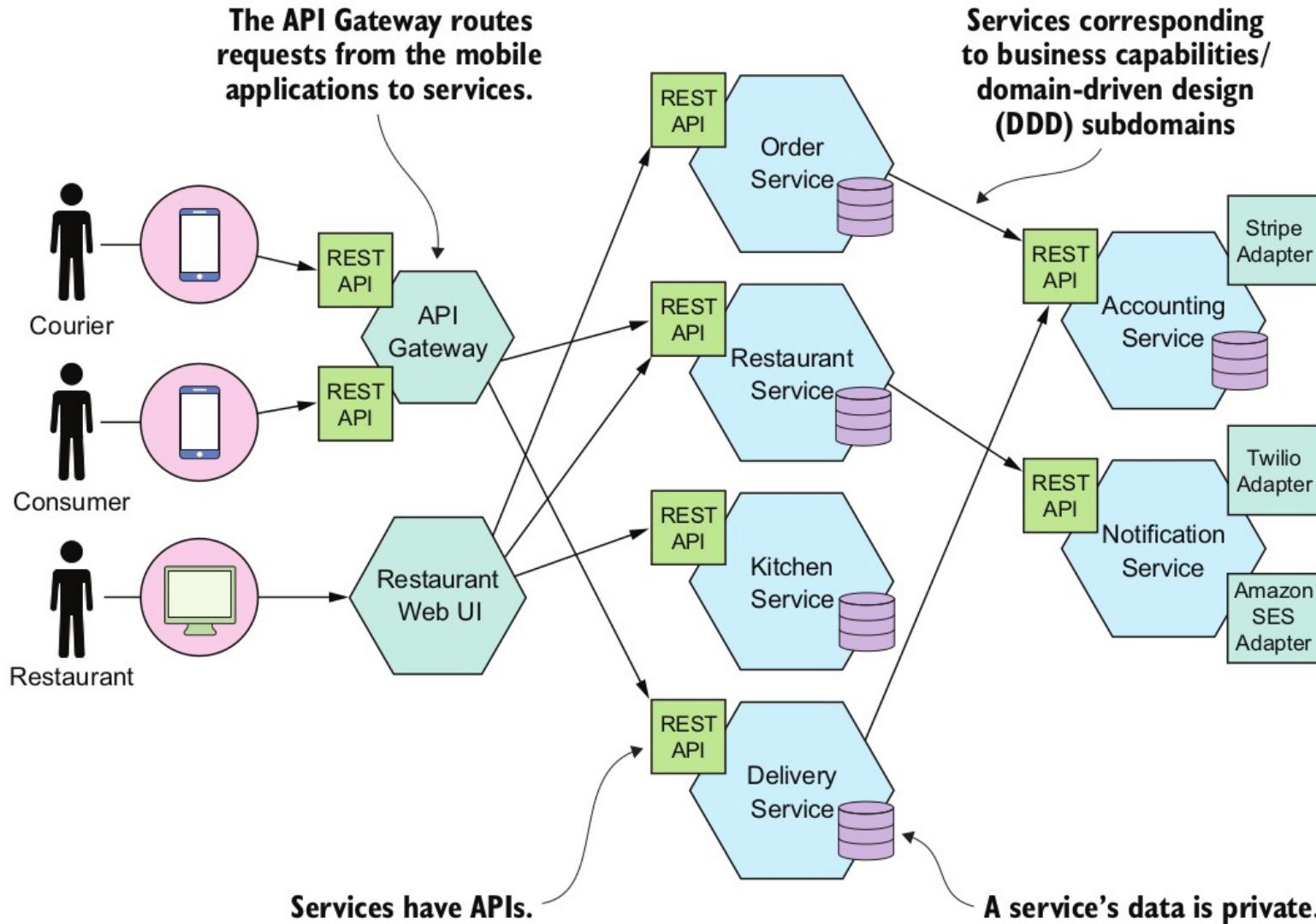
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



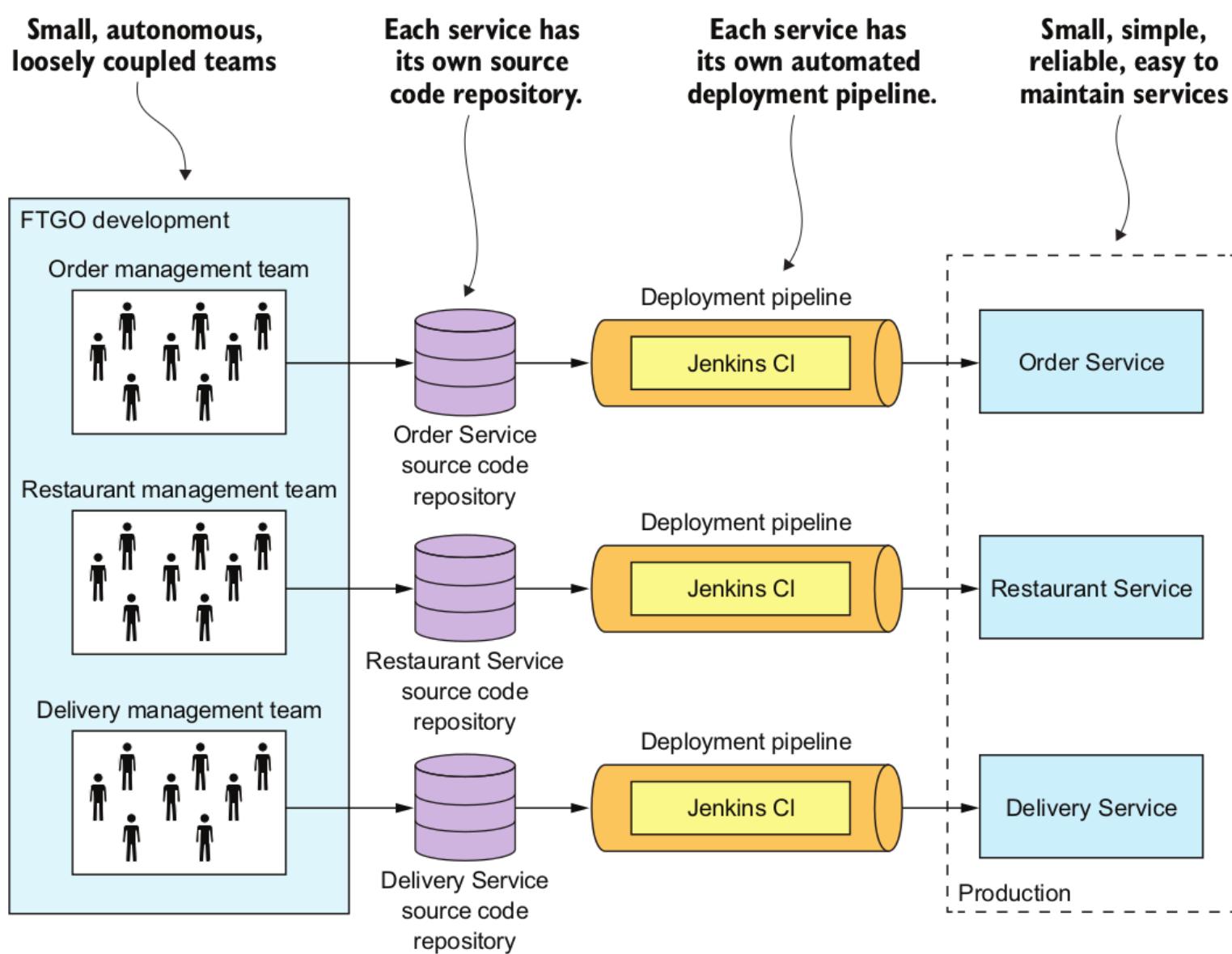
Les 12 facteurs de réussite

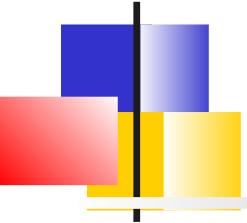
- I. Outil de scm** : Unique source de vérité
- II. Dépendances**: Déclare explicitement et isole les dépendances du code source
- III. Configuration** : Configuration séparée du code
- IV. Services** d'appui (backend) : Considère les services d'appui comme des ressources attachées, possibilité de switcher sans modification de code
- V. Build, release, run** : Permet la coexistence de différentes releases en production
- VI. Processes** : Exécute l'application comme un ou plusieurs processus stateless.
Déploiement immuable
- VII. Port binding** : Application est autonome (pas de déploiement sur un serveur). Elle expose juste un port TCP
- VIII. Concurrence** : Montée en charge grâce au modèle de processus
- IX. Disposability** : Renforce la robustesse avec des démarrages et arrêts rapides
- X. Dev/prod parity** : Garder les environnements de développement, de pré-production et de production aussi similaires que possible
- XI. Logs** : Traiter les traces comme un flux d'événements
- XII. Processus d'Admin** : Considérer les tâches d'administration comme un processus parmi d'autres

Une architecture micro-service



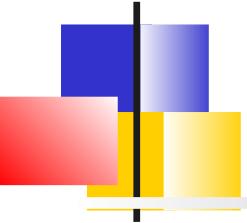
Organisation DevOps





Outils de pilotage

Rappels sur l'agilité
Outils de gestion d'issues



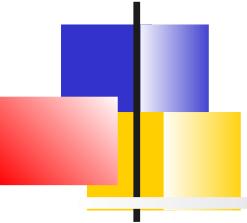
Introduction

Les outils de pilotage sont à destination de toute l'équipe agile :

- Métier
- Full-stack développeur

Ils intègrent les méthodes agiles et permettent :

- La planification
- L'élaboration, l'affinement des spécifications, le suivi des échanges entre développeurs et métier

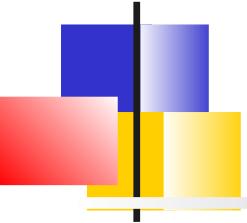


Evolution

Des solutions se sont positionnées exclusivement sur ces fonctionnalités de pilotage. Ex : Jira

Mais de plus, ces fonctionnalités sont intégrés dans des outils plus large prenant tous les aspect DevOps :

- Gitlab CI
- Github
- BitBucket
- Azure DevOps
-



Principes communs aux méthodes agiles

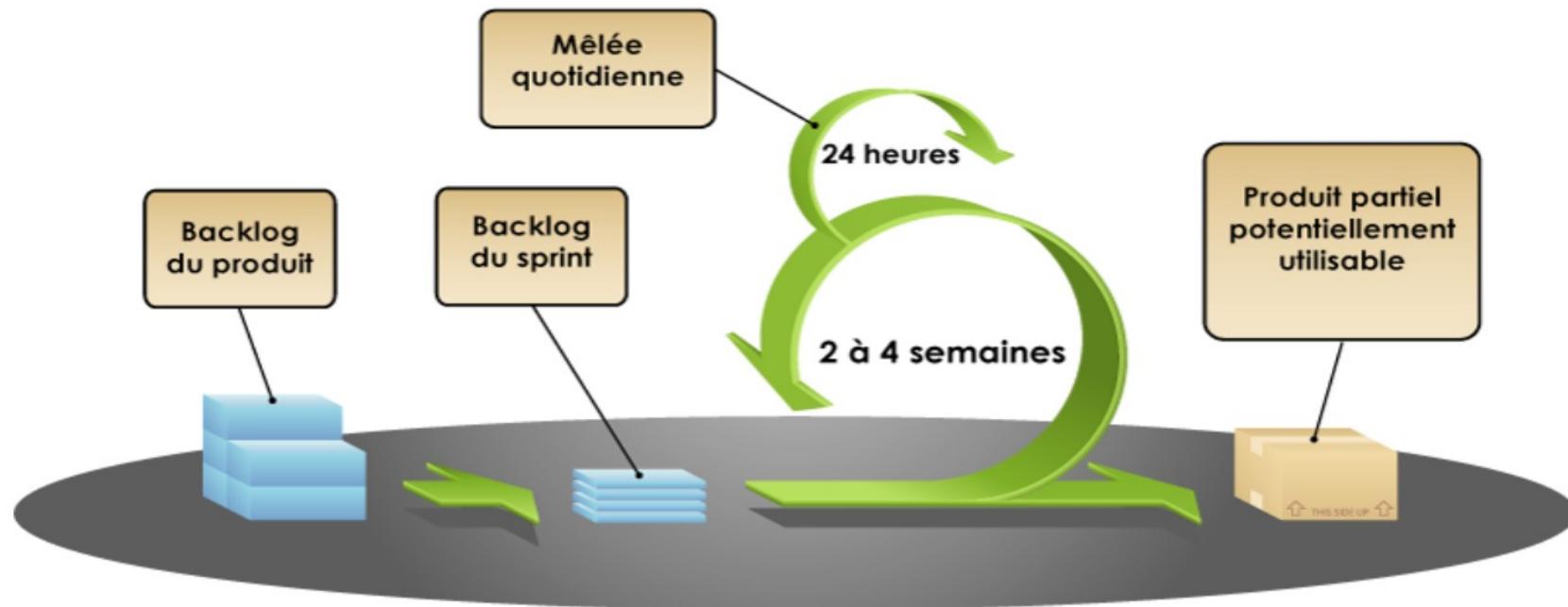
Les principes d'agilité :

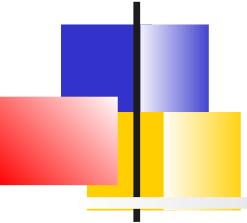
- la mise en place de pratiques **itératives**,
- des projets plus **petits**
- Collaboration renforcée entre le métier et la technique

→ On retrouve ces principes dans *RUP*, *XP Programming*, *Scrum*, *Safe*, *Spotify*, etc..¹

1. Voir annexe

Le rythme Scrum





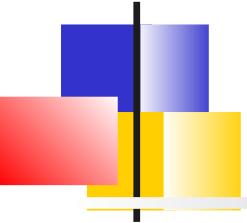
BackLog

Les spécifications sont exprimées sous forme de **UserStory** :

As a [ROLE], I want [DESIRE] so that [BENEFIT].

– => Les spécifications sont axées sur l'utilisateur et la *business value*

Les *UserStory* peuvent être complétées par des tests d'acceptance

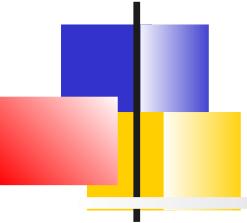


Exemple

Exemple de UserStory pour Magasin en ligne :

"Abandon lors de la saisie de carte de crédit invalides

Dans les tests utilisateurs, nous avons vu beaucoup de gens qui ont fait des erreurs en entrant leur n° de carte de crédit. Nous devons être aussi utiles que possible ici pour éviter de perdre des utilisateurs à ce stade crucial de la transaction."



Tests d'acceptance

Syntaxe Gherkin

#language : fr

Fonctionnalité: Abandon lors de la saisie de carte de crédit invalides

Dans les tests utilisateurs, nous avons vu beaucoup de gens qui ont fait des erreurs en entrant leur n° de carte de crédit. Nous devons être aussi utiles que possible ici pour éviter de perdre des utilisateurs à ce stade crucial de la transaction.

Contexte:

Étant donné que j'ai choisi certains articles à acheter

Et que je suis sur le point d'entrer les détails de ma carte de crédit

Scénario: numéro de carte de crédit trop court

Lorsque j'entre un numéro de carte de 15 chiffres seulement

Et tous les autres détails sont corrects

Et je soumets le formulaire

Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message m'informant du nombre correct de chiffres

Scénario: la date d'expiration ne doit pas être antérieure

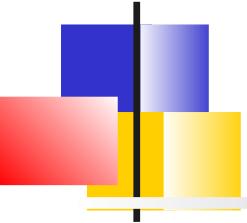
Lorsque j'entre une date d'expiration de carte qui est dans le passé

Et tous les autres détails sont corrects

Et je soumets le formulaire

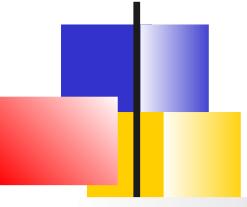
Alors, le formulaire doit être affiché de nouveau

Et je devrais voir un message me disant que la date d'expiration doit être fausse



Outils de pilotage

Rappels sur l'agilité
Outils de gestion d'issues



Introduction

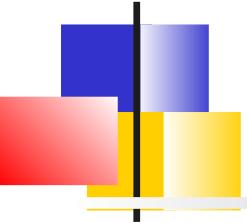
Les outils disponibles sont suffisamment configurable afin de s'adapter aux les particularités de chaque méthodologie agile.

Le concept principal est généralement une **issue** qui peut représenter :

- Une user story
- Un demande d'évolution
- Une déclaration de bug
- Un idée d'amélioration

Les issues sont généralement affectés à des **milestones** qui peuvent représenter :

- Une release
- Un sprint
- Une date de livraison
-



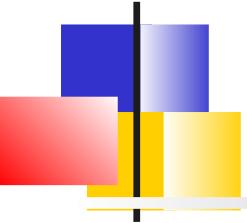
Labels

Les labels (illustrés par des badges de couleur) permettent de classifier les issues :

- Type (Bug, Idée, RFC, User Story, ...)
- Domaine (Front-end, Back-end, CI/CD,...)
- Statut (Review, Duplicate, ...)

Les outils propose des labels par défaut et il est possible de configurer ses propres labels

Exemple



GitLab Issues page showing a list of user stories. The sidebar on the left includes icons for Projects, Groups, Activity, Milestones, Snippets, and a search bar.

Relations between Issues

#4058 · opened a year ago by Job van der Voort · Discussion · Product work · UX · feature proposal · issues · 266 likes, 1 dislike, 127 comments · updated about 19 hours ago

Add group wiki page support

#4037 · opened a year ago by Ken Phllis Jr · Next 3-6 months · Accepting Merge Requests · Community Contribution · Platform · UX · customer · feature proposal · shortlist · wiki · 215 likes, 67 comments · updated a week ago

Merge train/Release train/Merge when master succeeds: run build on merged code before merging

#4176 · opened a year ago by Frederik Zahle · Backlog · CI/CD · EE Premium · Product work · ci-build · customer · direction · feature proposal · frequently duplicated · 208 likes, 1 dislike, 79 comments · updated 3 days ago

Custom Roles

#12736 · opened a year ago by Rolando Torres · Backlog · Platform · SP2 · customer · direction · feature proposal · frequently duplicated · permissions · user management · 169 likes, 1 dislike, 90 comments · updated 4 days ago

Please bring squash option when merging MRs to CE

#34591 · opened 3 months ago by Jonas Kello · Discussion · feature proposal · merge requests · stewardship · 166 likes, 2 dislikes, 24 comments · updated 5 days ago

Customize branch name when using create branch in an issue

#21143 · opened a year ago by Adi Gerber · 10.1 · Accepting Merge Requests · Community Contribution · Discussion · In review · UX ready · feature proposal · frequently duplicated · issues · repository · 140 likes, 1 dislike, 86 comments · updated about 9 hours ago

Move Fast-Forward Merge and Semi-Linear Merge to CE

#20076 · opened a year ago by Markus KARG · 10.1 · Deliverable · Discussion · In review · UX ready · backend · feature proposal · frontend · merge requests · 137 likes, 35 comments · updated a day ago

Provide an option/toggle in settings so that private repo commits show up on public user profile graph

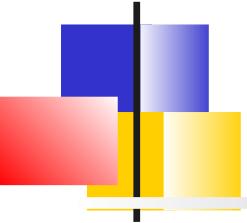
#14078 · opened a year ago by Cameron Banga · Next 3-6 months · Platform · feature proposal · frontend · graphs · settings · 133 likes, 1 dislike, 108 comments · updated a week ago

Let's encrypt support for GitLab Pages

#28996 · opened a year ago by Job van der Voort · Backlog · CI/CD · feature proposal · pages · 117 likes, 47 comments · updated 2 months ago

Shared CI runners for groups

#10244 · opened a year ago by Lucas Koenig · Next 3-6 months · CI/CD · CDD · customer · feature proposal · frequently duplicated · 103 likes, 45 comments · updated about 16 hours ago



Tableaux de bords

Les outils proposent des **boards** qui permettent de visualiser les issues/milestones de différentes façon :

- Tableau de bord Kanban
- Backlog, Sprint
- Roadmap
- Epics
- Tableaux de bords personnalisés
- ...

Tableau de bord : Scrum

JIRA Dashboards Projects Issues Boards Create Search ⌂ 0 days remaining Complete Sprint Board ⌂

All sprints Switch sprint +

QUICK FILTERS: Product UI Server Only My Issues Recently Updated

12 To Do	4 In Progress	1 Code Review	7 Done
<p>▼ TIS Developer Love 3 issues</p> <p>TIS-37 When requesting user details the service should return prior trip SeeSpaceEZ Plus 2</p> <p>TIS-10 Bad JSON data coming back from hotel API SeeSpaceEZ Plus</p>	<p>TIS-8 Requesting available flights is now taking > 5 seconds SeeSpaceEZ Plus</p>		
<p>▼ Everything Else 21 issues</p> <p>TIS-68 Homepage footer uses an inline style - should use a class Large Team Support</p> <p>TIS-20 Engage Saturn Shuttle Lines for group tours Space Travel Partn... 3</p> <p>TIS-12 Create 90 day plans for all departments in the Mars Office Local Mars Office 9</p> <p>TIS-15 Establish a catering vendor to provide meal service Local Mars Office 4</p>	<p>TIS-17 Engage Saturn's Rings Resort as a preferred provider Space Travel Partn... 3</p> <p>TIS-26 Engage the Red Titan Hotel as a preferred provider Space Travel Partn... 3</p> <p>TIS-33 Select key travel partners for the Saturn Summer Sizzle Summer Saturn Sale 1</p>	<p>TIS-67 Developer Toolbox does not display by default Large Team Support</p> <p>TIS-66 Add pointer to main css file to instruct users to create child themes Large Team Support</p>	<p>TIS-45 Email non registered users to sign up with Teams In Space Large Team Support 2</p> <p>TIS-49 Draft network plan for Mars Office Local Mars Office 5</p> <p>TIS-69 Add a String anonymizer to TextUtils Large Team Support</p> <p>TIS-23 Local Mars Office 1</p>

Developer Toolbox does not display by default
Attach Files

Screen Shot 2015-08-13 at 4.1 326 kB 20/Aug/15 12:06 PM

Sub-Tasks Create Sub-Task

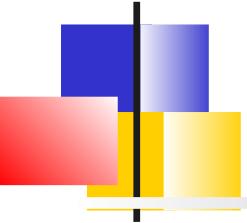
Issue Key	Summary	Status	Actions
TIS-127	Check Java version	OPEN	edit

Development

1 branch	Updated 17/May/14
7 commits	7:32 AM Latest 17/May/14
1 pull request OPEN	7:30 AM Updated 17/May/14
3 builds	7:32 AM Latest 16/May/14

Deployed to Staging and Production

Project administration

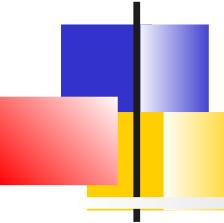


Collaboration autour de l'issue

De nombreuses fonctionnalités de collaboration sont proposées autour de l'issues :

- Threads de discussion et notifications/alertes
- Workflows
- Intégration DevOps :
 - Association aux modifications de code,
 - aux pull/merge requests
 - A la revue de code
 - Aux pipelines, aux résultats des tests automatisés
 - Aux environnements de test

Vue détaillée issue



Open Issue #1395 opened about 14 hours ago by  Marcia Ramos 0 of 2 tasks completed 1 [New issue](#) [Close issue](#) [Edit](#) 2 Todo [Mark done](#) ➔

GitLab Issue 12

Hello World! 🎉

This is my issue's description, written in markdown (GitLab Flavored Markdown).

This is an [h3](#)

Let's quote someone here

Add a task list:

Task 1
 Task 2

Mention merge requests ([gitlab-org/gitlab-ee!1784 \(merged\)](#)) and issues ([gitlab-org/gitlab-ee#2101 \(closed\)](#)) and hover over them to see their titles.

Invite users to collaborate with [@mentions](#): @marcia 13

Edited just now by Marcia Ramos

1 Related Merge Request

[!1784 How to Configure LDAP with GitLab EE in GitLab.org / GitLab Enterprise Edition](#) Merged 14

15

18 Create a merge request

✓ Create a merge request
Creates a branch named after this issue and a merge request. The source branch is 'master' by default.

Create a branch
Creates a branch named after this issue. The source branch is 'master' by default.

10 Notifications Unsubscribe

11 Reference: [gitlab.com/www-g...](#)

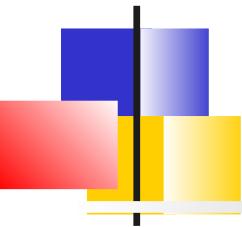
16

Write Preview

Write a comment or drag your files here...

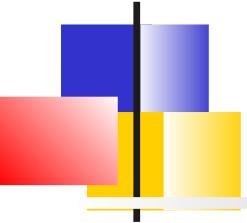
Markdown and slash commands are supported

17 Comment Close issue



Source Control Management

L'unique source de vérité
Workflows de collaboration



Introduction

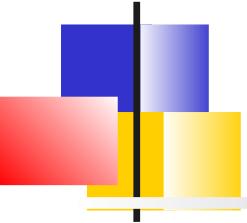
Dans l'approche DevOps, le SCM est **l'unique source de vérité**

Tout ce qui est nécessaire aux projets y est stocké et versionnés :

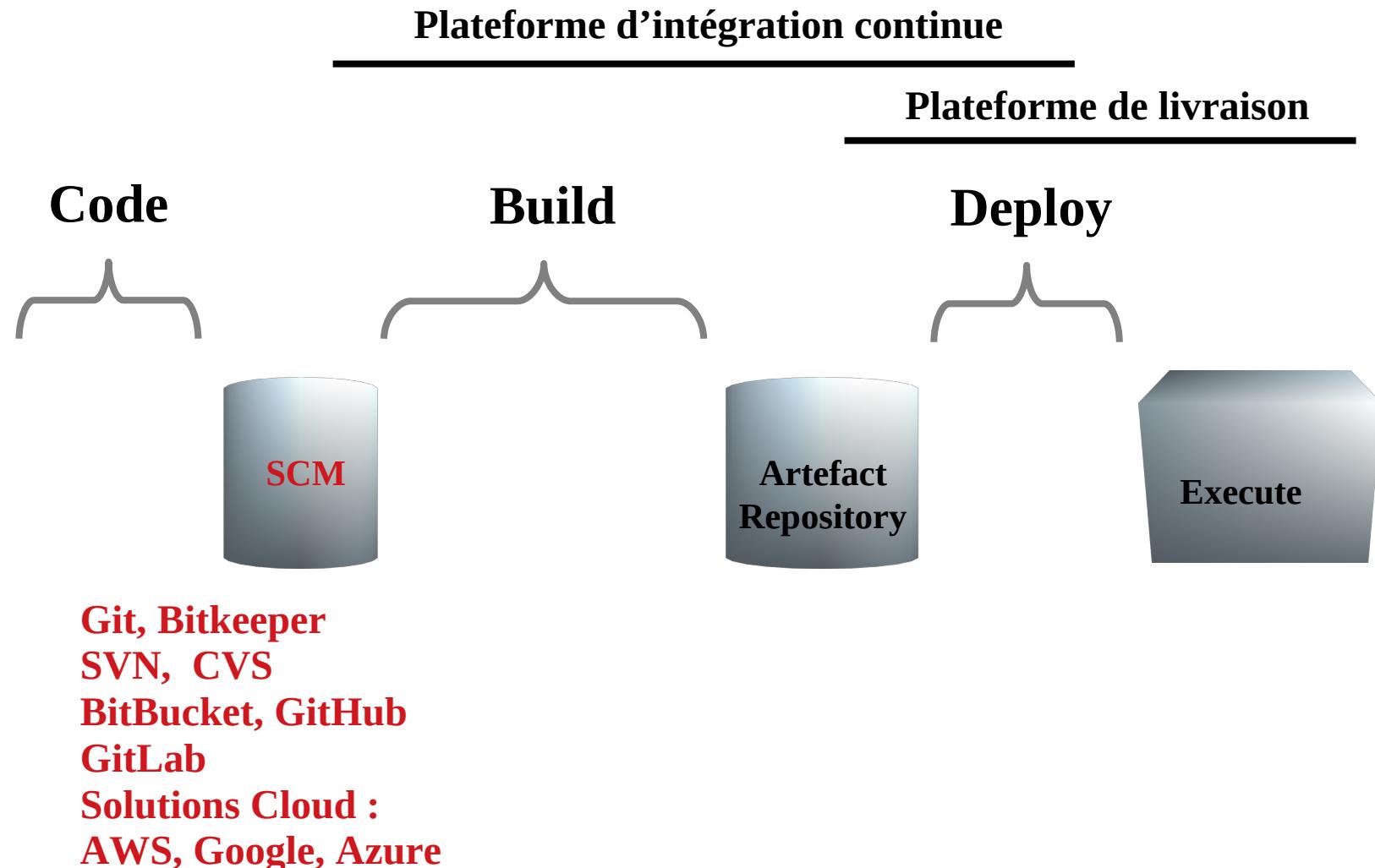
- Code source mais aussi
- Fichiers et outils de build, Pipelines de CI, Tests automatisés, scripts de provisionnement et de déploiements
- Docs

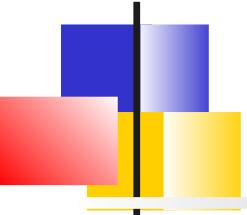
=> A partir d'un clone du dépôt, un nouveau participant au projet doit être capable de faire tout seul :

- Build
- Deploy



SCMS et Cycle de vie



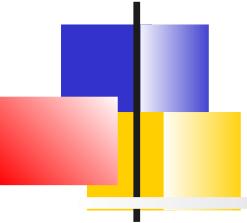


SCM

Un **SCM** (*Source Control Management*) est un système qui enregistre les changements faits sur un fichier ou une structure de fichiers afin de pouvoir revenir à une version antérieure

Le système permet :

- De restaurer des fichiers
- Restaurer l'ensemble d'un projet
- Visualiser tous les changements effectués leurs auteurs et leurs commentaires associés
- Le développement concurrent (branche)



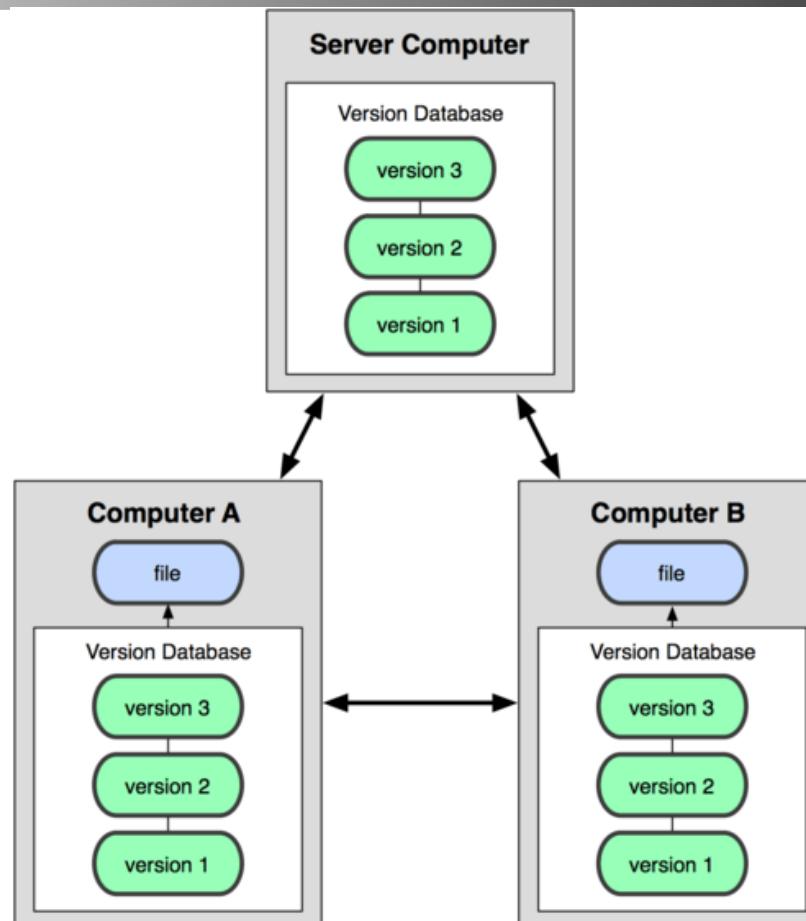
Patch

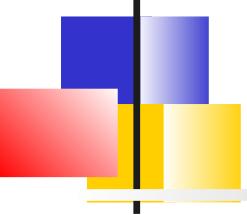
Chaque modification du code source enregistré dans le SCM peut être visualisé sous forme de **patches**

Un patch indique les blocs de lignes d'un fichier ayant été modifiés

```
@ -35,7 +35,7 @@ stage('Parallel Stage') {
  stage('Déploiement artefact') {
    steps {
      echo 'Deploying..'
      -     sh './mvnw -Pprod clean deploy'
      +     sh './mvnw --settings settings.xml -Pprod clean deploy'
      dir('target/') {
        stash includes: '*.jar', name: 'service'
      }
    }
  }
}
```

SCM distribués : Git, Bitbucket





Atouts des SCM distribués (Git)

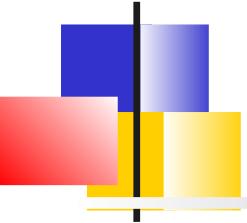
La plupart des opérations sont locales et donc très rapides

Les branches sont de simples pointeurs.

- La création et la suppression de branches sont des opérations légères.
- On peut avoir de nombreuses branches dans un projet
- Extraire une branche particulière est très rapide car Git n'a pas besoin de reconstruire les fichiers à partir des patchs

=> Les workflows de collaboration sont variés.

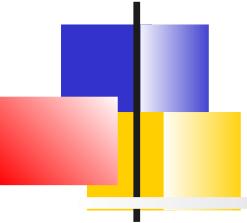
Ils doivent être bien spécifiés et bien compris par les acteurs d'un projet DevOps donné



SCM : Commit, Branches et Tag

Les SCM contiennent l'historique complet des sources du projet

- Les **commits** (Ids) permettent d'isoler chaque modification apportée par les développeurs, les historiser et les documenter
- Les **branches** permettent d'effectuer des travaux en isolation, lorsque les travaux sont terminés, ils sont intégrés dans une branche plus stable. (Par exemple, branche master de production)
- Les **tags** sont à priori immuables et permettent de fixer un instantané du projet. Ils correspondent en général à des releases déployable en production



Principales opérations Git

clone : Recopie intégrale du dépôt

checkout : Extraction d'une révision particulière

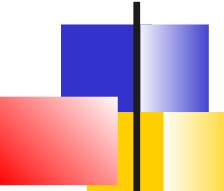
commit : Enregistrement de modifications de source

push/pull : Pousser/récupérer des modifications d'un dépôt distant

log : Accès à l'historique

merge, rebase : Intégration des modifications d'une branche dans une autre

Historique : gitk, Merge d'une feature branch



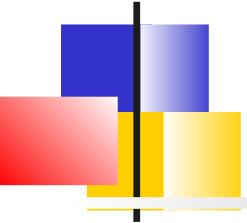
gitk screenshot showing the commit history of the v2 branch. The commits are listed in chronological order from top to bottom.

Commit Message	Date
v2 [8ca63811a3f545c13e7df30c774f2a92bc19886e] Merge branch 'sprint1' into v2 : Formulaire Filiere avec CKEditor	2021-04-23 17:37:39
sprint1 [dff8b6c6ba137f3ab36d44caabfa4b70fc3bac00] Remove ManageAndBack Reference	2021-04-23 17:37:20
Retours Vincent	2021-04-23 17:30:55
Champ description	2021-04-23 17:05:15
Barre de boutons : Apply et Supprimer; Toggle Components	2021-04-23 13:12:48
Bouton appliquer	2021-04-23 11:30:12
Toggle Autres filières	2021-04-23 11:05:48
Mini-description	2021-04-23 09:29:16
Nouvelle filière	2021-04-23 08:58:22
Création script de migration SQL	2021-04-23 08:58:06
New filière	2021-04-22 15:50:49
Couleurs fix	2021-04-20 17:53:32
Couleur dans le formulaire	2021-04-20 17:44:32
Adding colors dans filiere list	2021-04-20 17:01:37
v2.0.1 [5d58f1dd6b12de9a4f1cd5a5d41e5994689d3a90] Merge branch 'filieres' into v2	2021-04-20 15:31:34
List categorie, suppression colonne url dans filiere	2021-04-20 15:31:14
Categorie API resource	2021-04-20 14:48:24

Commit details for the first commit:

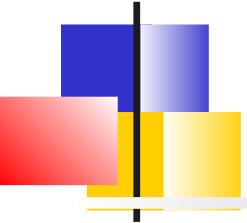
Auteur: David THIBAU <david.thibau@gmail.com> 2021-04-23 17:37:39
Validateur: David THIBAU <david.thibau@gmail.com> 2021-04-23 17:37:39
Parent: [dff8b6c6ba137f3ab36d44caabfa4b70fc3bac00](#) (Merge branch 'filieres' into v2)
Parent: [5d58f1dd6b12de9a4f1cd5a5d41e5994689d3a90](#) (Remove ManageAndBack Reference)
Branche: [remotes/origin/v2](#), v2
Suit: [v2.0.1](#)
Précède:

Merge branch 'sprint1' into v2 : Formulaire Filiere avec CKEditor



Source Control Management

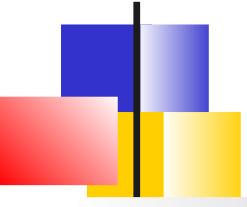
L'unique source de vérité
Workflows de collaboration



Introduction

Les SCMs distribués ont introduit différents workflows de collaboration entre développeurs :

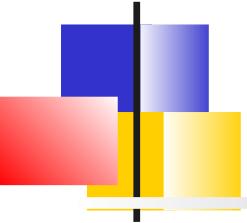
- Projets OpenSource (Linux, Github, ...) :
Workflow avec intégrateur basé sur les *pull-request*
- Editeur logiciel avec maintenance concurrente de plusieurs releases : **Gitflow**
- Projet DevOps avec déploiement continu :
GitlabFlow basé sur les merge-request



Gitlab Flow

Gitlab Flow est une stratégie simplifiée d'utilisation des branches pour un développement piloté par les features ou le suivi d'issues

- 1) Les corrections ou fonctionnalités sont développées dans une **feature branch**
- 2) Lorsque le développeur considère que le travail est terminé, il émet un *merge request* :
Demande de fusion dans la branche principale
- 3) Des tests sont effectués en isolation, une revue de code peut être effectué par le lead développeur.
Le responsable décide alors d'accepter ou de refuser la merge request
- 4) Si la MR est acceptée, les modifications de code sont intégrées dans la branche principale qui reste toujours potentiellement livrables en production

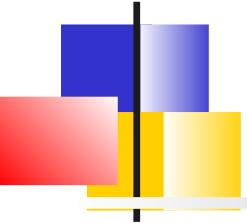


Gitlab Flow

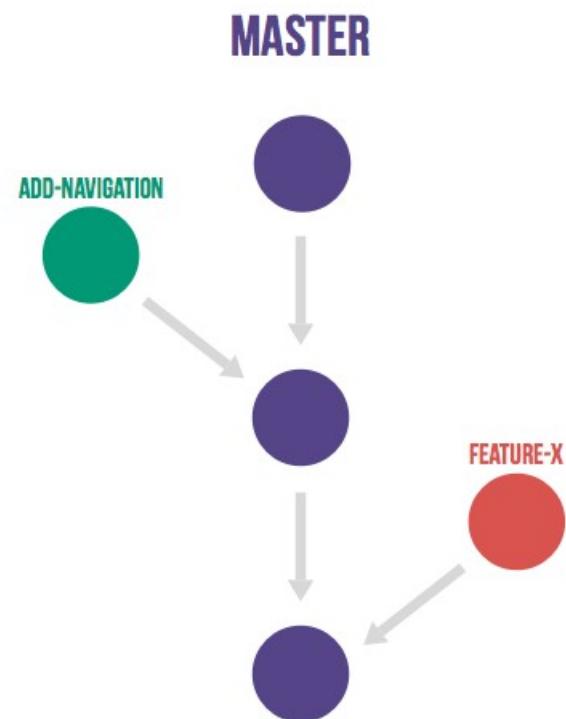
Gitlab Flow est une stratégie simplifiée d'utilisation des branches pour un développement piloté par les features ou le suivi d'issues

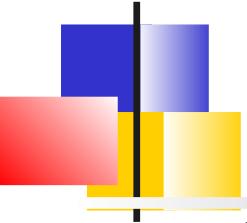
- 1) Les fix ou fonctionnalités sont développés dans une feature branch
- 2) Via un merge request, elles sont intégrées dans la branche master

Dans un scénario simple de type devops, la branche master est la branche de production



Features et Master



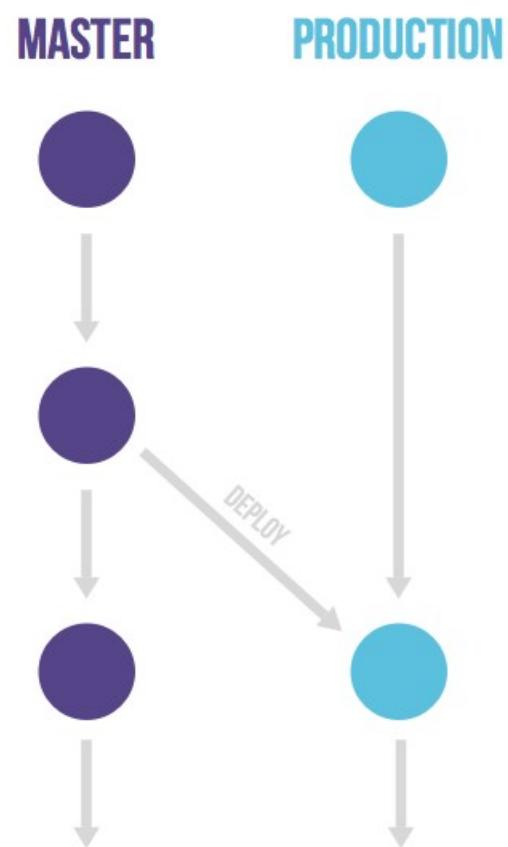


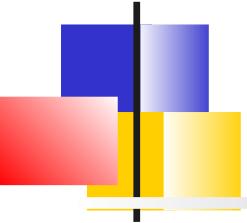
Branche de production

Si l'on veut maîtriser les déploiement vers la production, Il est possible d'utiliser une branche **production** qui reflète le code qui a réellement été déployé

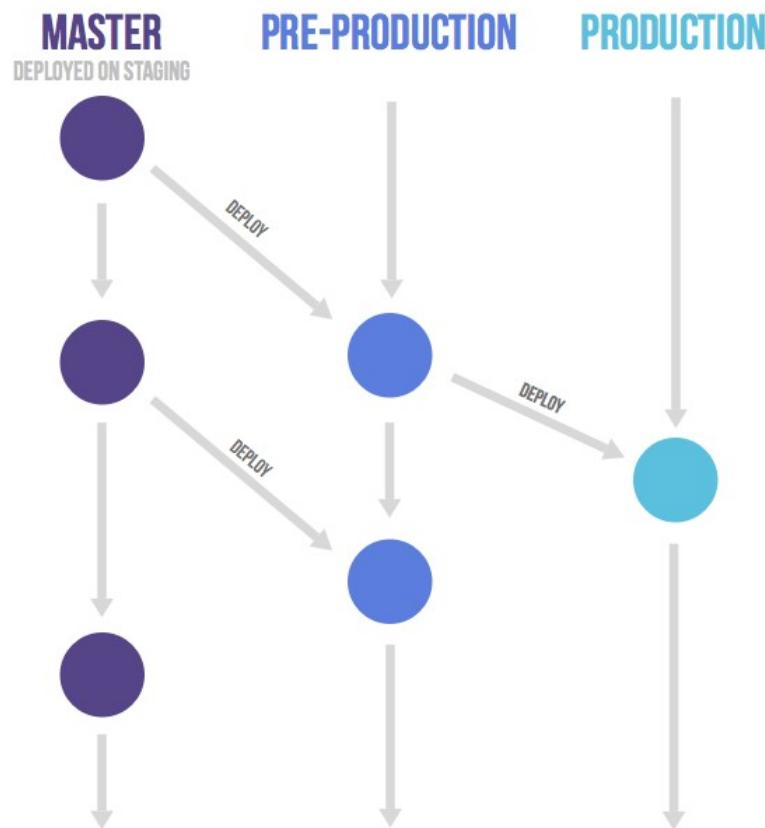
La mise en production consiste alors à fusionner *master* avec la branche de production puis déployer à partir de production.

Master et Production

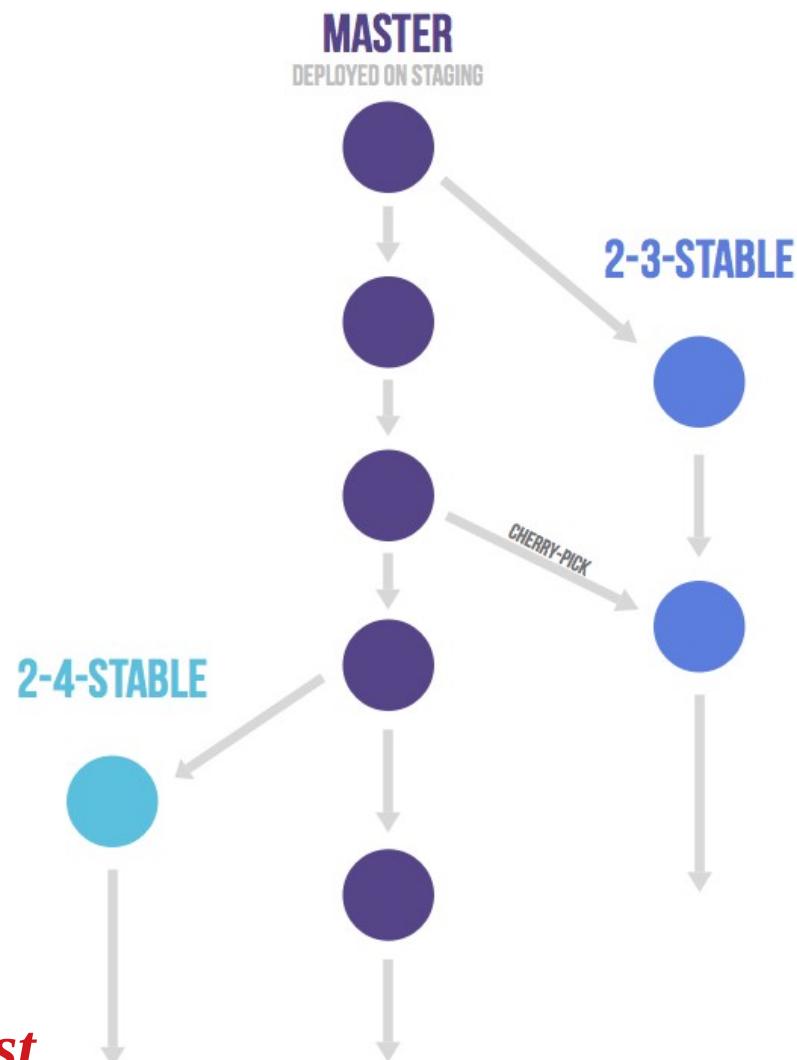


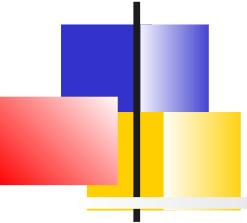


Déclinaison avec staging



Branches de releases

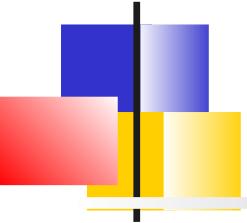




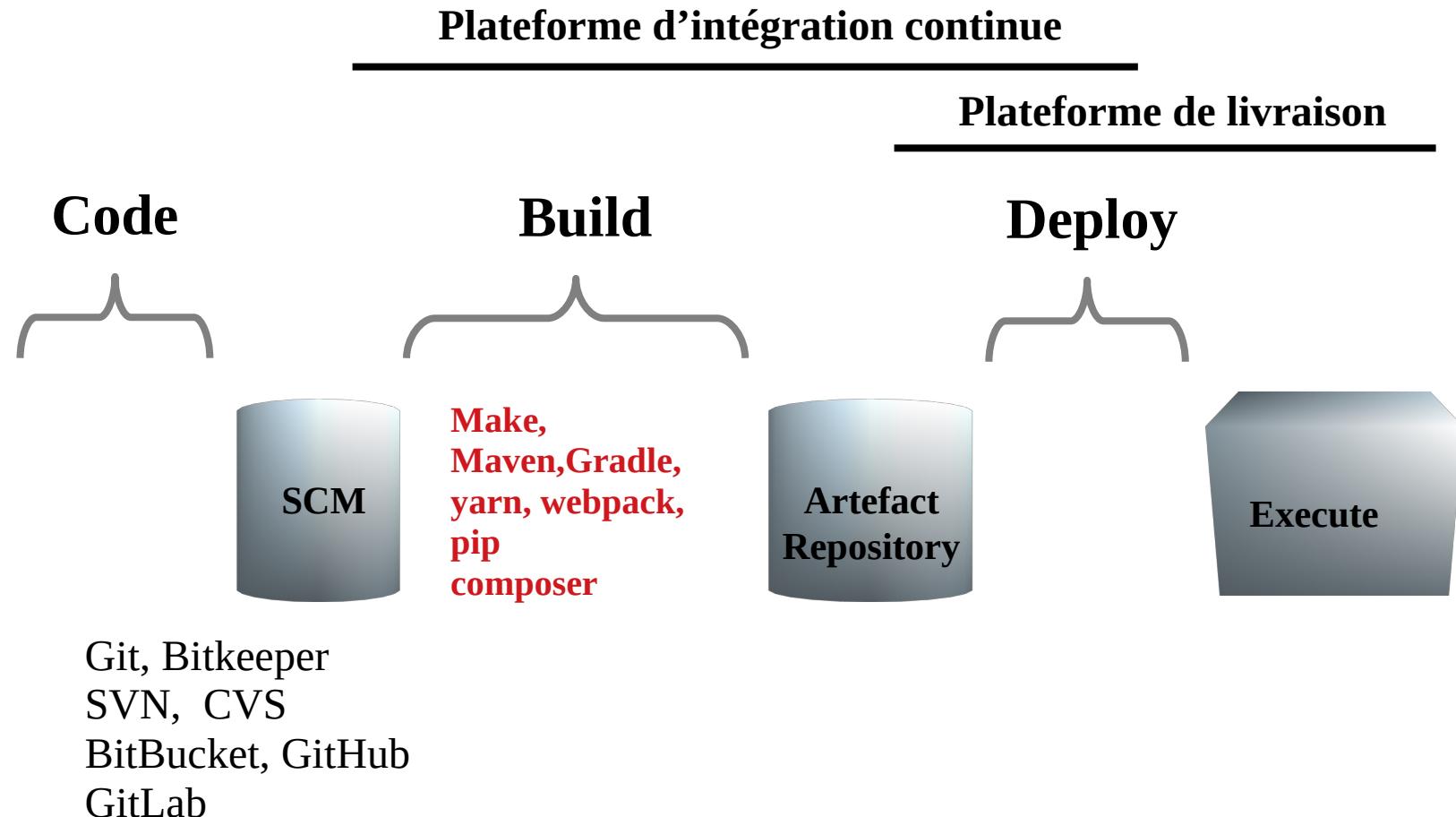
Build

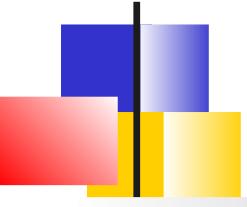
Caractéristiques d'un outil de build

Tests et analyse statique
Release et dépôts d'artefacts



Les outils de build dans le cycle de vie



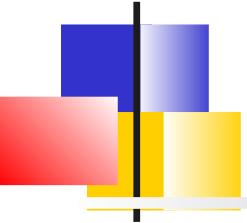


Introduction

Les objectifs d'un outil de build est de produire un artefact (exécutable) dans lequel on ait un minimum de confiance.

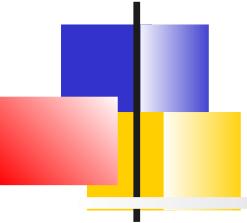
Cela implique :

- Avant de produire un artefact, effectuer le plus de vérifications possibles. Tests, analyse statique de code
- Créer des artefacts les plus adaptés aux environnements d'exécution en particulier la prod. Archive de code, Image de conteneur
- Publier un artefact réussi dans un dépôt



Pré-requis d'un build

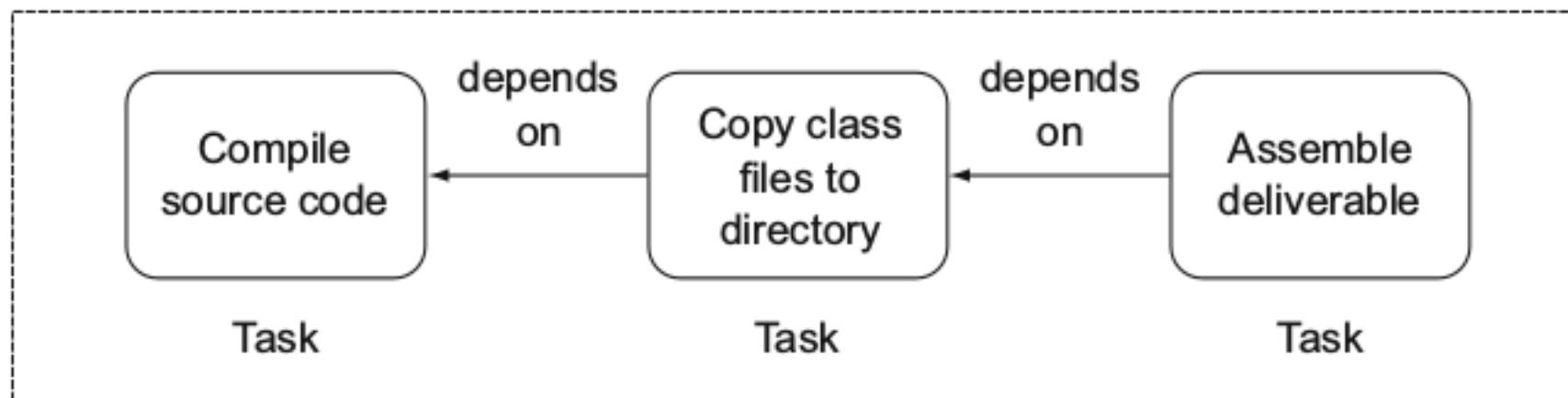
- **Proscrire les interventions manuelles** sujettes à erreur et chronophage
- Créer des builds **reproductibles** : Pour tout le monde qui exécute le build
- **Portable** : Ne doit pas nécessiter un OS ou un IDE particulier, il doit être exécutable en ligne de commande
- **Sûr** : Confiance dans son exécution
- **Rapide** : Parallélisme de tâches unitaires, Gestion de cache

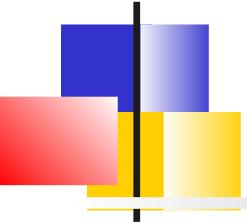


Graphe de build

Un build est une séquence ordonnée de tâches unitaires.

Les tâches ont des dépendances entre elles qui peuvent être modélisées via un **graphe acyclique dirigé** :





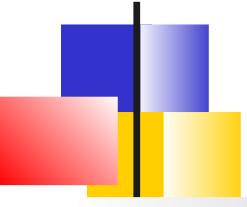
Composants d'un outil de build

Le fichier de build : Contient la configuration requises pour le build, les dépendances externes, les instructions pour exécuter un objectif sous forme de tâches inter-dépendantes

Une tâche unitaire prend une entrée effectue des traitements et produit une sortie. Une tâche dépendante prend comme entrée la sortie d'une autre tâche

Moteur de build : Le moteur traite le fichier de build et construit sa représentation interne. Des outils permettent d'accéder à ce modèle via une API

Gestionnaire de dépendances : Traite les déclarations de dépendances et les résout par rapport à un dépôt d'artefact contenant des méta-données permettant de trouver les dépendances transitive

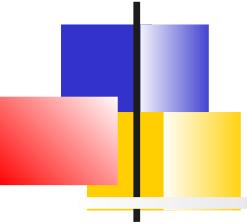


Monde Java

Apache Ant : L'ancêtre. Pas de gestionnaire de dépendances. Plein de tâches prédéfinies. Facilité d'extension. Pas de convention

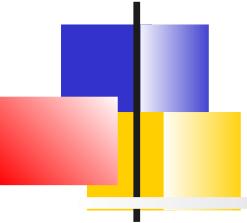
Maven : Le plus répandu et le plus supporté. Gestionnaire de dépendances. Extension par mécanisme de plugin. Verbeux car XML

Gradle : « *Build As Code* ». Allie les qualités de Maven et des capacités de codage du build. Concis, puissant mais moins supporté



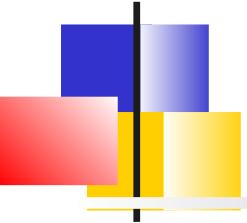
Exemple pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <!-- Coordonnées du projet -->
    <groupId>org.dthibau</groupId>
    <artifactId>forum</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>Forum</name>
    <!-- Dépendances du projet -->
    <dependencies>
        <dependency>
            <groupId>io.github.jhipster</groupId>
            <artifactId>jhipster</artifactId>
            <version>2.0.4</version>
        </dependency>
        <dependency>
            <groupId>com.jayway.jsonpath</groupId>
            <artifactId>json-path</artifactId>
            <version>2.0.4</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```



Exemple pom.xml (2)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-resources-plugin</artifactId>
      <version>${maven-resources-plugin.version}</version>
      <!-- Adaptation du cycle de build -->
      <executions>
        <execution>
          <id>docker-resources</id>
          <phase>validate</phase>
          <goals>
            <goal>copy-resources</goal>
          </goals>
          <configuration>
            <outputDirectory>target/</outputDirectory>
            <resources>
              <resource>
                <directory>src/main/docker/</directory>
              </resource>
            </resources>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</project>
```



Monde JavaScript/TypeScript

npm, yarn : Gestion de dépendance

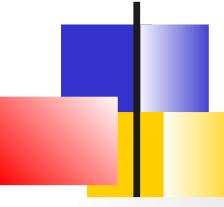
webpack : Création de bundle pour la production

grunt : Automatisation de tâches

gulp : Optimisation, Minification de code

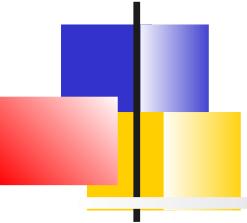
yeoman : Générateur de code

tslint : Analyse de code source, règles de codage



Exemple *package.json*

```
{  // Identification projet
  "name": "forum",
  "version": "0.0.0",
  "description": "Description for forum",
  "private": true,
  "license": "UNLICENSED",
  "cacheDirectories": [
    "node_modules"
  ], // Dépendances
  "dependencies": {
    "@angular/common": "4.3.2",
    "@angular/forms": "4.3.2",
    "@angular/http": "4.3.2",
    "@angular/platform-browser": "4.3.2",
    "@ng-bootstrap/ng-bootstrap": "1.0.0-beta.5",
    "bootstrap": "4.0.0-beta"
  }, // Dépendances pour le développement
  "devDependencies": {
    "@angular/cli": "1.4.2",
    "@angular/compiler-cli": "4.3.2",
    "@types/jasmine": "2.5.53",
    "webpack": "3.6.0",
    "webpack-dev-server": "2.8.2"
  }, // Moteur
  "engines": {
    "node": ">=6.9.0"
  }, // Raccourci
  "scripts": {
    "start": "yarn run webpack:dev",
    "serve": "yarn run start",
    "build": "yarn run webpack:prod",
    "test": "karma start src/test/javascript/karma.conf.js",
  }
}
```



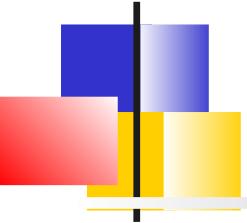
Autres langages

Php :

- **Composer** : Principalement gestionnaire de dépendances
- **PHPUnit** : Exécution de tests unitaires

Python

- **pip** : Gestionnaire de dépendances (packages)
- **unittest** : Tests unitaires

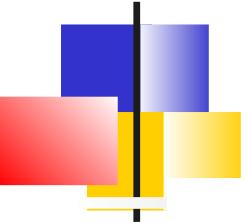


Outils de build

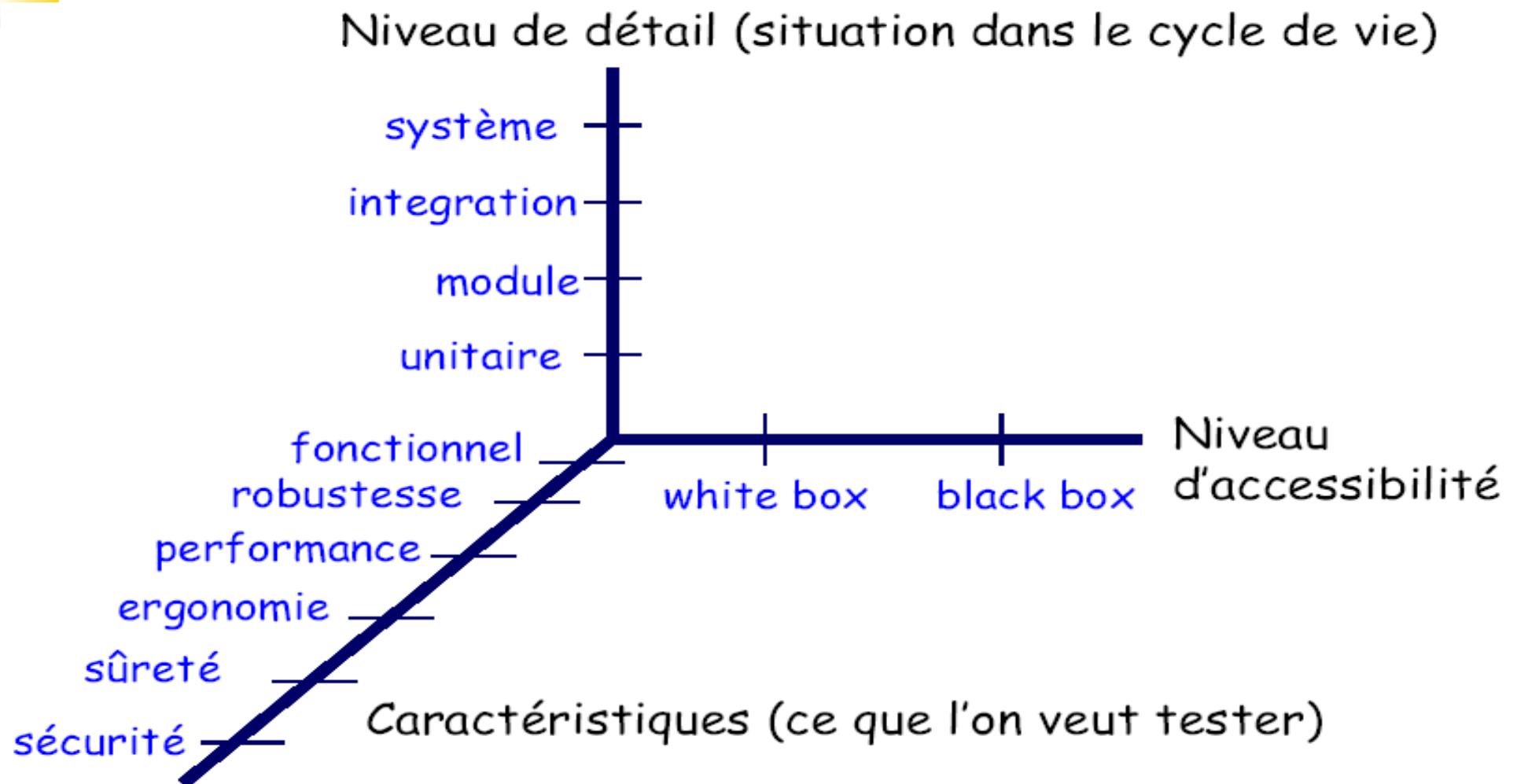
Caractéristiques d'un outil de build

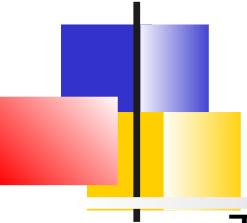
Tests et analyse statique

Release et dépôts d'artefacts



Typologie des test





Types de test

Test Unitaire :

Est-ce qu'une simple classe/méthode fonctionne correctement ?

JUnit, PHPUnit, UnitTest, Karma, Mockito

Test d'intégration :

Est-ce que plusieurs classes/couches fonctionnent ensemble ?

BD/Serveurs embarqués

Test fonctionnel :

Pour l'utilisateur final, est-ce que mon application fonctionne ?

Selenium, HttpUnit, Protractor

Test de performance :

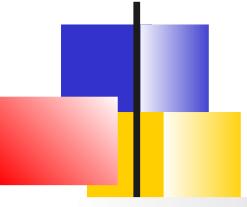
Est-ce que mon application fonctionne bien ?

JMeter, Gatling

Test d'acceptation :

Est-ce que mon client aime mon application ?

Cucumber



Classification pipeline DevOps

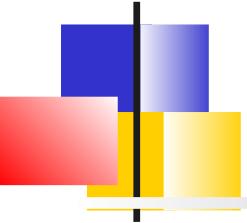
Dans le cadre d'une pipeline DevOps, classification en fonction de :

- Le test nécessite t il un provisionnement d'infrastructure ?
- Durée d'exécution

=> Conditionne la position des tests dans la pipeline

Typiquement :

- Tests unitaires/intégration en premier,
- Test fonctionnel, d'acceptation de performance sur les infrastructures d'intégration, recette, pré-production et de production

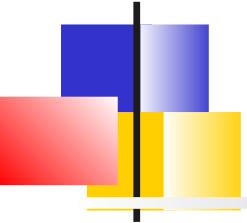


Analyse statique

L'analyse statique est l'analyse du code sans l'exécuter.

Les objectifs sont :

- La mise en évidence d'éventuelles erreurs de codage
- La vérification du respect du formatage convenu.
- La production de métriques adossés à la norme ISO25010 relatif à la qualité



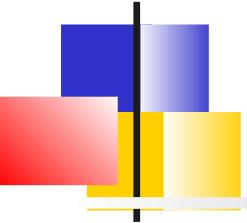
Métriques internes et Outils Qualité

SonarQube propose une plate-forme qui regroupe tous les outils de calcul de métriques interne d'un logiciel (toute technologie confondue)

Il intègre 2 aspects :

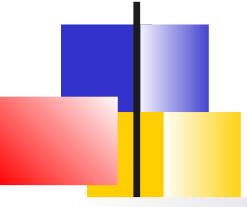
- Détection des transgressions de règles de codage et estimation de l'influence sur la fiabilité, la sécurité et la maintenabilité de l'application
- Calculs des métriques internes

Pour chaque projet, il est possible de définir des **portes qualités** : seuils minimal des métriques afin que le projet soit livrable



Outils de build

Caractéristiques d'un outil de build
Tests et analyse statique
Release et dépôts d'artefacts



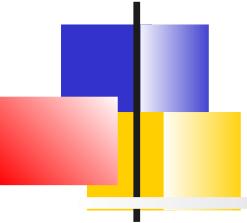
Introduction

Le but principal d'un outil de build est d'une construire un **artefact** : le format d'exécution du logiciel

Les artefacts sont stockés et taggés dans des **dépôts**

2 types d'artefacts :

- Les **SNAPSHOTS**, ils fournissent le dernier état (plutôt stable) de la version en cours de développement
- Les **releases**, ils sont immuables. Ce sont des versions stables et bien testées qui sont déployables en production
Généralement le processus de release consiste à :
 - Tagger le dépôt des sources : SCM
 - Produire un artefact et le stocker (à tout jamais) dans un repository d'artefact avec son n° de version

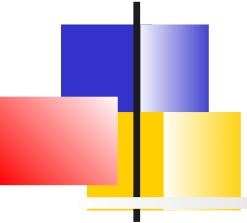


Production d'une release

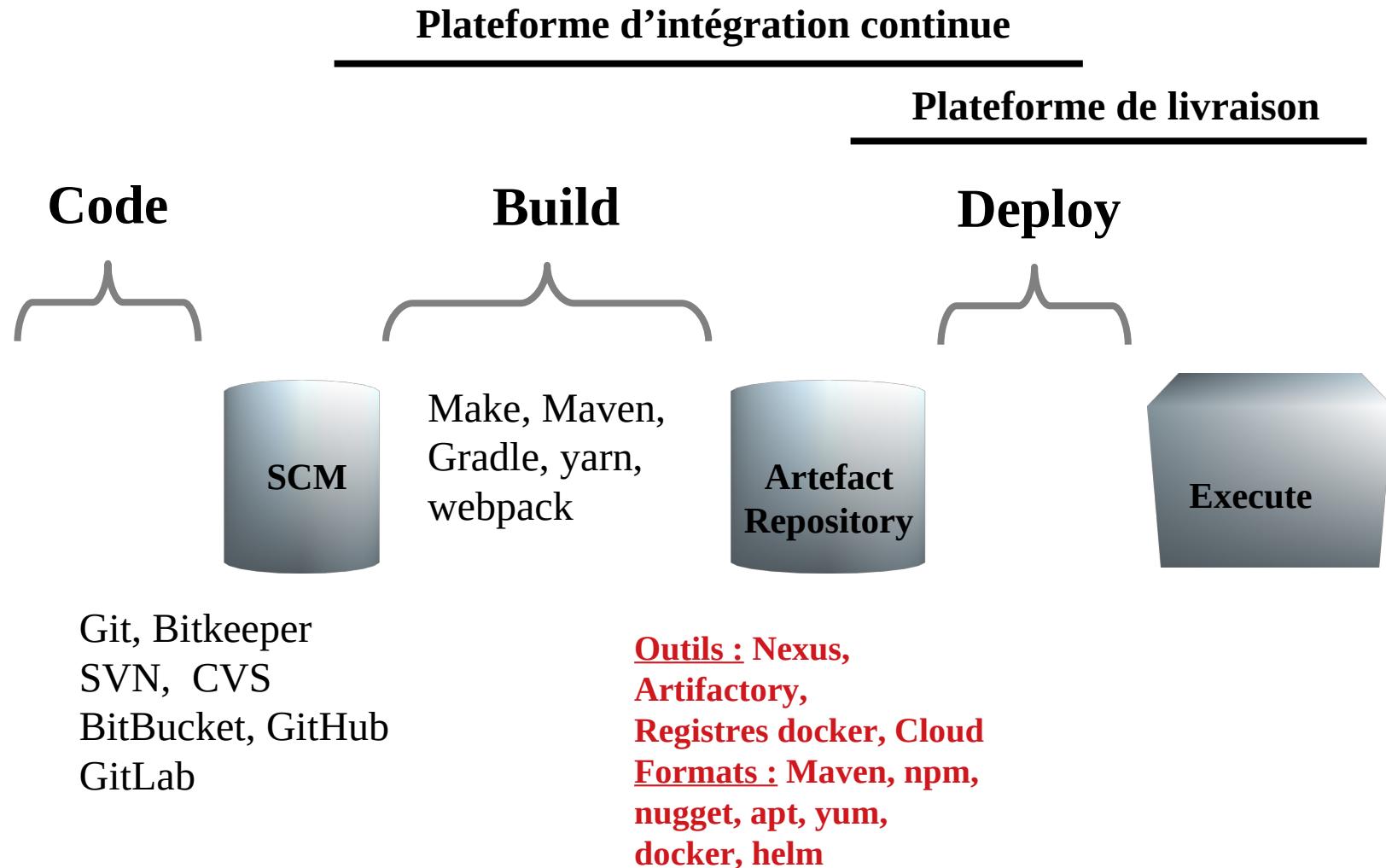
La processus de production d'une release consiste généralement en :

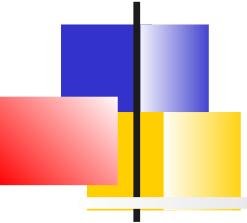
- Générer l'artefact et s'assurer que tous les tests sont OK
- Modifier le n° de version et intégrer la branche de travail dans la branche principale
- Tagger le commit avec un n° de version
- Publier l'artefact dans le dépôt d'artefact
- Incrémenter le n° version (et lui apposer le suffixe SNAPSHOT).

Ce processus doit également être automatisé.



Les gestionnaires de dépôts dans le cycle de vie



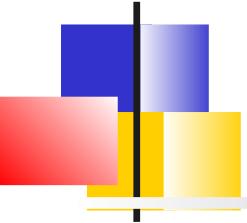


Nexus

Nexus est un gestionnaire de dépôt qui offre principalement 2 fonctionnalités :

- Proxy d'un dépôt distant (Ex : Maven central) et cache des artefacts téléchargés
- Hébergement de dépôts internes (privé à une organisation)

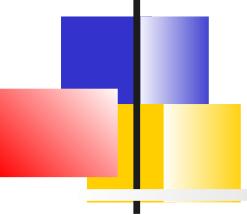
Nexus support de nombreux types de dépôts



Types de dépôt

Nexus permet de définir 3 différents types de dépôts :

- **Proxy** : Nexus se comporte alors proxy d'un dépôt distant (en général public) avec des fonctionnalités de cache
- **Hosted** : Nexus stocke des artefacts produits par l'entreprise
- **Group** : Permet de grouper sous une URL unique plusieurs dépôts

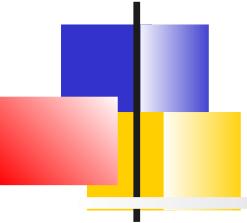


Installation par défaut

Exemple Maven

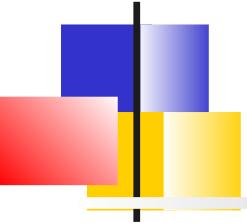
L'installation par défaut à déjà créé le dépôt groupe nommé **maven-public** qui regroupe :

- **maven-releases** : Pour stocker les releases internes
- **maven-snapshots** : Pour stocker les snapshots internes
- **maven-central** : Proxy de Maven central



Plateforme d'intégration continue

Concepts communs
Pipelines typiques
Solutions



PICs dans le Cycle de vie

Plateforme d'intégration continue

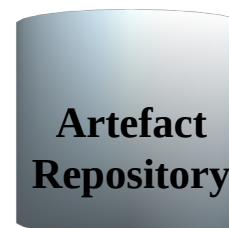
Jenkins, Gitlab CI, Travis CI, Strider, Azure DevOps

Plateforme de livraison

Code

Build

Deploy

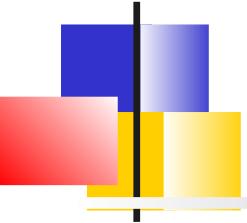


Git, Bitkeeper
SVN, CVS
BitBucket, GitHub
GitLab

Make, Maven,
Gradle, yarn,
webpack

Artefact
Repository

Nexus,
Artifactory,
Archiva



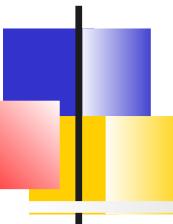
Plateforme d'intégration continue

Une PIC a pour objectifs :

- Automatiser les builds et les déploiements en intégration ou en production
- Fournir une information complète sur l'état du projet (état d'avancement, qualité du code, couverture des tests, métriques performances, documentation, etc.)

Il est en général multi-projets, multi-branches, multi-configuration

=> Il nécessite beaucoup de ressources



Architecture Maître / esclaves

Le serveur central distribue les jobs de build sur différentes ressources appelés les esclaves/runners/workers.

Les esclaves sont :

- Des **machines physiques, ou virtuelles** où sont préinstallés les outils nécessaires au build
- Des **conteneurs** qui sont alors provisionnés/exécutés lors du build et qui disparaissent ensuite

Netflix 2012

1 master avec 700 utilisateurs

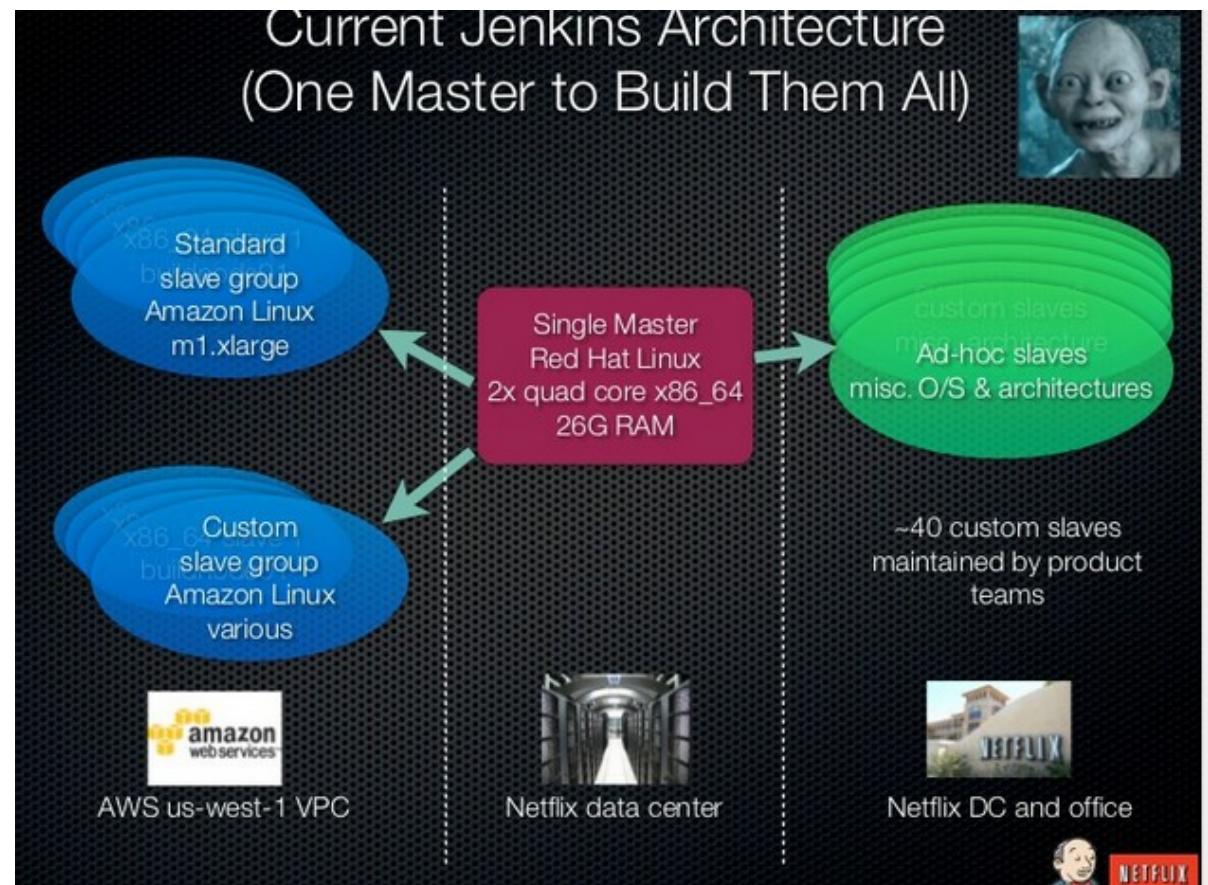
1,600 jobs :

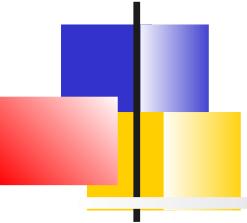
- 2,000 builds/jour
- 2 TB
- 15% build failures

=> 1 maître avec 26Go de RAM

=> Agent Linux sur amazon

=> Esclaves Mac. Dans le réseau interne



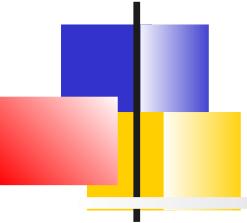


Outils annexes, plugins

La plateforme doit interagir avec de nombreux outils annexes : SCM, Outils de build, de test, plateforme de déploiement

Les outils peuvent être intégrés

- Directement par la solution
- Via des plugins
- En utilisant des CLI pré-installés sur les esclaves (Installation manuelle, automatique ou conteneur)

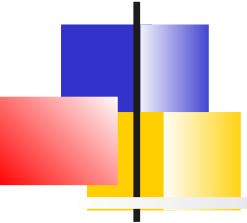


Déploiement

Le PIC a vocation à automatiser des déploiements dans différents environnements

Il doit donc pouvoir :

- Accéder aux environnements statiques.
Ex : dépôt de l'artefact via ftp, redémarrage d'un service
- Provisionner dynamiquement l'infrastructure. Ex : Création dynamique d'une machine virtuelle
- Déclencher la plateforme de déploiement.
Ex : Dépôt de l'image d'un conteneur dans un repository

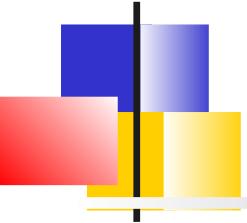


Approche standard

L'approche standard consiste à disposer d'une PIC centralisé.

Des administrateurs :

- installent les plugins nécessaires
- créent des jobs via l'UI et spécifient leur séquencement
- Exploitent et surveillent l'exécution des pipelines



Approche DevOps

Dans l'approche DevOps, la pipeline est décrite par un DSL

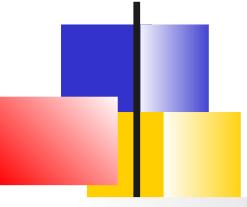
Le fichier de description est stocké dans le SCM, il est géré par l'équipe projet.

- La pipeline peut s'exécuter sans l'administrateur de la PIC

Ou dans certains cas, l'infrastructure de PIC nécessaire au projets (plugins et autre) est elle même conteneurisé et stocké dans le SCM



Pipelines typiques



Principes

Les pipelines sont généralement découpées en **phases** représentant les grandes étapes de la construction.

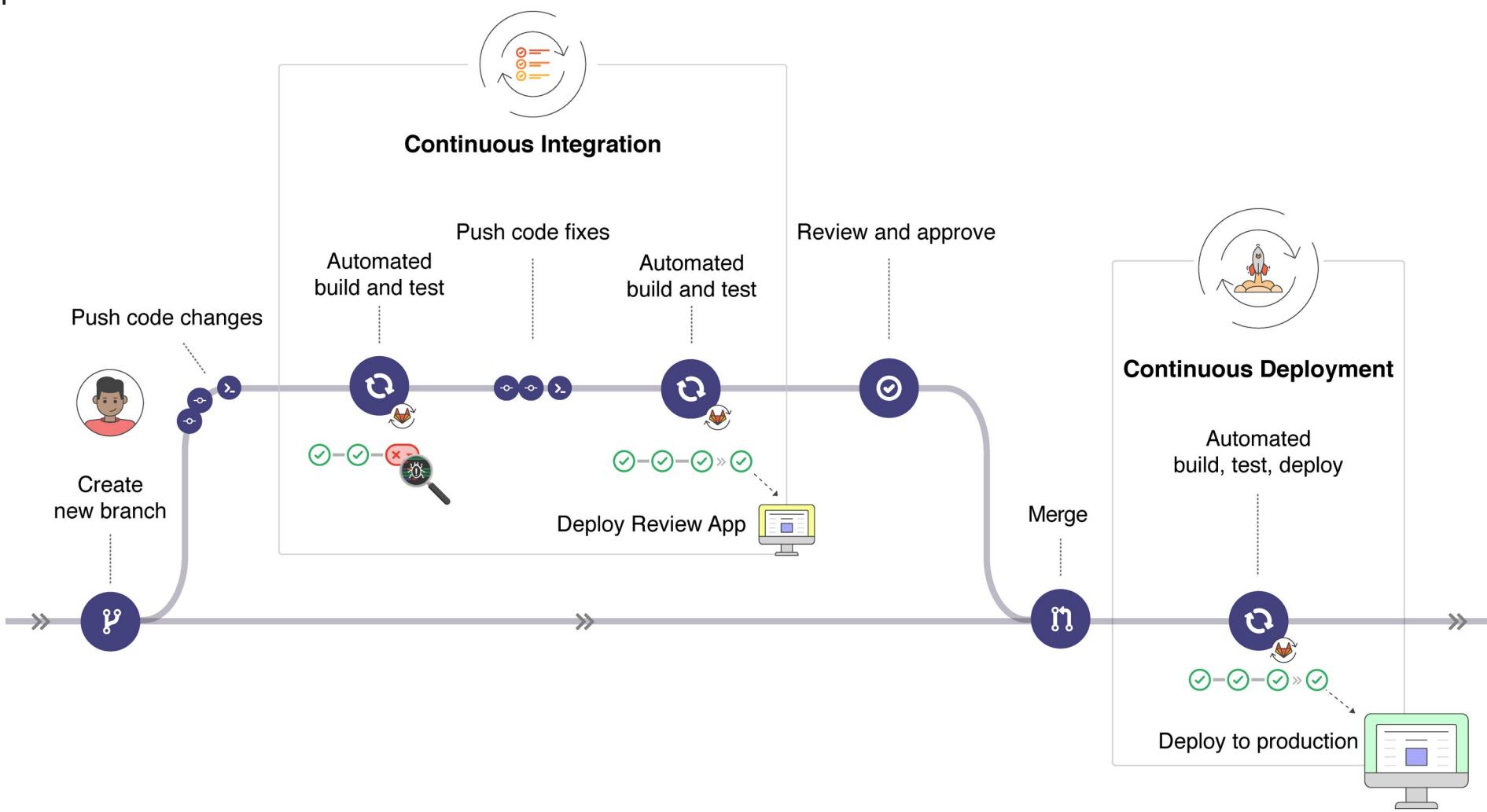
Par exemple : Build, Test, QA, Production

Chaque phase est constituée de tâches ou **jobs** exécutés séquentiellement ou en parallèle

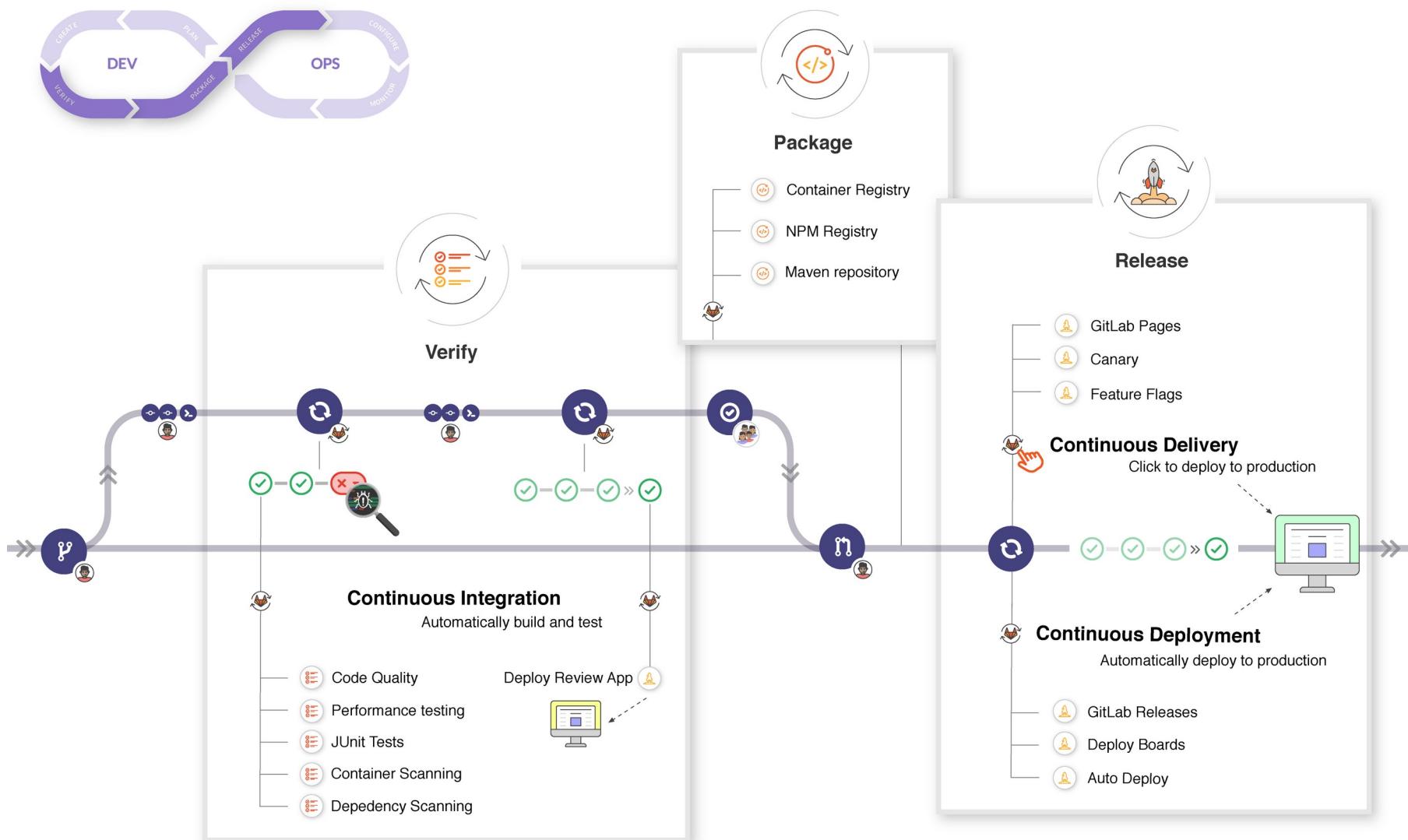
Les jobs exécutent des actions à partir du dépôt ou réutilisent des artefacts précédemment construits

Les jobs publient des rapports résultats

Exemple Gitlab CI

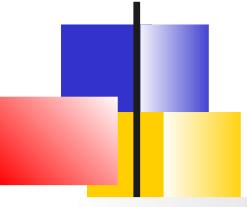


En plus détaillé





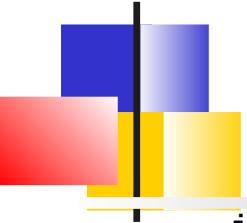
Solutions



Gitlab CI

GitLab propose désormais la gestion de constructions et/ou de tests via **Gitlab-CI**

- Développé en Go, il propose un mode 'server-runner'
- Les *runners* exécutent les pipelines. Les outils nécessaires pour le build sont pré-installés ou des images Docker sont utilisés.
- Les pipelines sont codés via un fichier au format YAML
- Elles s'exécutent sur toutes les branches de GitlabFlow.
- GitlabCI gère les déploiements sur les différents environnements et propose une intégration avec les clusters Kubernetes pour le déploiement



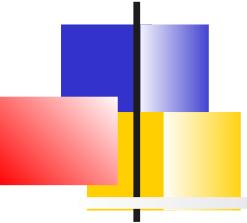
Exemple *.gitlab-ci.yml*

```
image: "ruby:2.5"

before_script:
  - apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
  - ruby -v
  - which ruby
  - gem install bundler --no-document
  - bundle install --jobs $(nproc) "${FLAGS[@]}"

rspec:
  script:
    - bundle exec rspec

rubocop:
  script:
    - bundle exec rubocop
```

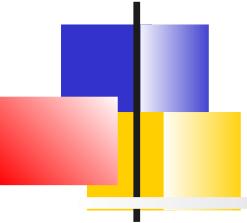


Gitlab CI

Running 0 Finished 327 All 327

List of finished builds from this project

Status	Build ID	Commit	Ref	Runner	Name	Duration	Finished at
✓ success	Build #351965	23b89d99	artifacts	golang-cross#1059	Bleeding Edge	6 minutes 4 seconds	about 19 hours ago
✓ success	Build #351548	634b6f5e	artifacts	golang-cross#1059	Bleeding Edge	5 minutes 43 seconds	about 22 hours ago
✓ success	Build #349948	56329a8e	artifacts	golang-cross#1059	Bleeding Edge	6 minutes 2 seconds	1 day ago
✓ success	Build #349883	c01876c1	master	golang-cross#1059	Bleeding Edge	5 minutes 39 seconds	1 day ago
✗ failed	Build #349807	623f3f5a	master	golang-cross#1059	Bleeding Edge	1 minute 50 seconds	1 day ago
✗ failed	Build #349804	338d0a8b	artifacts	golang-cross#1059	Bleeding Edge	1 minute 35 seconds	1 day ago



Jenkins

Anciennement Hudson, (fork depuis le rachat par Oracle)

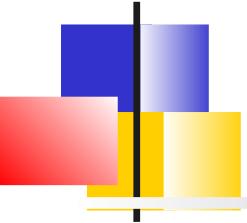
Encore, le plus répandu

Soutenu par la société CloudBees

De nombreux plugins disponibles :

- Plugins Legacy
- Plugin pipeline (DevOps) et visualisation des pipeline via Blue Ocean

Forte intégration avec Docker

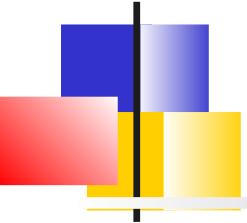


Approche DevOps

Dans la dernière version de Jenkins,
l'approche DevOps est permise.

Un fichier **Jenkinsfile** faisant partie
intégrante des sources du projet décrit la
pipeline de déploiement continu

- Le fichier est commité et versionné dans
un dépôt Git
- La description est effectuée via un langage
spécifique DSL construit avec Groovy



Illustration

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh './mvnw -Dmaven.test.failure.ignore=true clean test'
            } post {
                always { junit '**/target/surefire-reports/*.xml'      }
            }
        }
        stage('Parallel Stage') {
            parallel {
                stage('Intégration test') {
                    agent any
                    steps {
                        sh './mvnw clean integration-test'
                    }
                }
                stage('Quality analysis') {
                    agent any
                    steps {
                        sh './mvnw clean verify sonar:sonar'
                    }
                }
            }
        }
    }
}
```

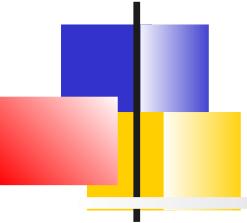


Tableau de bord (multi-branches)

Jenkins

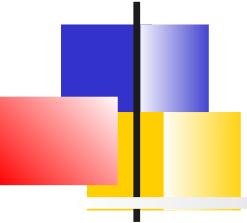
Pipelines Administration Déconnexion

Tableau de bord Nouveau Pipeline

Favorites

✓ plbsi-angular	master	#5c3e3ac	2 months ago	↻ ↻ ★
✓ plbsi-angular	develop	#d07b3cb	4 months ago	↻ ↻ ★

Nom	Santé	Branches	Pull requests
AutomaticDeployDev	🟡	-	- ★
DeployPrevious	🟡	-	- ★
DeployProduction	🟡	-	- ★
MinimalCheck	⚡	-	- ★
plbsi-angular	🟡	1 en échec	★
TestManual	🟡	-	★



Déploiement

Considérations

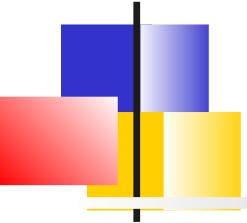
Virtualisation

Outils de gestion de configuration

Containerisation. Le cas docker

Orchestrateurs de conteneur :
Kubernetes

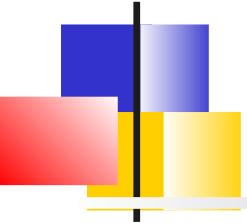
Kubernetes dans la pipeline CD



Introduction

2 décisions importantes concernant l'infra :

- L'architecture applicative : Applications monolithiques ou micro-services
- Format des artefacts : Déploiements mutables ou immuables

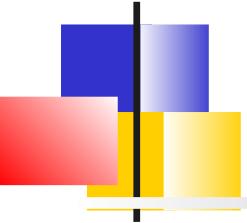


Modèle classique

Le serveur monstre mutable

Un serveur Web qui contient toute l'application et mis à jour à chaque déploiement.

- Fichiers de configuration, Artefacts, schémas base de données.
=> il mute au fur et à mesure des déploiements.
=> On n'est plus sur que les environnements de dév, de QA ou même les instances en production soient identiques
=> Difficulté de revenir à une version précédente



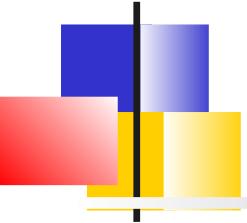
Modèle DevOps

Serveur immuable et Reverse Proxy

L'approche DevOps s'appuie sur les déploiements immuables qui garantissent que chaque instance déployée est exactement identique.

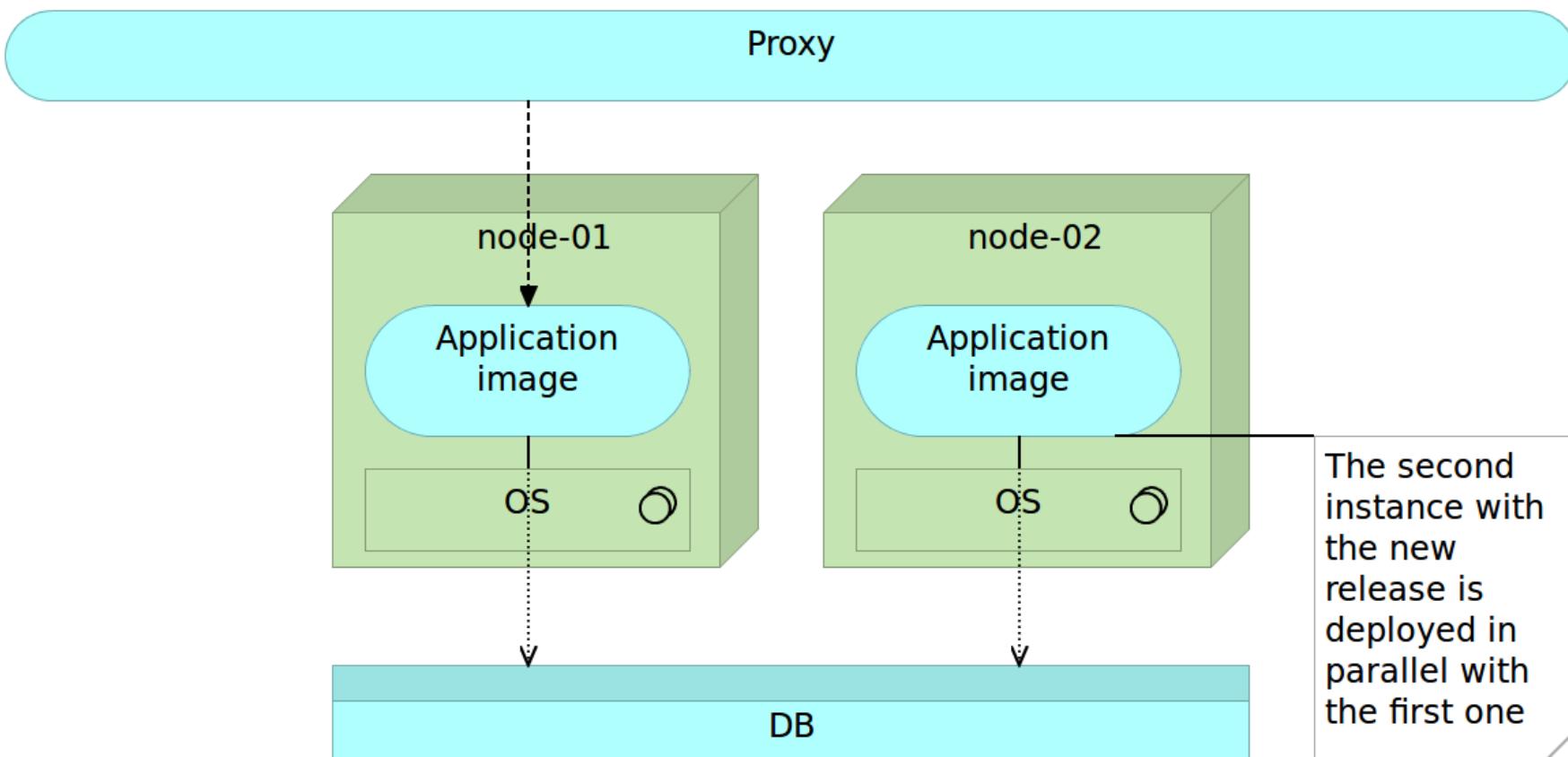
Un package immuable contient tout : serveur d'applications, configurations et artefacts

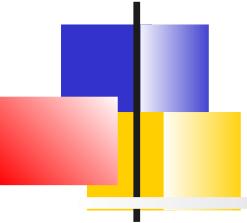
C'est ce tout qui est déployé



Modèle DevOps

Serveur immuable et Reverse Proxy



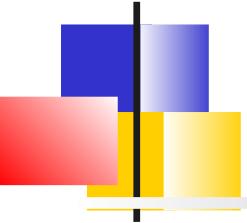


Architecture micro-services

La durée du déploiement dépend principalement de :

- La durée de l'instanciation du déploiement immuable
- Des tests de post-déploiement

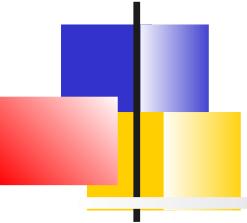
Une architecture micro-services basée sur des containers optimise énormément ces temps, permettant d'augmenter la fréquence de déploiement et de repli
=> scalabilité dynamique, serverless



Déploiement

Considérations **Virtualisation**

Outils de gestion de configuration
Containerisation. Le cas docker
Orchestrateurs de conteneur :
Kubernetes
Kubernetes dans la pipeline CD

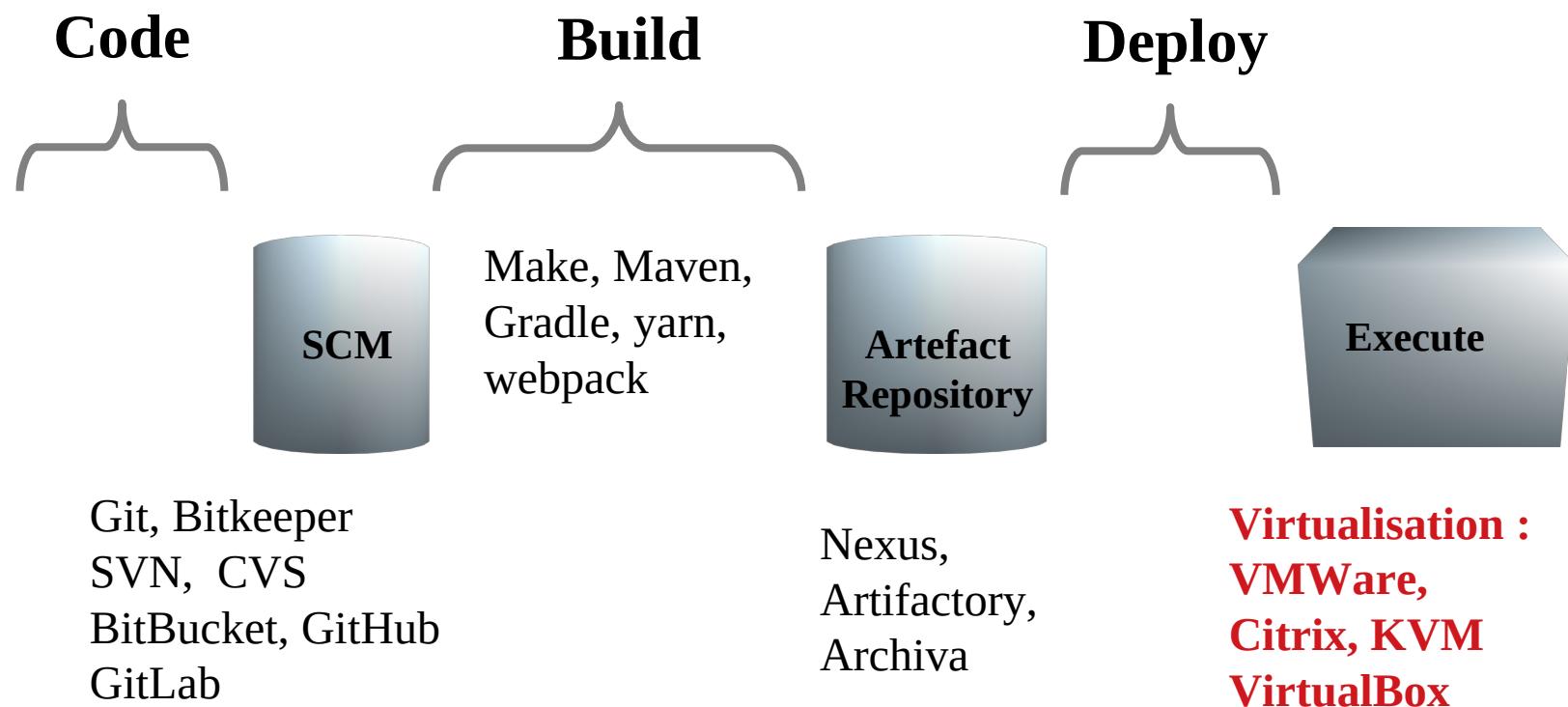


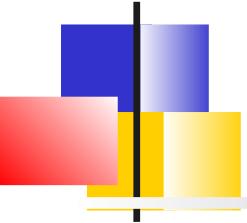
La virtualisation dans le cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

Plateforme de livraison



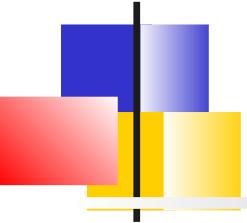


Hyperviseur

Les **hyperviseurs** permettent à une machine nue de superviser plusieurs machines virtuelles.

Les déploiements peuvent ainsi être isolés sur des environnements dédiés.

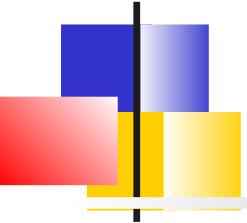
Bien que le serveur soit virtualisé, le principe reste équivalent à celui d'un serveur dédié.



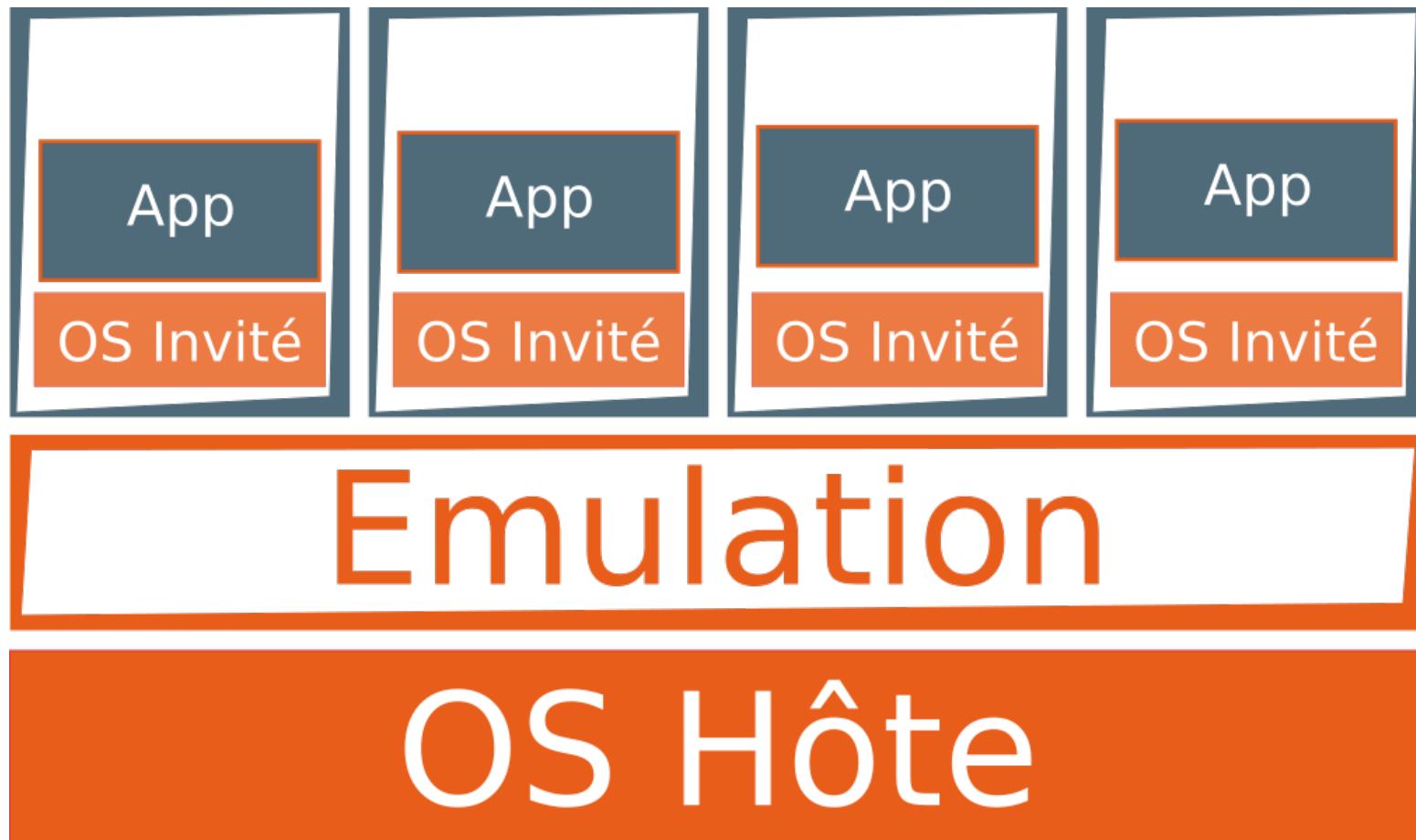
Limitations

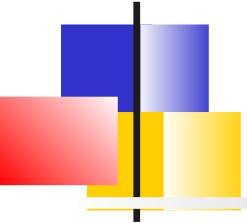
Pour virtualiser un environnement complet
le serveur doit être capable

- de supporter la charge de son OS hôte ainsi que les OS invités
 - + l’émulation du matériel (CPU, mémoire, carte vidéo...) et la couche réseau associée.
- => Les ressources nécessaires pour exécuter une Machine Virtuelle sont importantes.



Hôte / invité



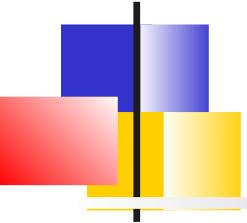


Solutions

Les principales solutions commerciales sont :

- **Microsoft Hyper-V**,
- **VMware vSphere** : Virtualisation + de nombreux autres services
- **Citrix XenServer** : Version OpenSource
- **Red Hat KVM** : Intégré dans le noyau Linux depuis 2.6.20

A un plus petit niveau, Oracle VirtualBox

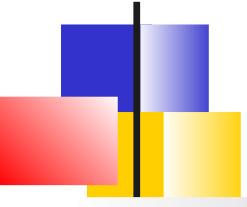


Vagrant

Vagrant est un outil permettant de gérer les machines virtuelles.

Il permet de configurer des machines virtuelles avec de simple fichier (*Vagrantfile*) et ainsi d'automatiser la configuration et le provisionnement

Il est compatible avec plusieurs de virtualisation : VirtualBox, VMware, AWS, ou autre

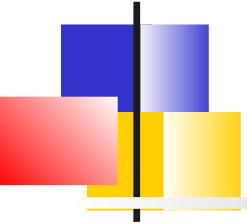


Projet Vagrant

Un projet consiste en la mise au point d'un fichier **Vagrantfile** (committé dans le SCM) qui :

- Marque la racine du projet.
- Décrit la machine, les ressources, les logiciel et les modes d'accès

On peut alors s'appuyer sur des **vagrant box** qui définissent des machines de base téléchargeables automatiquement

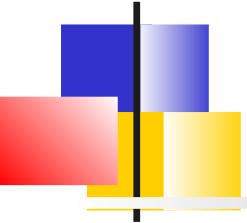


Configuration de la machine virtuelle

Par défaut, le répertoire contenant le *Vagrantfile* est monté sur le répertoire */vagrant* de la machine virtuelle.

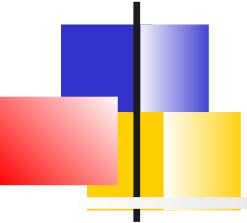
=> Il est possible d'y positionner des scripts permettant de provisionner la box

Les autres directives de configuration permettent de faire du mapping de port , d'affecter une IP fixe, ou la raccorder à une réseau existant, de dimensionner la mémoire, etc..



Exemple

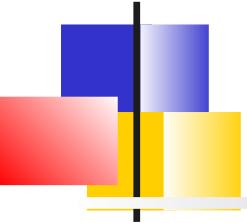
```
Vagrant.configure("2") do |config|  
  
  config.vm.define "db" do |db|  
    db.vm.provider "virtualbox" do |v|  
      v.memory = 2048  
      v.cpus = 1  
      v.name = "db"  
    end  
    db.vm.box = "ubuntu/bionic64"  
    db.vm.provision :shell, path: "postgres.sh"  
    db.vm.network "private_network", ip: "192.168.10.3"  
    db.vm.network :forwarded_port, guest: 5432, host: 5444  
  end  
  
  config.vm.define "spring" do |spring|  
    spring.vm.box = "ubuntu/bionic64"  
    spring.vm.provision :shell, path: "spring.sh"  
    spring.vm.network "private_network", ip: "192.168.10.2"  
    spring.vm.network :forwarded_port, guest: 8080, host: 8000  
  end  
end
```



Principales commandes

Le client vagrant permet alors de nombreuses commandes :

- **up, halt, suspend, resume, destroy** : Démarrage, arrêt, pause, reprise, suppression de toutes les traces
- **provision** : Met à jour les logiciels sur une machine s'exécutant
- **ssh, powershell, rdp** : Accès distant sur la machine
- **box** (add,list, remove) : Gestion des vagrants box
- **snapshot** : Sauvegarde d'une machine s'exécutant
- **push** : Pousser la configuration sur un serveur FTP ou autre



Déploiement

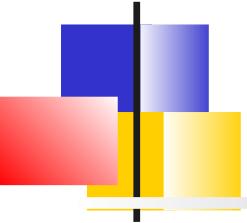
Considérations
Virtualisation

Outils de gestion de configuration

Containerisation. Le cas docker

Orchestrateurs de conteneur :
Kubernetes

Kubernetes dans la pipeline CD

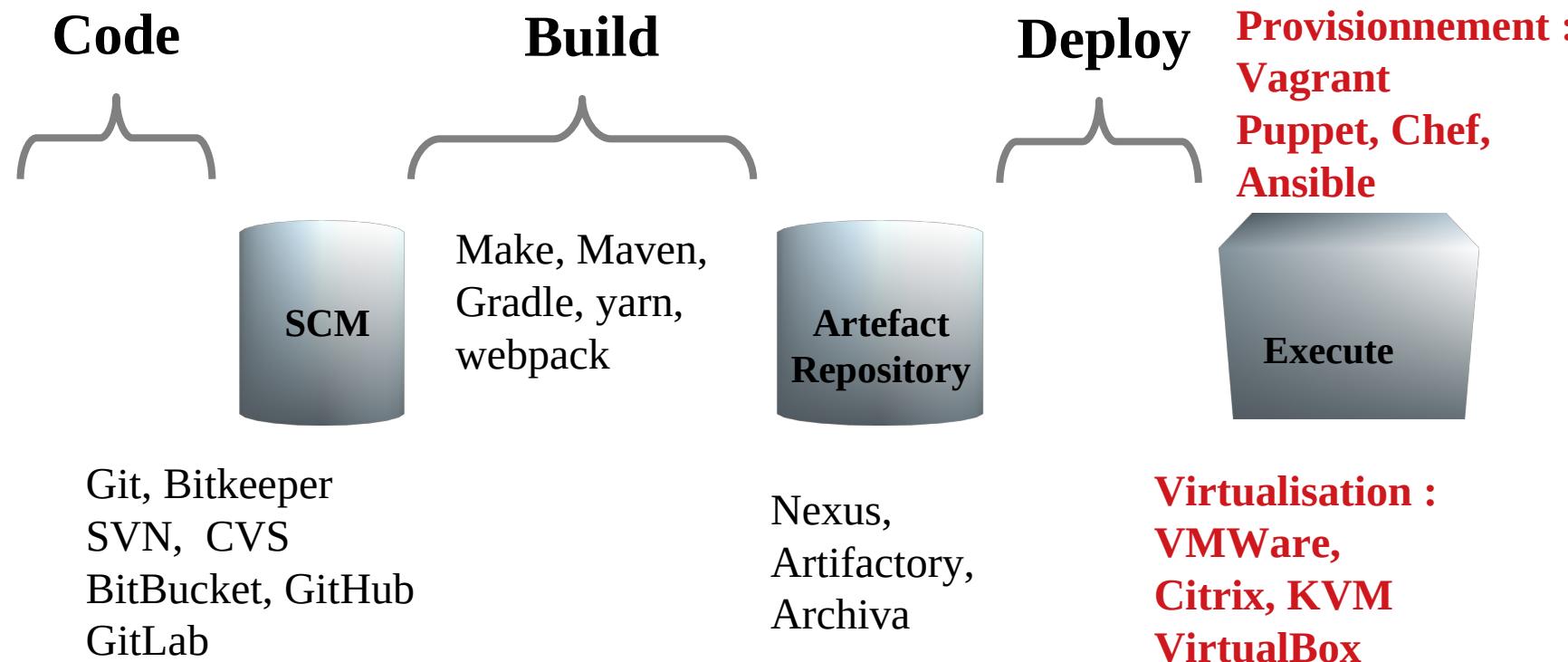


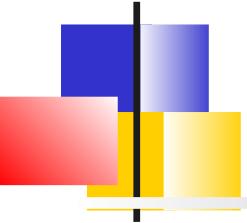
Provisionnement dans le cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

Plateforme de livraison



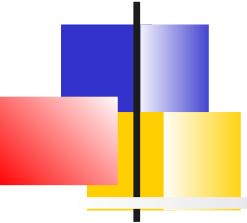


Infrastructure As Code

Pour gérer les configurations des machines virtuels (configuration réseau, sécurité, utilisateur ayant accès, etc.), des solutions sont apparues.

Elles permettent de décrire dans des fichiers scripts, les opérations de configuration .

- Outils : Chef, Puppet, Ansible, SaltStack
- Langages : Ruby, Python, DSL

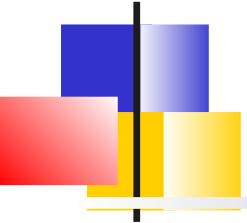


Gestion de déploiements

Les services de déploiements permettent d'intégrer une version sortie de la PIC sur un environnement de production

2 approches existent :

- L'approche centralisé nécessitent l'installation d'agents sur les serveurs cibles services. Un serveur centralise et orchestre les déploiements
Ex : *Chef, Puppet*
- Les services "agentless" ne nécessite pas de pré-installation et se base généralement sur *ssh*. Pendant, l'opération de déploiement des modules sont installés temporairement sur la machine cible.
Ex : *Ansible*

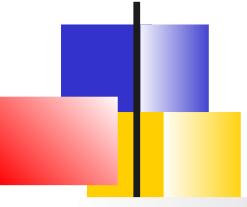


Ansible

Ansible est un moteur d'automatisation de déploiement et de provisionnement.

L'un des intérêts majeurs de Ansible est la "non utilisation d'agent", en d'autres termes les machines cibles n'ont pas besoin d'avoir de services toujours up

- Le seul pré-requis est l'installation de Python et il n'y a pas de support pour Windows



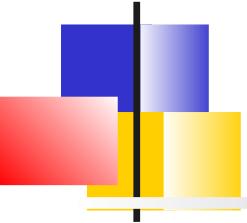
Fonctionnement

Ansible se connecte (via SSH par défaut) aux différents nœuds et y poussent de petits programmes, nommés "**modules Ansible**"

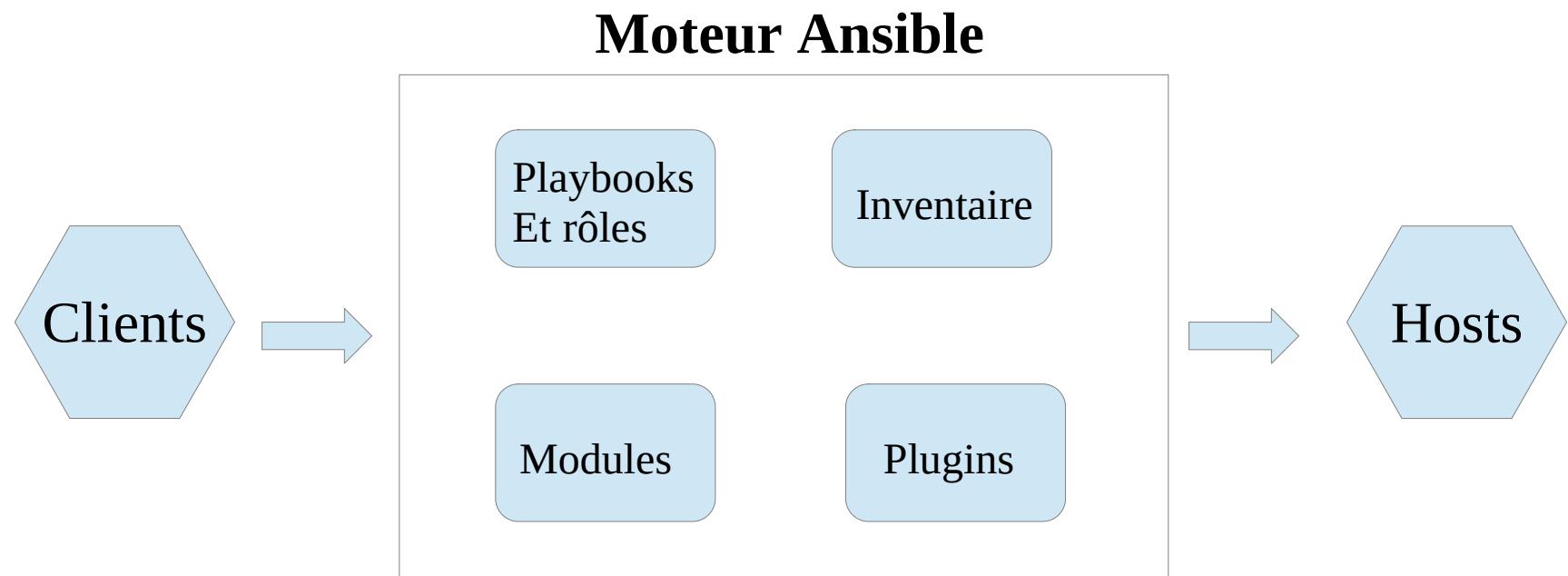
- Ansible exécute les modules et les enlève une fois l'exécution terminée
- Les modules Ansible peuvent être écrits dans n'importe quel langage du moment qu'ils retournent du JSON. Ils sont idempotents.

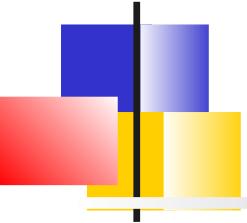
Ex :

```
ansible all -s -m apt -a 'pkg=nginx  
state=installed update_cache=true'
```



Architecture



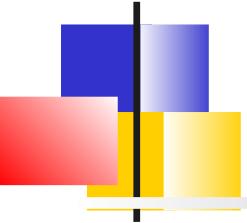


Inventaire

Ansible représente les machines qu'il gère via **l'inventaire**

Par défaut, un simple fichier texte de type INI mais peut être une autre source de données.

L'inventaire permet également de définir des **groupes de machine**, de leur assigner des variables



Exemple Inventaire

mail.example.com

[webservers]

foo.example.com

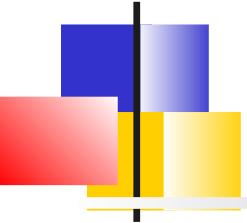
bar.example.com

[dbservers]

one.example.com

two.example.com

three.example.com



Principales commandes

ansible : Exécute une tâche sur un ou plusieurs hosts

ansible-playbook : Exécute un playbook

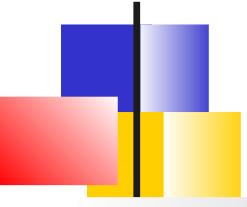
ansible-doc : Affiche la documentation

ansible-vault : Gère les fichiers cryptés

ansible-galaxy : Gère les rôles avec galaxy.ansible.com

ansible-pull : Exécute un playbook à partir d'un SCM

...



Commande en ligne

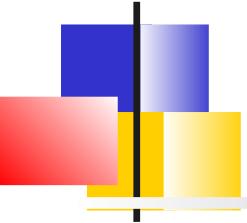
ansible : Exécute une tâche sur un ensemble de host

```
ansible all -m ping
```

```
ansible <HOST_GROUP> -m authorized_key -a "user=root  
key='ssh-rsa AAAA...XXX == root@hostname'"
```

```
ansible -m raw -s -a "yum install libselinux-python -  
y" new-atmo-images
```

D'autres commandes en ligne sont disponible :
ansible-config, *ansible-console*, *ansible-pull*, ...

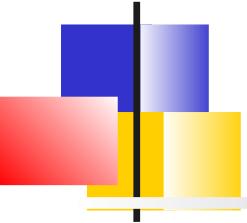


Playbooks

Les **playbooks** définissent ce qui doit être appliqué sur les serveurs cibles.

Ils déclarent des configurations mais peuvent également orchestrer les étapes d'un déploiement faisant intervenir différentes machines.

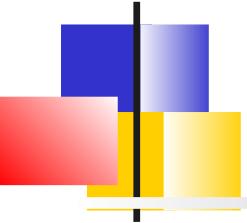
Ils sont commités dans le SCM



Configuration d'un déploiement

La configuration présente dans un *playbook* est constituée :

- Des **tâches**. Une tâche peut effectuer plusieurs opérations en utilisant un module Ansible.
- **Handler** s'exécute à la fin d'une série de tâches si un certain événement a été émis
- **Variables** : Valeurs dynamiques provenant de différentes sources
- **Gabarits** (Jinja2) : Fichiers variabilisés (Test et boucles disponibles).
- **Roles** sont des abstractions réutilisables qui permettent de grouper des tâches, variables, handlers, etc.



Exemple

```
---
```

- hosts: webservers
 remote_user: root

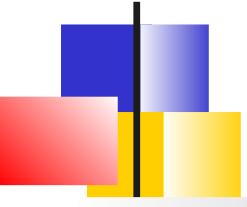

```
tasks:
```

- name: ensure apache is at the latest version
 yum:
 name: httpd
 state: latest
- name: write the apache config file
 template:
 src: /srv/httpd.j2
 dest: /etc/httpd.conf

- hosts: databases
 remote_user: root


```
tasks:
```

- name: ensure postgresql is at the latest version
 yum:
 name: postgresql
 state: latest
- name: ensure that postgresql is started
 service:
 name: postgresql
 state: started

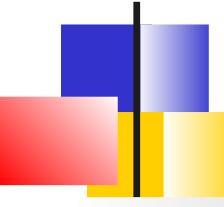


Rôles

Les **rôles** permettent d'organiser des tâches dépendantes et de permettre la réutilisation

Un rôle est un « *sous-playbook* », il peut contenir :

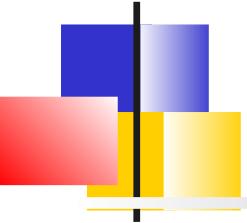
- **tasks** : Appels de modules Ansible
- **variables** : Des valeurs dynamiques positionnées lors de l'utilisation du rôle
- **template** : Des gabarits de fichiers (de configuration)
- **files** : Les fichiers à copier sur la cible
- **Handlers** : **Modules réagissant à un état**



Arborescence classique

Au final, un projet *Ansible* contient principalement des rôles qui sont réutilisés dans des playbooks

```
ansible.cfg
playbook1.yml
playbook2.yml
roles
  role1
    files
      file1
      file2
    tasks
      Main.yml
    templates
      Template1.j2
      Template2.j2
```



Déploiement

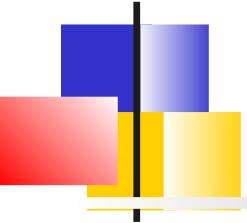
Considérations
Virtualisation

Outils de gestion de configuration

Containerisation. Le cas docker

Orchestrateurs de conteneur :
Kubernetes

Kubernetes dans la pipeline CD



Introduction

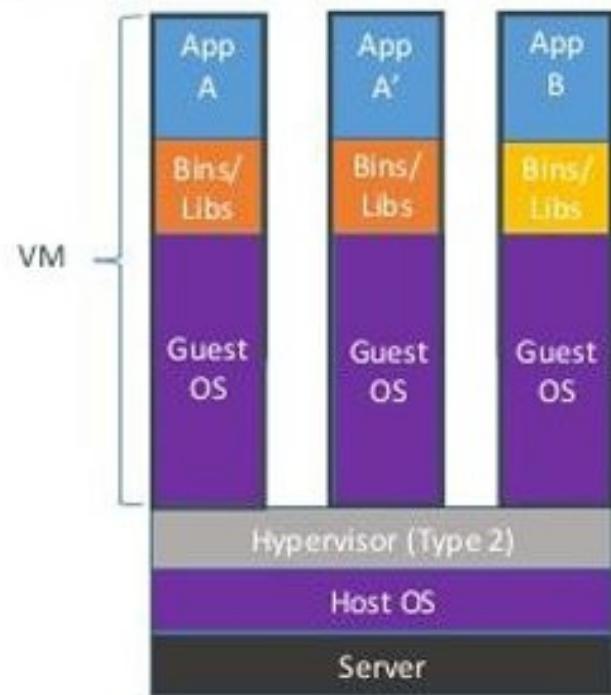
Plutôt que de virtualiser une machine complète, juste créer l'environnement d'exécution minimal pour fournir l'application, le service.

- il n'est plus question de simuler le matériel et les services d'initialisation du système d'exploitation sont ignorés.
- Seul le strict nécessaire réside dans le conteneur : l'application cible et quelques dépendances.

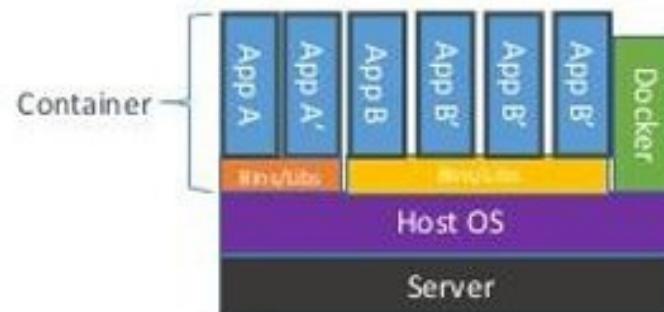
L'apport d'une solution de containerisation est l'isolation d'un processus dans un système de fichiers à part entière

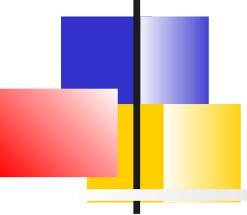
Containers vs VMs

Containers vs. VMs



Containers are isolated,
but share OS and, where
appropriate, bins/libraries





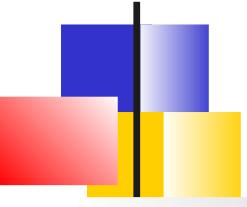
Avantages de la containerisation

Rationalisation des ressources, à la différence de la virtualisation, seuls ce dont on se sert est chargé !

Chargement du container 50 fois plus rapide que le démarrage d'une VM

A ressources identiques, nb d'applications multipliées par 5 à 80.

Permet l'avènement des architecture micro-services (application composée de nombreux services/container)



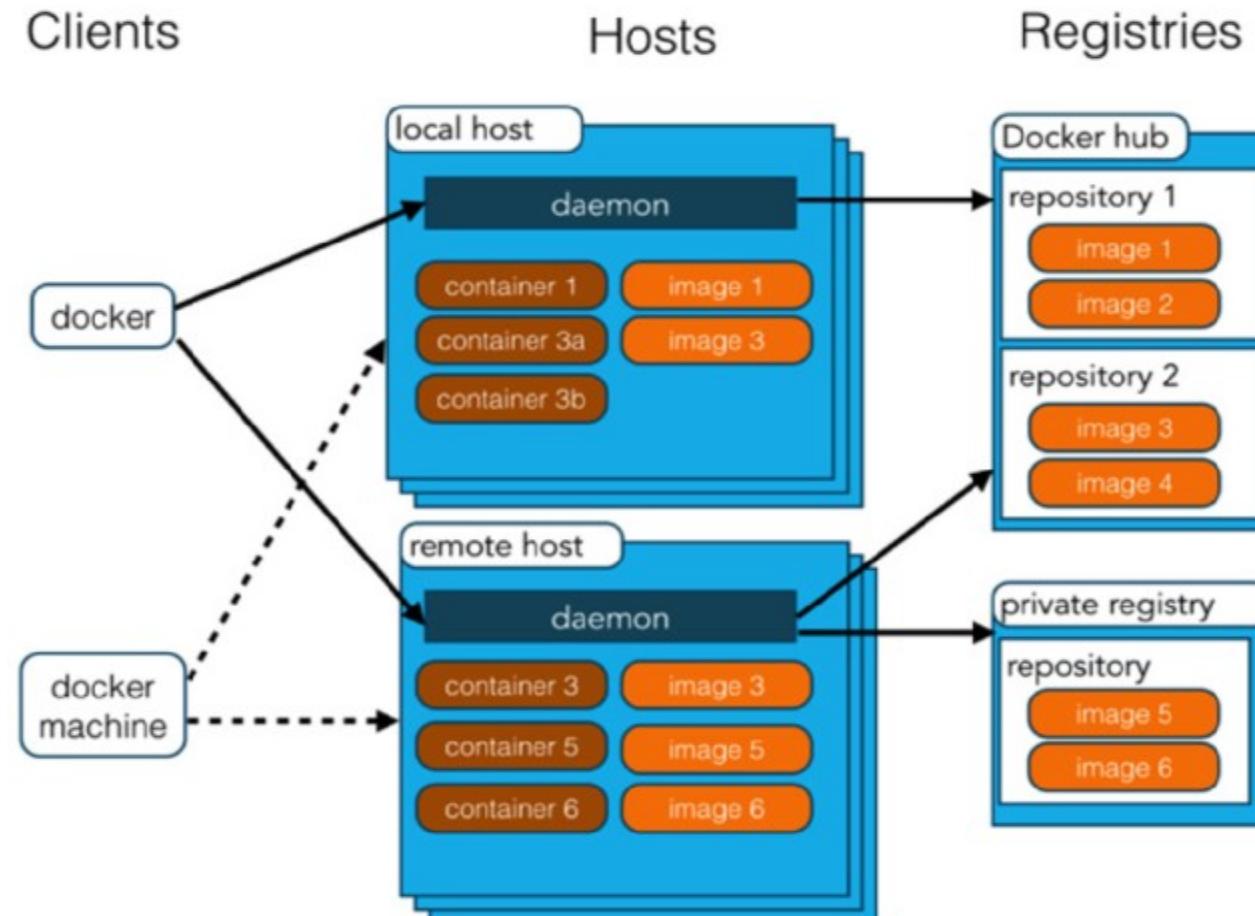
Impact sur le déploiement

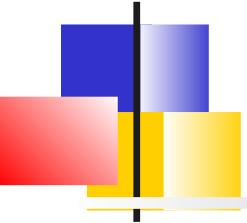
Les développeurs et la PIC travaillent alors avec la même image de conteneur que celle utilisée en production.

- => Réduction considérable du risque de dysfonctionnements dû à une différence de configuration logicielle.

Il n'y a plus à proprement parler un déploiement brut sur un serveur mais plutôt l'utilisation d'orchestrateur de conteneurs.

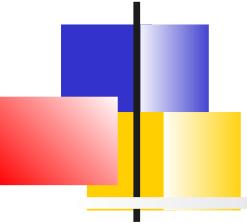
Docker architecture





Commandes Docker

```
#Récupération d'une image
docker pull ubuntu
#Récupération et instantiation d'un conteneur
docker run hello-world
#Mode interactif
docker run -i -t ubuntu
#Visualiser les sortie standard d'un conteneur
docker logs <container_id>
#Conteneurs en cours
docker ps
#Toutes les exécutions de conteneurs (même arrêt)
docker ps -a
#Lister les images
docker images
#Construire une image à partir d'un fichier Dockerfile
docker build . -t monImage
#Committer les différences
docker commit <container_id> <image_name>
#Tagger une image d'un repository
docker tag <image_name>[:tag] <name>[:tag]
#Pousser vers un dépôt distant
docker push <image_name>[:tag]
#Statistiques d'usage des ressources
docker stats
```

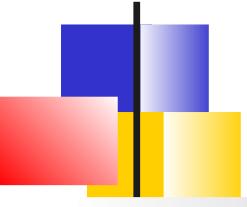


Exemple DockerFile

```
FROM ubuntu
MAINTAINER Kimbro Staken

RUN apt-get install -y software-properties-common python
RUN add-apt-repository ppa:chris-lea/node.js
RUN echo "deb http://us.archive.ubuntu.com/ubuntu/ precise
universe" >> /etc/apt/sources.list
RUN apt-get update
RUN apt-get install -y nodejs
RUN mkdir /var/www

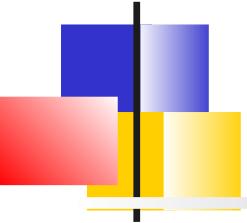
ADD app.js /var/www/app.js
EXPOSE 8080
CMD ["/usr/bin/node", "/var/www/app.js"]
```



Construction d'image

La construction d'image est de plus en plus intégré dans les outils de build :

- **Spotify Maven Plugin** nécessite un Dockerfile et ajoute des objectifs permettant d'automatiser la construction de l'image dans le build Maven
- **Spring Boot plugin** permet d'intégrer la construction d'image directement à partir des sources du projet en s'appuyant sur les build packs de CloudFoundry
- **Palantir Gradle** est capable de générer un Dockerfile ou d'utiliser celui que l'on fournit
- **Jib** est également un projet Google qui permet de construire des images optimisées (Docker ou OCI) sans Docker installé

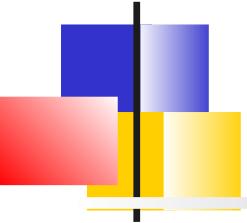


Isolation du conteneur

Chaque conteneur s'exécutant a sa propre interface réseau, son propre système de fichiers (gérés par Docker)

Par défaut, Il est isolé

- De la machine hôtes
 - => Montage de répertoires, association de ports TCP
- Des autres containers
 - => docker-compose et définition de réseau

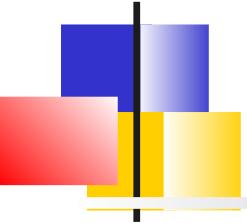


Communication avec la machine hôte

A l'instanciation d'un conteneur on peut :

- Associer un port exposé par le conteneur à un port local
Option **-p**
- Monté un répertoire du conteneur sur le système de fichier local.
Option **-v**

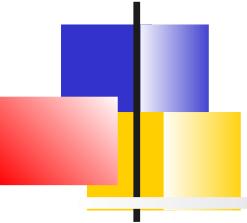
```
docker run -p 80:8080
           -v /home/jenkins:/var/lib/jenkins
           myImage
```



docker-compose

docker-compose est un outil pour définir et exécuter des applications Docker utilisant plusieurs conteneurs

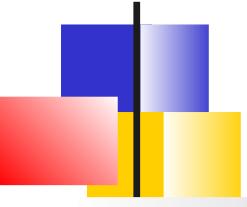
- Avec un simple fichier, on spécifie les différents conteneurs, les ports exposés, les liens entre conteneurs.
- Ensuite avec une commande unique, on peut démarrer, arrêter, redémarrer l'ensemble des services.



Exemple configuration

```
# Le fichier de configuration définit des services, des networks et des volumes.
version: '2'
services:
  annuaire:
    build: ./annuaire/ # context de build, présence d'un Dockerfile
    networks:
      - back
      - front
    ports:
      - "1111:1111" # Exposition de port
  documentservice:
    build: ./documentService/
    networks:
      - back
  proxy:
    build: ./proxy/
    networks:
      - front
    ports:
      - 8080:8080

# Analogue à 'docker network create'
networks:
  back:
  front:
```



Commandes

build : Construire ou reconstruire les images

config : Valide le fichier de configuration

down : Stoppe et supprime les conteneurs

exec : Exécute une commande dans un container up

logs : Visualise la sortie standard

port : Affiche le port public d'une association de port

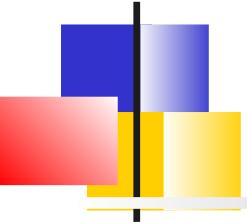
pull / push : Pull/push les images des services

restart : Redémarrage des services

scale : Fixe le nombre de container pour un service

start / stop : Démarrage/arrêt des services

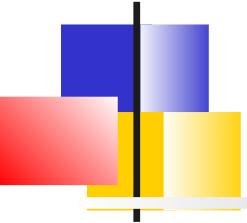
up : Création et démarrage de conteneurs



PIC et Docker

Plusieurs cas d'usage de Docker dans un contexte Jenkins :

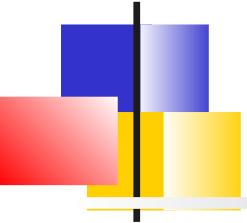
- Utiliser des images pour exécuter les builds
- Construire et pousser des images pendant l'exécution d'une pipeline
- Utiliser des images pour exécuter des services nécessaires à une étape de build (Démarrer un serveur lors de test d'intégration/fonctionnel)
- Dockeriser des configurations Jenkins



Techniques d'intégration

L'intégration peut nécessiter :

- Disposer d'un accès à un orchestrateur de conteneur (Cloud, Kubernetes) ou Préinstaller *docker* sur les nœuds esclaves ou de
- Déclarer des registres d'images et les crédentiels pour y accéder
- Permettre du *docker in docker*. (Une container de build démarre un autre container).
Utilisation de l'image *dind*



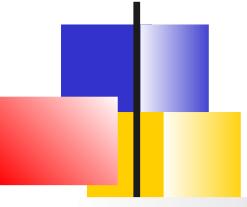
Application dockérisée

Un scénario désormais classique du CI/CD est:

- 1) Créer une image applicative
- 2) Exécuter des tests sur cette image
- 3) Pousser l'image vers un registre distant
- 4) Déployer l'image vers un serveur

En commande docker :

```
docker build -t my-image dockerfiles/  
docker run my-image /script/to/run/tests  
docker tag my-image my-registry:5000/my-image  
docker push my-registry:5000/my-image
```



Exemple Jenkins

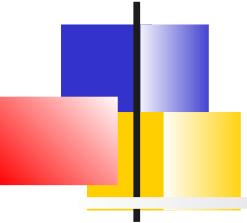
```
#!/usr/bin/env groovy

node {
    stage('checkout') {
        checkout scm
    }

    ..

    def dockerImage
    stage('build docker') {
        dockerImage = docker.build("dthibau/catalog", ".")
    }

    stage('publish docker') {
        docker.withRegistry('https://registry.hub.docker.com', 'docker-login') {
            dockerImage.push 'latest'
        }
    }
}
```



Déploiement

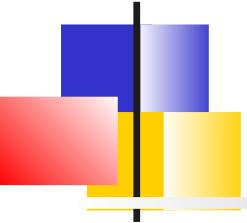
Considérations
Virtualisation

Outils de gestion de configuration

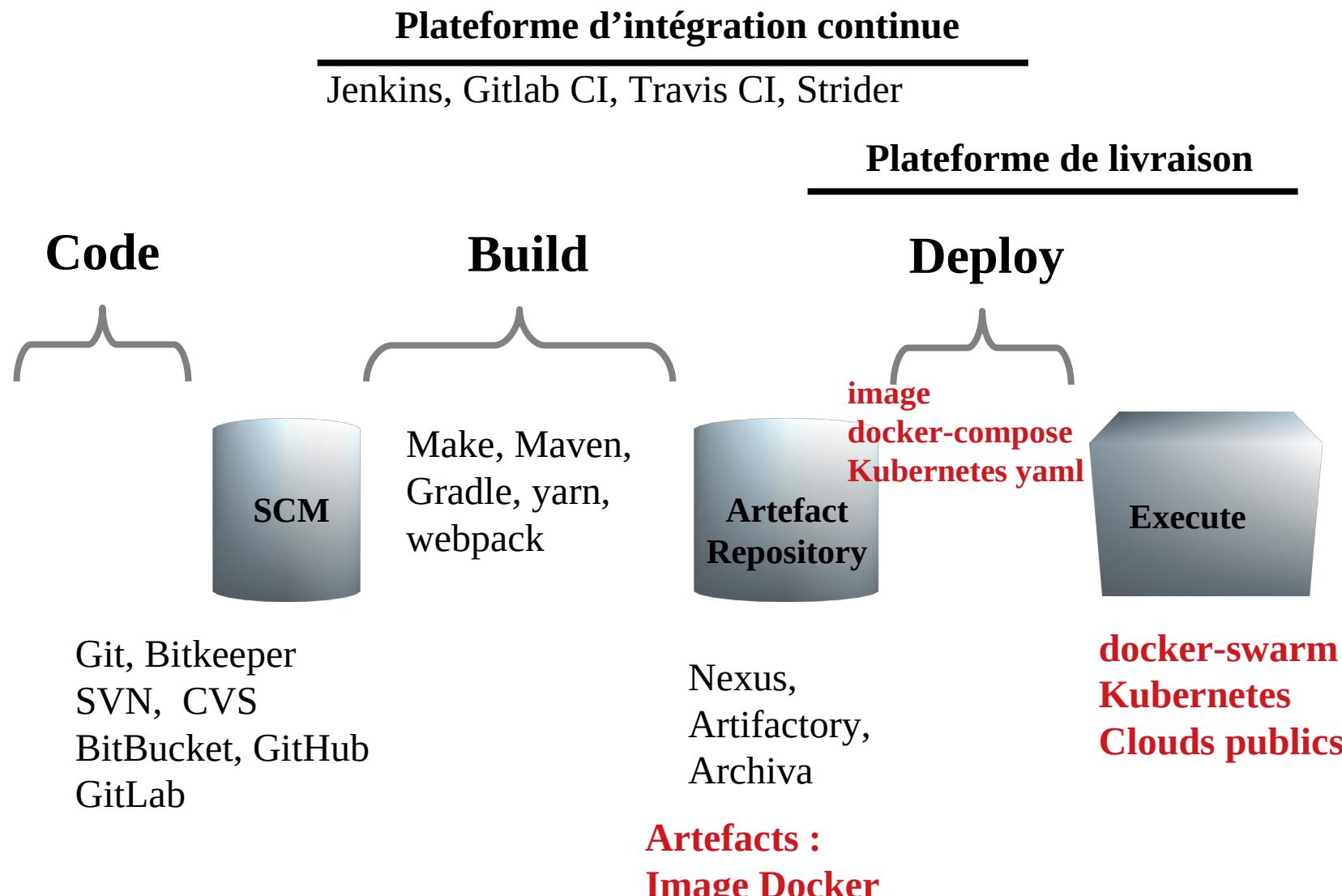
Containerisation. Le cas docker

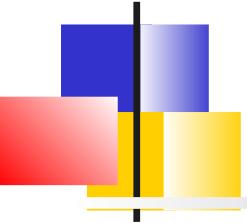
Orchestrateurs de conteneur :
Kubernetes

Kubernetes dans la pipeline CD



Cloud dans le cycle de vie



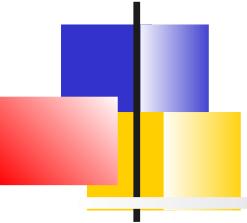


Principes de l'orchestration de conteneurs

Un simple fichier "*manifest*" définit comment démarrer un conteneur et la configuration nécessaire.

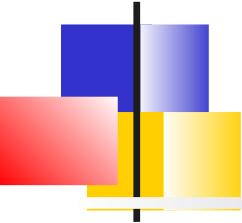
L'orchestrateur va alors trouver une machine disponible, démarrer l'application et faire le nécessaire pour qu'elle soit tout le temps accessible :

- Redémarrage si plantage
- Scaling si charge importante



Orchestrateur

- Gère un pool de ressources : les nœuds, les volumes, les adresses Ips
- Il connaît la topologie du cluster
 - Ressources disponibles
 - Applications déployées
- Il connaît l'état de santé ...
 - ...des services
 - ...de la plateform



Apport de l'orchestrateur : Déploiement blue-green

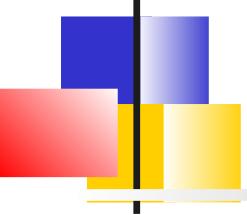
Nativement, les orchestrateurs offre des déploiements **blue-green**, : les instances de la version n est supprimée seulement lorsque les nouvelles instances de la version n+1 sont prêtes à recevoir des requêtes

Des outils additionnels permettent encore des stratégies de déploiement plus sophistiquées, comme le **canary deployment** :

– les versions n et n+1 existent simultanément. Certaines requêtes sont dirigées vers la n+1 :

- Tests automatisés d'une pipeline CI
- Tests manuels via machine interne au réseau
- Canary testing : Beta-testeurs sont dirigés vers la n+1

– Lors la version n+1 est considérée comme complètement opérationnelle, la version n supprimée de l'environnement de production



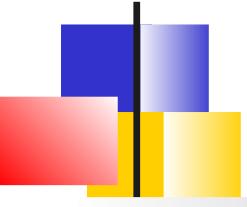
Apport de l'orchestrateur pour les micro-services

Les architectures micro-services nécessitent des services techniques. Ceux-ci peuvent être apportés par l'orchestrateur :

- Un service de **discovery** permettant de localiser les instances des services déployés sur le réseau interne de l'orchestrateur
- Un service de **proxy** et de **répartition de charge** permettant d'offrir des points unique d'accès aux services déployés
- Des services de **monitoring**
- Un service de gestion centralisée des **configuration et des clés**
- Des services de **résilience** : Redémarrage, scaling



Kubernetes



Kubernetes

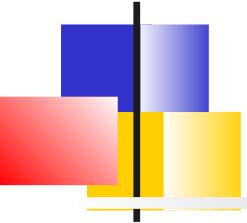
OpenSource : Proviens des projets Google's *Borg* et *Omega*

Construit depuis le départ comme un ensemble de composants faiblement couplés orientés vers

le déploiement, la surveillance et le scaling de charges de travail (workload)

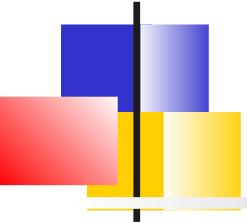


Kubernetes (Mai 2020) :
91000 commits
+2000 contributeurs
+60k* sur GitHub
Géré par la Cloud Native Computing Foundation (Groupe neutre)



Caractéristiques

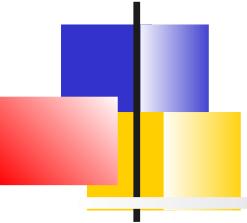
- Peut être vu comme le *noyau linux des systèmes distribués*
- Fournit une abstraction du matériel sous-jacent via une API Rest afin que des charges de travail soit consommées par un pool de ressources partagées
- Agit comme un moteur qui fait converger l'état courant d'un système vers un système désiré
- Gère des applications/déploiements pas des machines
- Tous les services gérés sont clusterisés et load-balancés par nature



Auto-correctif

Kubernetes va TOUJOURS essayer de diriger le cluster vers son état désiré.

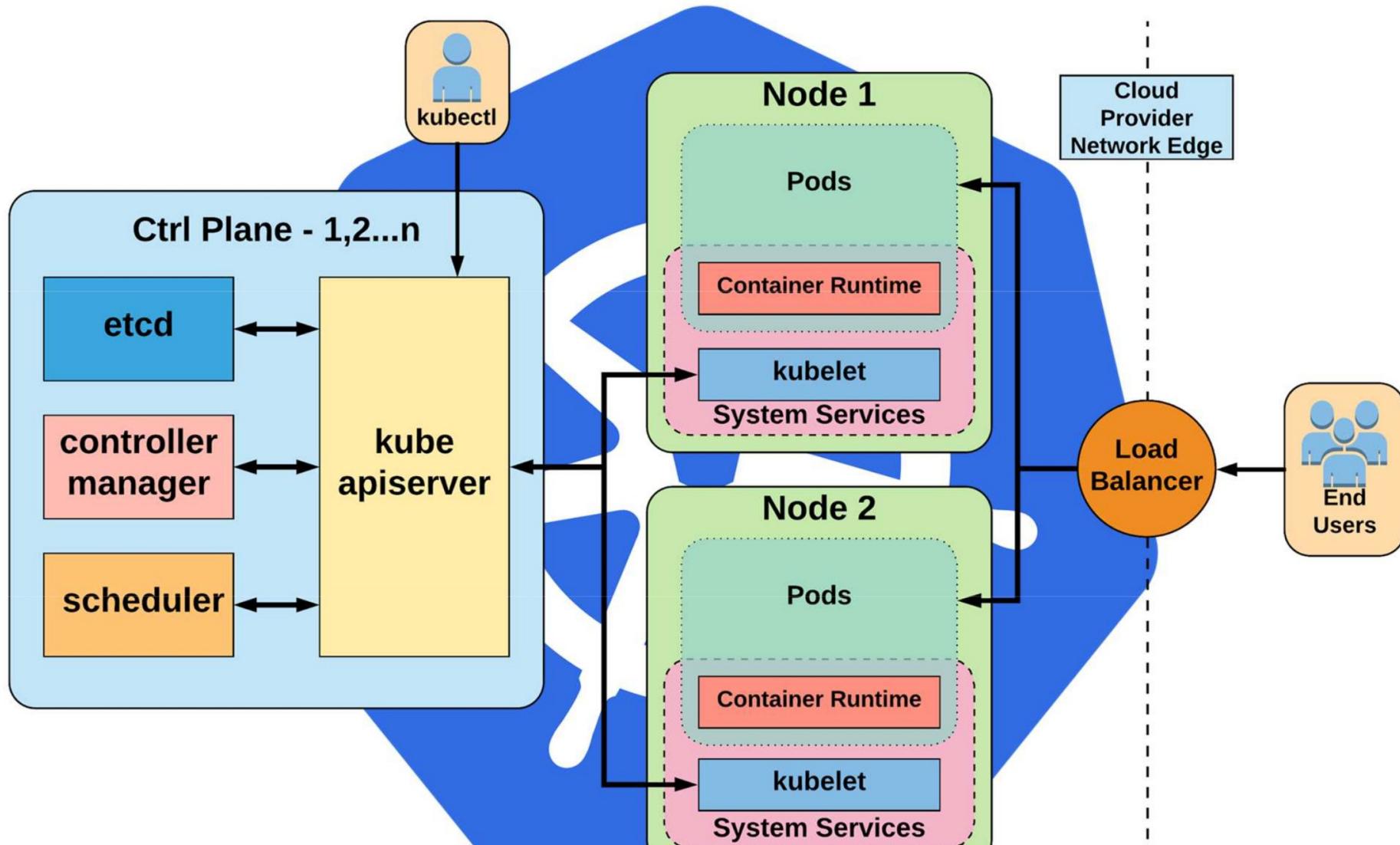
- **Moi**: «Je veux que 3 instances de Redis toujours en fonctionnement.»
- **Kubernetes**: «OK, je vais m'assurer qu'il y a toujours 3 instances en cours d'exécution. »
- **Kubernetes**: «Oh regarde, il y en a un qui est mort. Je vais essayer d'en créer un nouveau. »

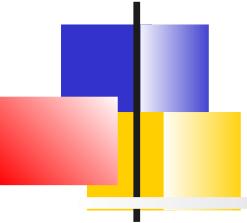


Fonctionnalités applicatives

- Scaling automatique
- Déploiements Blue/Green
- Démarrage de jobs planifiés
- Gestion d'application Stateless et Stateful
- Méthodes natives pour la découverte de services
- Intégration et support d'applications fournies par des tiers (*Helm*)

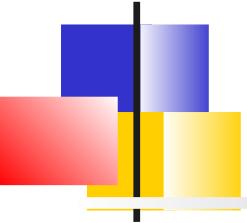
Architecture cluster





Règles réseau à l'intérieur du cluster

- Tous les conteneurs d'un *pod* peuvent communiquer entre eux sans entrave via *localhost*
- Tous les *pods* peuvent communiquer avec tous les autres *pods* sans NAT.
- Tous les nœuds peuvent communiquer avec tous les pods (et inversement) sans NAT.
- L'adresse IP avec laquelle se voit un *pod* est la même adresse que les autres voient de lui.
- Il est possible de partitionner un cluster en plusieurs clusters avec des espaces de noms



API

L'interaction se fait par une API Rest très riche.

L'API est très cohérente et tous les appels suivent le même format

Format:

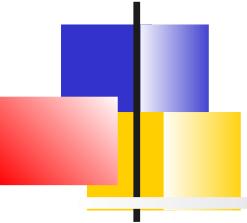
`/apis/<group>/<version>/<resource>`

Examples:

`/apis/apps/v1/deployments`

`/apis/batch/v1beta1/cronjobs`

L'outil ***kubectl*** et le format ***yaml*** sont les plus appropriés pour effectuer les requêtes REST

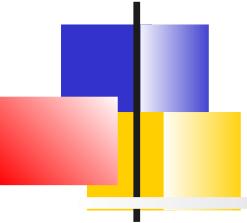


Principes

L'API est une API Rest, elle permet principalement des opérations CRUD sur des **ressources**

En particulier, le client *kubectl* propose les commandes :

- **create** : Créer une ressource
- **get** : Récupérer une ressource
- **edit/set** : Mise à jour d'une ressources
- **delete** : Suppression d'une ressource



Ressources applicatives

Les principales ressources applicatives d'un cluster Kubernetes sont :

- **deployment** : Un déploiement permet de déployer une stack applicative. Leur descripteurs font référence à des *ReplicaSet*, ils peuvent être historisés
- **replicaSet** : Ils définissent le nombre d'instances maximales pour une image de conteneur applicative
- **pod** : Ce sont des conteneurs qui s'exécutent, ils sont distribués sur les nœuds par le scheduler de *Kubernetes*
- **service** : Ce sont des points d'accès stables à un service applicatif

pod

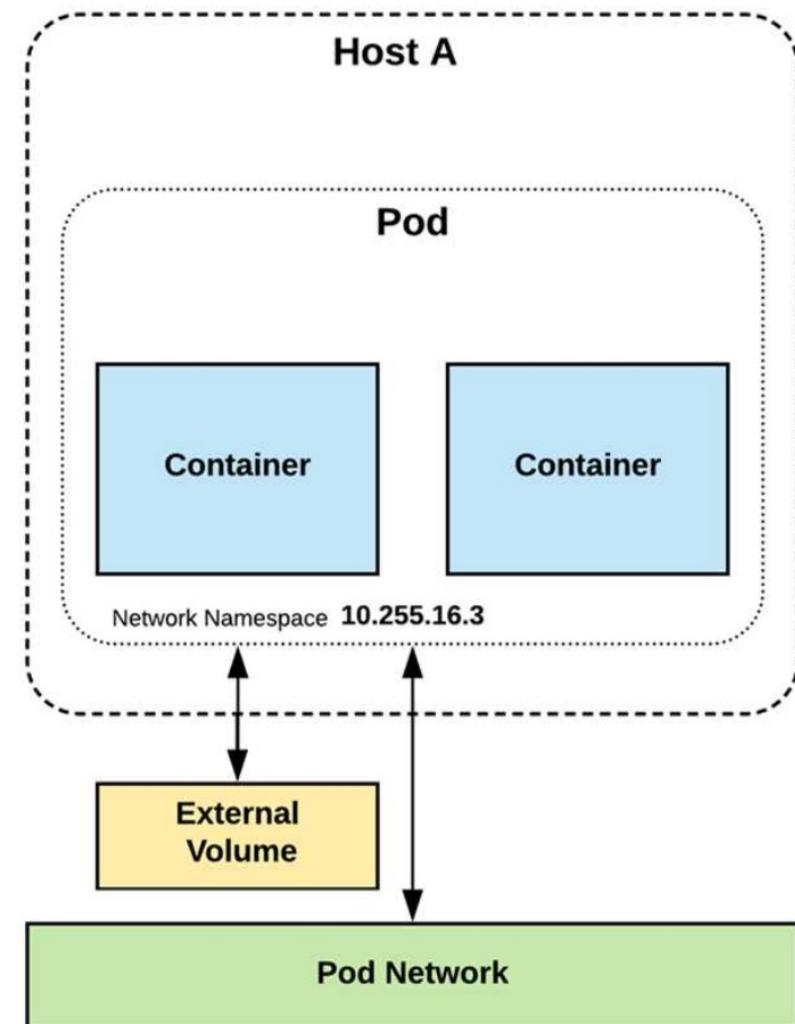
Un **pod** est la plus petite unité de travail

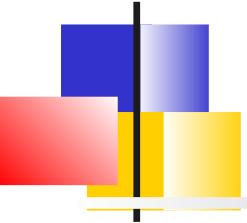
Un *pod* regroupe un ou plusieurs conteneurs qui partagent :

- Une adresse réseau
- Les mêmes volumes

Les pods sont éphémères. Ils disparaissent lorsqu'ils :

- Sont terminés
- Ont échoués
- Sont expulsés par manque de ressources





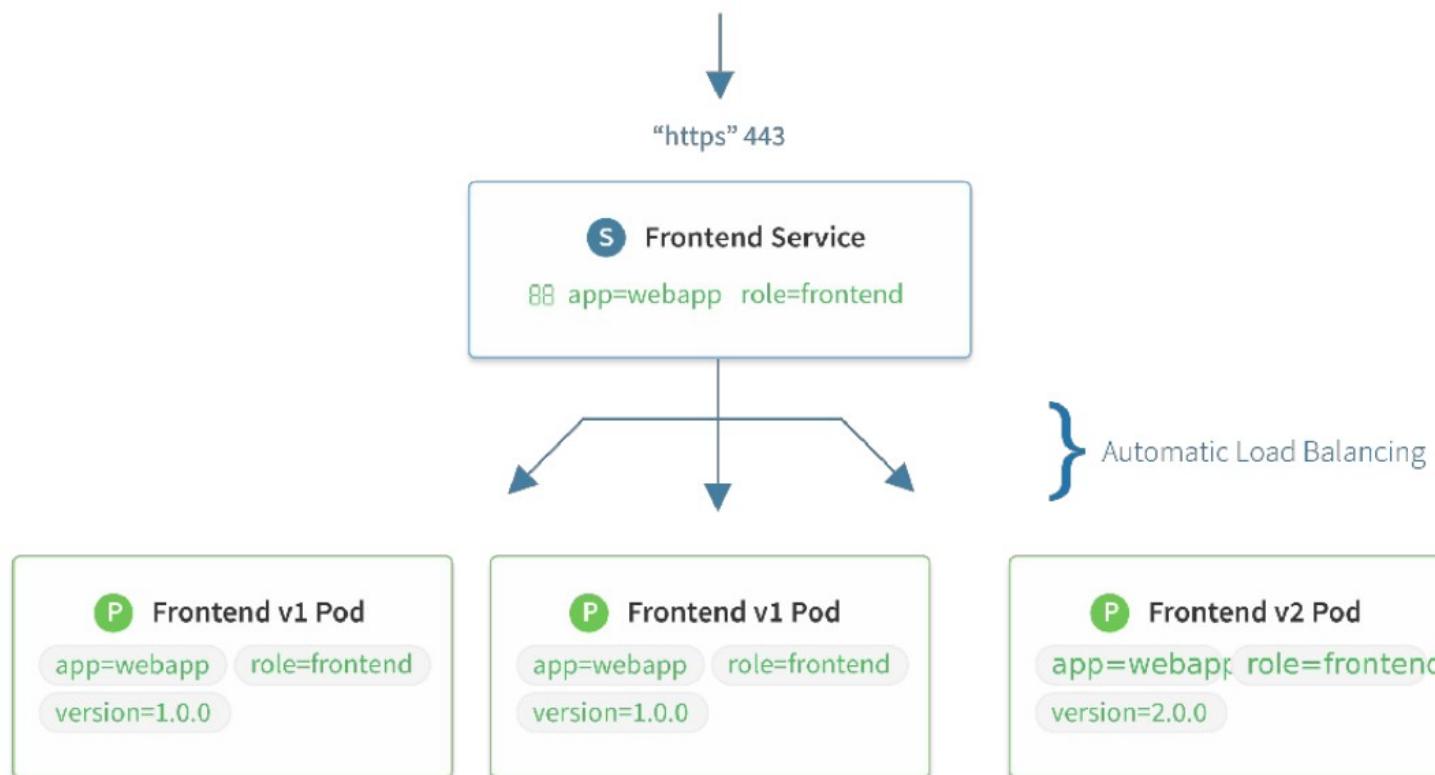
Services

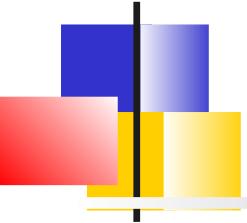
Un **service** est une méthode unifiée d'accès aux charges de travail exposées des *pods*.

Ressource durable. Les services ne sont pas éphémères :

- IP statique du cluster
- Nom DNS statique (unique à l'intérieur d'un espace de nom)

Service





Ressource *deployment*

Exemple description d'un déploiement:

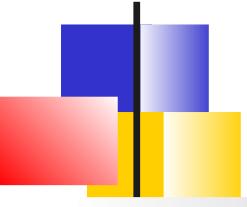
```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  spec:
    containers:
      - image: dthibau/annuaire
        name: annuaire
```

A partir de ce type de fichier *.yml*, on peut créer la ressource via :

kubectl create -f ./my-manifest.yaml

Ou

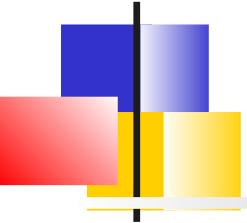
kubectl apply -f ./my-manifest.yaml



Ressource service

Un service nommé *my-service* qui représente tous les pods ayant le **label app=MyApp** et qui mappe son port 80 vers le port 80 des pods

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```



Commandes *kubectl*

create : Crée une ressource à partir d'un fichier ou de stdin.

get : Afficher 1 ou plusieurs ressources

describe : Afficher les détails sur une ou plusieurs ressources

set : Mettre à jour des attributs sur une ressource

edit : Éditer une ressource

delete : Supprimer des ressources

execute : Exécuter une image particulière sur le cluster

logs : Afficher les logs d'un container

attach : S'attacher à un container qui s'exécute

exec : Exécuter une commande dans un container

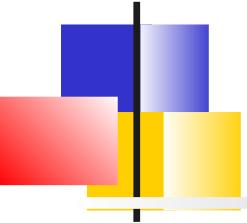
expose : Exposer un déploiement en tant que service

port-forward : Forward un ou plusieurs ports d'un pod

cp : Copier des fichiers entre conteneurs

auth : Inspecter les autorisations

...



Exemples

```
# Affiche les paramètres fusionnés de kubeconfig
kubectl config view

# Liste tous les services d'un namespace
kubectl get services

# Liste tous les pods de tous les namespaces
kubectl get pods --all-namespaces

# Description complète d'un pod
kubectl describe pods my-pod

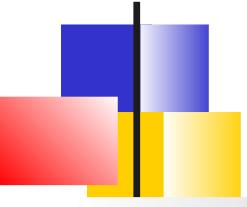
# Supprime les pods et services ayant le noms "baz"
kubectl delete pod,service baz

# Affiche les logs du pod (stdout)
kubectl logs my-pod

# S'attacher à un conteneur en cours d'exécution
kubectl attach my-pod -i

# Exécute une commande dans un pod existant (un seul conteneur)
kubectl exec my-pod -- ls /

# Écoute le port 5000 de la machine locale et forwarde vers le port 6000 de my-pod
kubectl port-forward my-pod 5000:6000
```

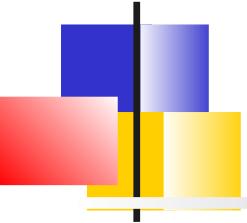


Déploiement

La ressource **deployment** permet de manipuler un ensemble de *Replicaset* (*ensemble de conteneurs répliqués*)

Les principales actions que l'on peut faire sur un déploiement sont :

- Le **rollout**: Création/Mise à jour entraînant la création des pods en arrière-plan
- Le **rollback**: Permet de revenir à une ancienne version des *ReplicaSets*
- La **scalabilité** horizontale : Permet de mettre en échelle l'application horizontalement
- La mise en pause
- La suppression de vieilles versions



Commandes de déploiement *kubectl*

Mettre à jour une image dans un déploiement existant

Enregister la mise à jour

```
kubectl set image deployment/nginx-deployment  
nginx=nginx:1.9.1 -record
```

Regarder le statut d'un rollout

```
kubectl rollout status deployment/nginx-deployment
```

Obtenir l'historique des révisions

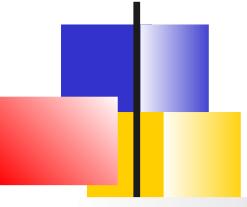
```
kubectl rollout history deployment/nginx-deployment
```

Roll-back sur la version précédente

```
kubectl rollout undo deployment/nginx-deployment
```

Scaling

```
kubectl scale deployment/nginx-deployment --replicas=10
```



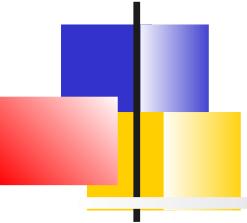
Scheduler et Workload

Les actions de l'API sont souvent asynchrones

Pour *Kubernetes*, ces ordres sont considérés comme des **workloads** à exécuter via le scheduler.

Les *workload* sont visibles via l'API, elles comportent 2 blocs de données :

- **spec** : La spécification de la ressource
- **status** : Est géré par *Kubernetes* et décrit l'état actuel de l'objet et son historique.



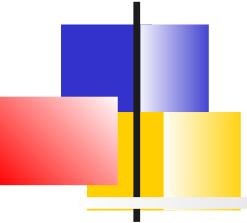
Exemple

Example Object

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
    - name: nginx
      image: nginx:stable-alpine
      ports:
        - containerPort: 80
```

Example Status Snippet

```
status:
  conditions:
    - lastProbeTime: null
      lastTransitionTime: 2018-02-14T14:15:52Z
      status: "True"
      type: Ready
    - lastProbeTime: null
      lastTransitionTime: 2018-02-14T14:15:49Z
      status: "True"
      type: Initialized
    - lastProbeTime: null
      lastTransitionTime: 2018-02-14T14:15:49Z
      status: "True"
      type: PodScheduled
```



Autres ressources du cluster

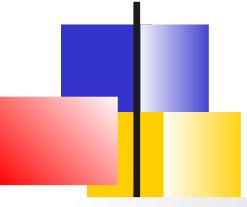
ClusterRole : Rôle avec permissions sur l'API

VolumePersistant : Système de stockage

PersistentVolumeClaims : Demande d'usage d'un volume persistant

ConfigMaps : Stockage clé-valeur pour la configuration

Secrets : Stockage de crédentiels



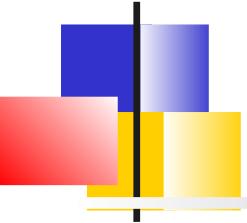
Namespace

Kubernetes prend en charge plusieurs clusters virtuels soutenus par le même cluster physique.

Ces clusters virtuels sont appelés **espaces de noms**.

- Les noms des ressources doivent être uniques dans un espace de noms, mais pas entre les espaces de noms.
- Chaque ressource Kubernetes ne peut être que dans un seul *namespace*

Les *namespaces* sont généralement utilisés dans des clusters utilisés par différentes équipes



Labels et sélecteurs

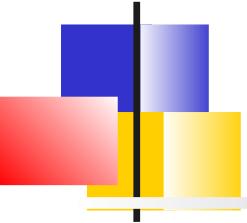
Les **labels** sont des paires clé / valeur attachées à des objets, tels que des pods, des services, des déploiements

Ils sont utilisés pour organiser et sélectionner des sous-ensembles d'objets.

Les **sélecteurs** permettent de rechercher des objets ayant des labels spécifiques.

Il y a 2 types de sélecteurs: égalité ou ensemble.

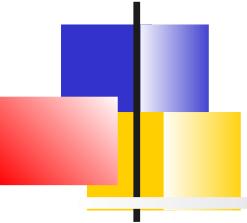
- Ils sont utilisés par les opérations *LIST* et *WATCH* de l'API
- Les services et les ReplicaSet utilisent les labels et les sélecteurs pour sélectionner les pods



Annotations

Les **annotations** (*metadata*) permettent d'attacher des métadonnées arbitraires non identifiables à des objets.

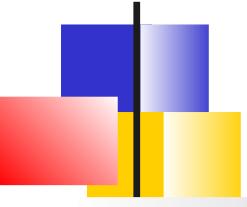
- Les clients tels que les outils et les bibliothèques peuvent récupérer ces métadonnées.



Écosystème *Kubernetes*

De nombreux outils peuvent être ajoutés à une installation cœur de Kubernetes :

- **CoreDNS** : Permet de déclarer dans un DNS interne les services (qui deviennent accessibles via leur nom)
- **Helm** : Système de gestion de package permettant d'automatiser l'installation d'autres outils (ressources Kubernetes)
- **Prometheus** : Monitoring du cluster, généralement associé à Grafana
- **Ingress** : Permettant d'exposer les services à l'extérieur du cluster
- **Istio** : Maillage de service (services mesh), ajoute un proxy sur chaque pod qui sécurise, monitore, gère les communications inter-pods



Distribution Kubernetes

Kubernetes est disponible en OpenSource mais une installation nécessite encore beaucoup d'expertise ... et beaucoup de ressources

Kubernetes est donc proposé par les acteurs du cloud

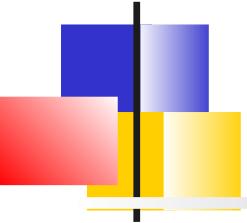
- Amazon Elastic Container Service for Kubernetes
- Azure Kubernetes Services
- Google Kubernetes Engine
- Digital Ocean
- ...

Il est également disponible en version « dev » mono-nœud :
microk8s, minikube, kind

Des versions en ligne comme : <https://labs.play-with-k8s.com/>

L'outil *Rancher* permet de gérer graphiquement plusieurs installation

Terraform permet de provisionner des cluster (et services) as Code

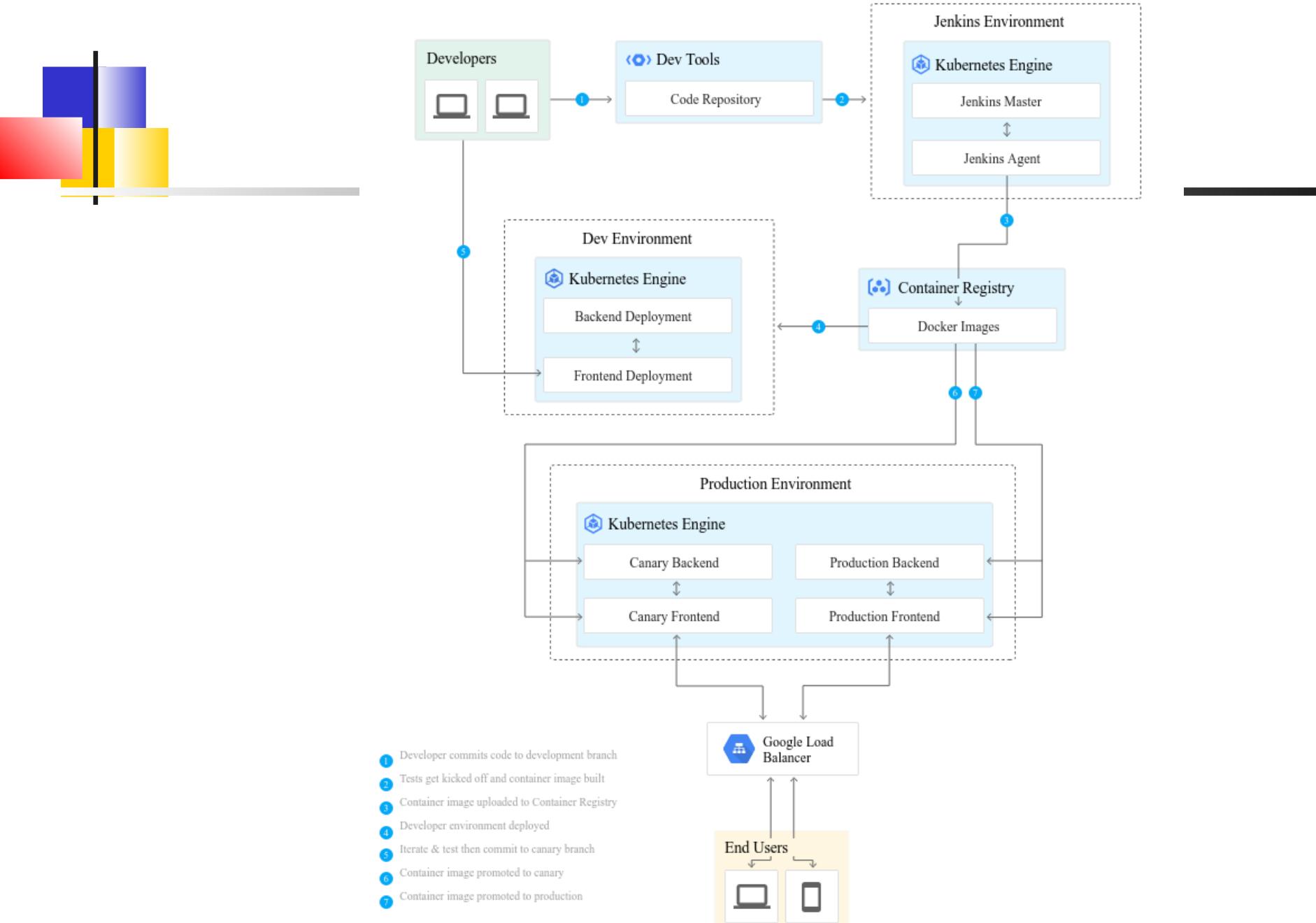


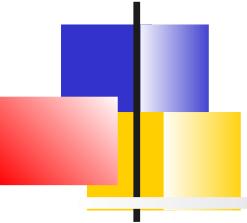
Déploiement

Considérations
Virtualisation

Outils de gestion de configuration
Containerisation. Le cas docker
Orchestrateurs de conteneur :
Kubernetes

Kubernetes dans la pipeline CD

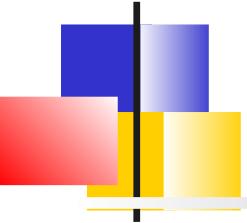




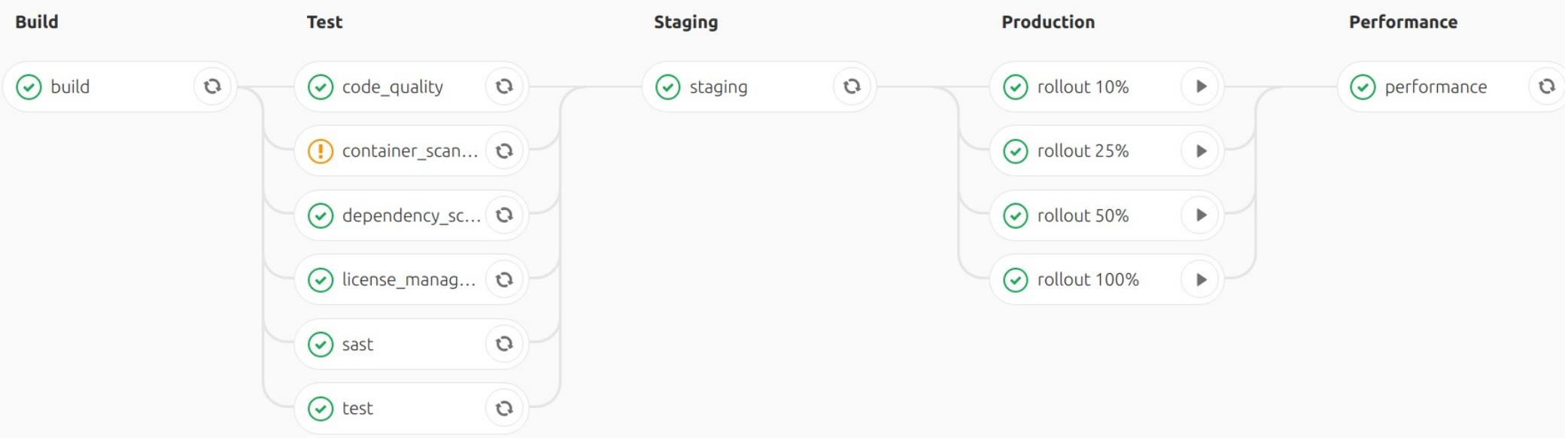
Exemple AutoDevOps GitlabCI

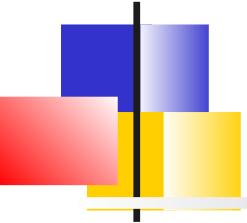
AutoDevOps est la pipeline par défaut qu'essaye d'appliquer GitLabCI, il utilise

- Docker pour builder, tester construire le conteneur
- Base Domain (Pour les review apps) : Un domaine configuré avec un DNS * utilisé par tous les projets
- Kubernetes (GKE ou Existant) : Pour les déploiements
- Prometheus : Pour obtenir les métriques
- Helm : pour installer les outils nécessaires sur le cluster Kubernetes



Pipeline AutoDevops de Gitlab CI





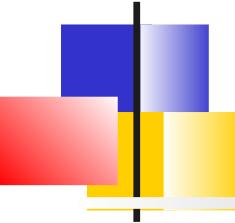
Jenkins X

Nouveau projet de Jenkins/Cloudbees basé sur *Kubernetes*

- CI/CD automatisé : Jenkins applique des pipelines tout seul !
- Chaque équipe a des environnements dédiés aux modifications en cours
- Notion de *pull-request* pour promouvoir un environnement

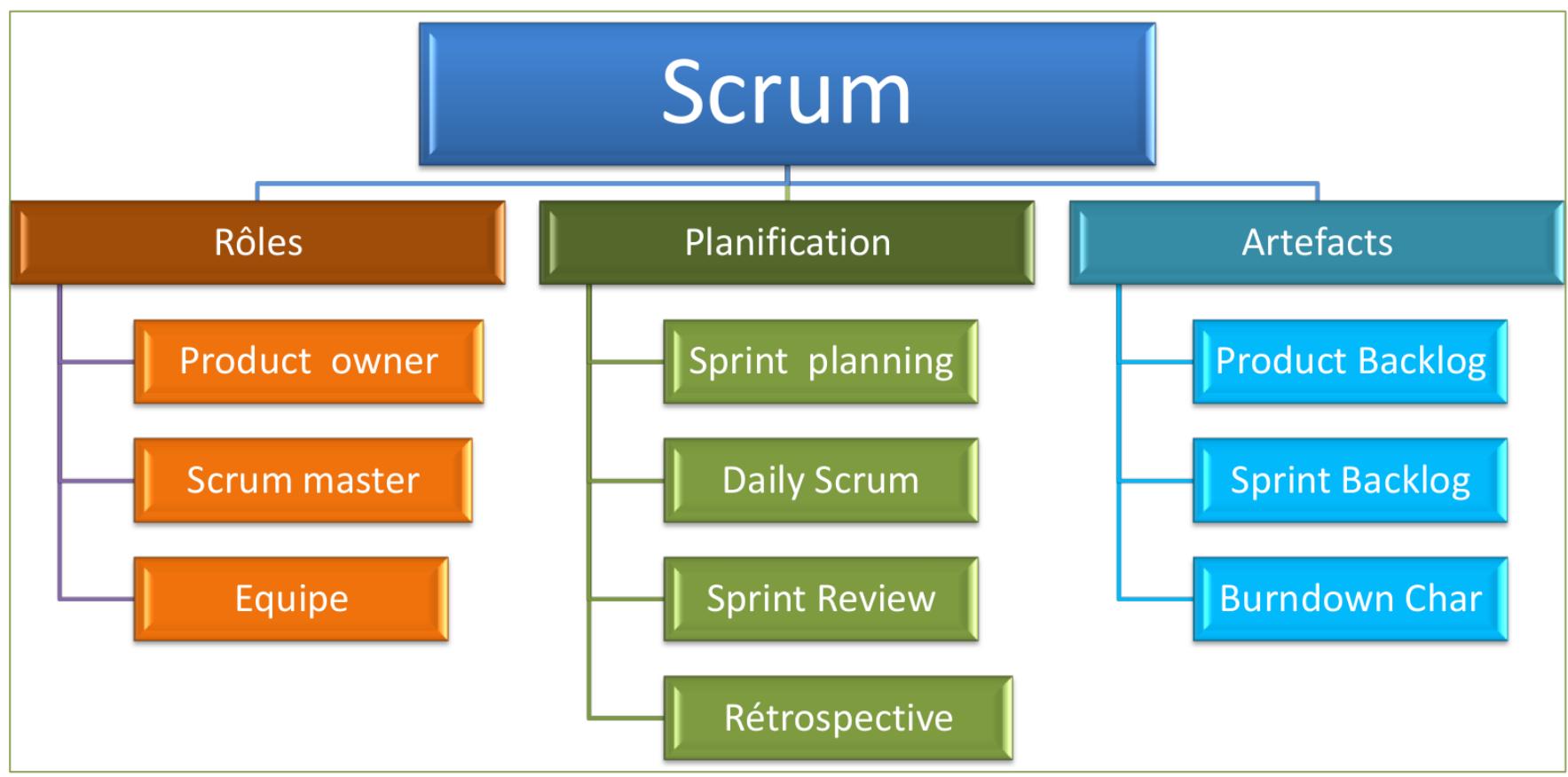


Annexes

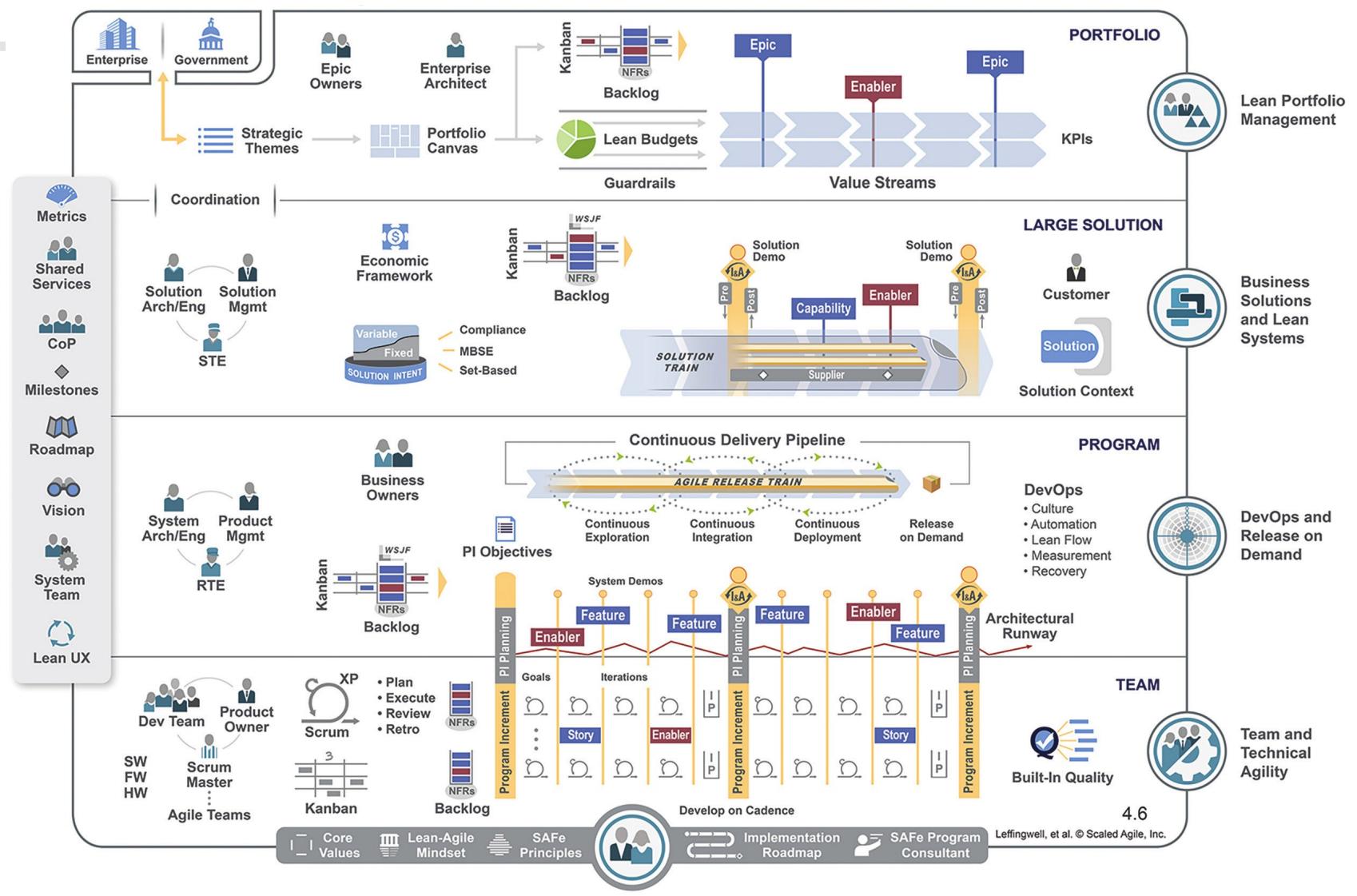


SCRUM

Rôles/Planification/Artefacts



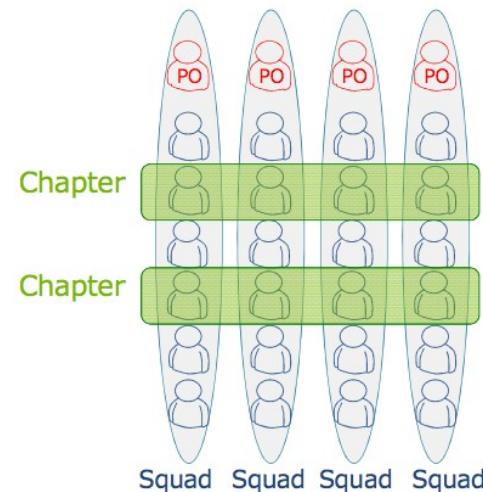
Safe : Agile à l'échelle



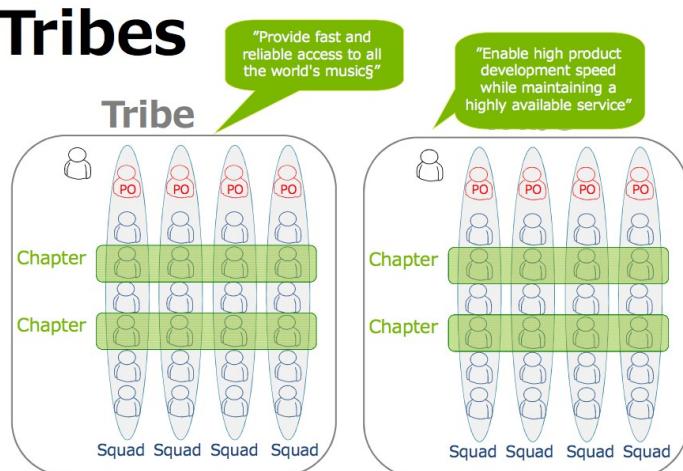
Les squads Spotify

Squad

- ✓ "Feel like a mini-startup"
- ✓ Self-organizing
- ✓ Cross-functional
- ✓ 5-7 engineers, less than 10
- ✓ Stable



Tribes



Guilds

