





Infrastructure Devops : Les outils

David THIBAU – 2019

david.thibau@gmail.com



Agenda

Introduction

- Le constat DevOps
- CI et CD

Gestion des sources

- Typologie des SCMs
- Workflows de collaboration et usage des branches
- Principales commandes *git*

Outils de build

- Caractéristiques demandées, composants, outils
- Les tests dans le build
- Release et dépôts d'artefacts

Plateforme d'intégration continue

- Concepts, Architecture
- Pipelines typiques
- Exemple Jenkins

Virtualisation et gestion de conf.

- Solutions de virtualisation
- Panorama des outils de gestion de configuration
- Vagrant, Ansible

Containerisation

- Présentation, Architecture et principales commande Docker
- Docker-compose
- CI/CD et Docker

Orchestrateur de conteneurs

- Orchestrateur, scalabilité et déploiement
- Solutions, Kubernetes
- Intégration dans la pipeline CI/CD



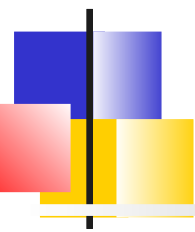
Le constat DevOps



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

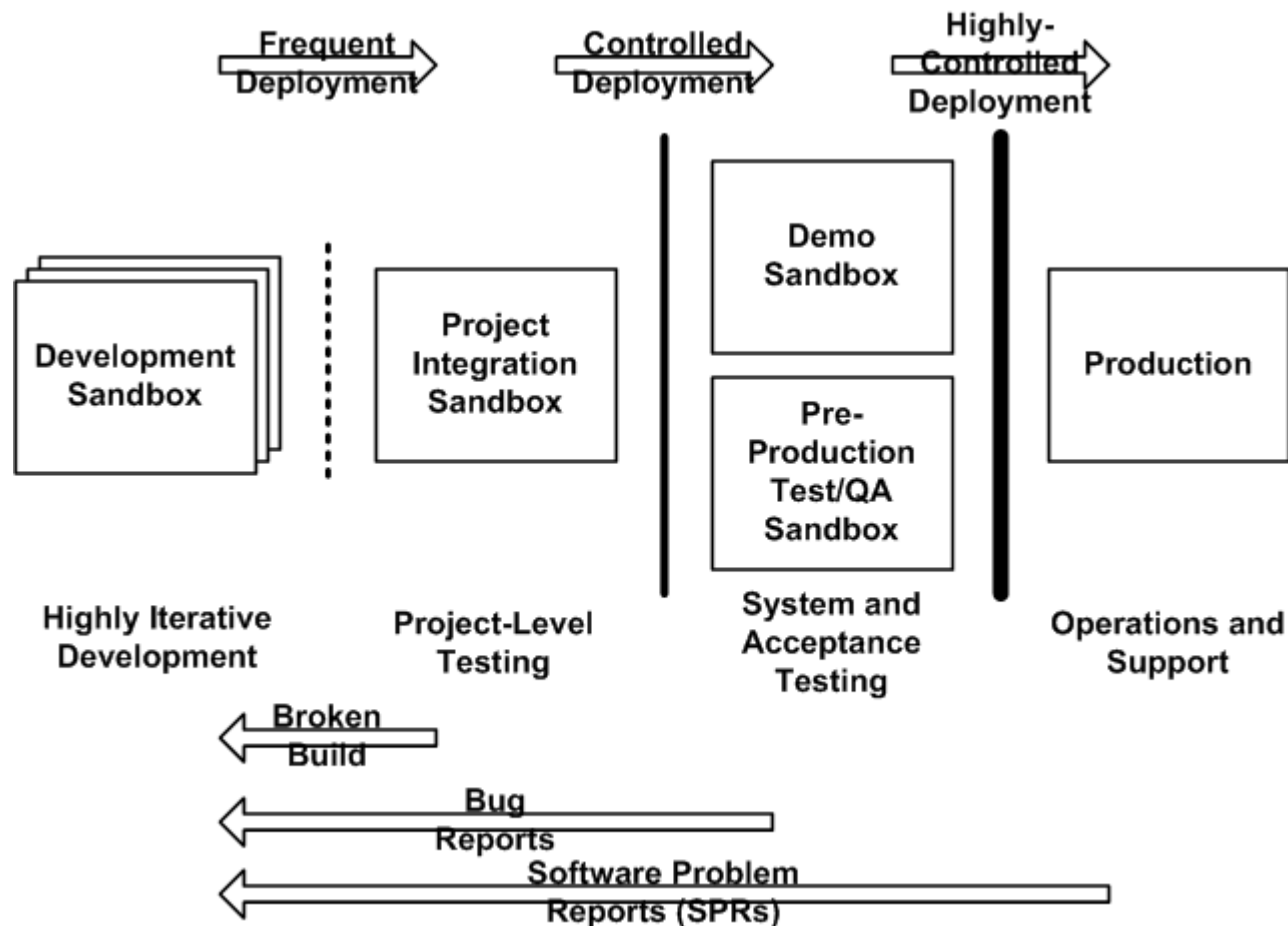


Disparité des environnements

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 de charge, de données
- **QA** : Configuration pour les tests QA. Presque la production mais pas complètement.
- **Production** : Qualité de service, charge réelle, données de production

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 »

Toutes les ressources liées au bon fonctionnement du système en production sont présent dans le gestionnaire de source.

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

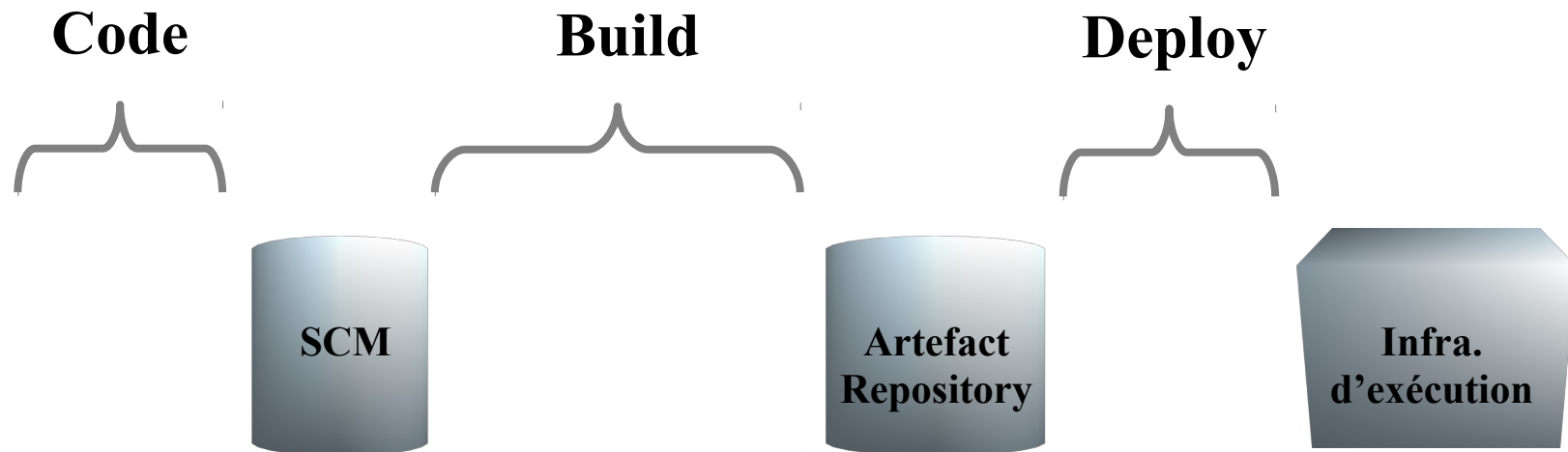
- Déployer souvent et rapidement
- Automatisation complète
- Zero-downtime
- Possibilité d'effectuer des roll-backs
- Fiabilité constante de tous les environnements
- Possibilité de scaler sans effort
- Créer des systèmes auto-correctifs, capable de se reprendre en cas de défaillance ou erreurs

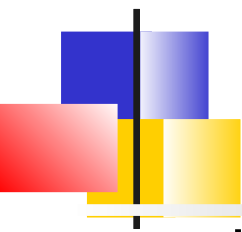


CI/CD



Cycle de vie du code





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 similaire 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



PIC : Plateforme d'intégration continue

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 et teste automatiquement l'application

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



Build is tests !

Chaque modification poussée par un développeur dans le SCM va être automatiquement testée au maximum dans tous les environnements.

=> A tout moment l'application est considérée comme étant potentiellement livrable

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



Outil de communication et de motivation

La plateforme d'intégration continue permet également de publier **en temps réel** des métriques sur la santé du projet :

- Résultats des différents tests
- Qualité du code
- Couverture fonctionnelle et avancement du projet
- Documentation

=> Donne de la confiance dans la robustesse du code développé et réduire les 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



Collaboration avec les fonctionnels

La PIC met à disposition des fonctionnels l'application en cours de développement sur un serveur d'intégration.

- 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.

Le déploiement étant automatisé, les fonctionnels peuvent décider de basculer les modifications vues en intégration vers la production.

On parle de **livraison continue (Continuous Delivery)**

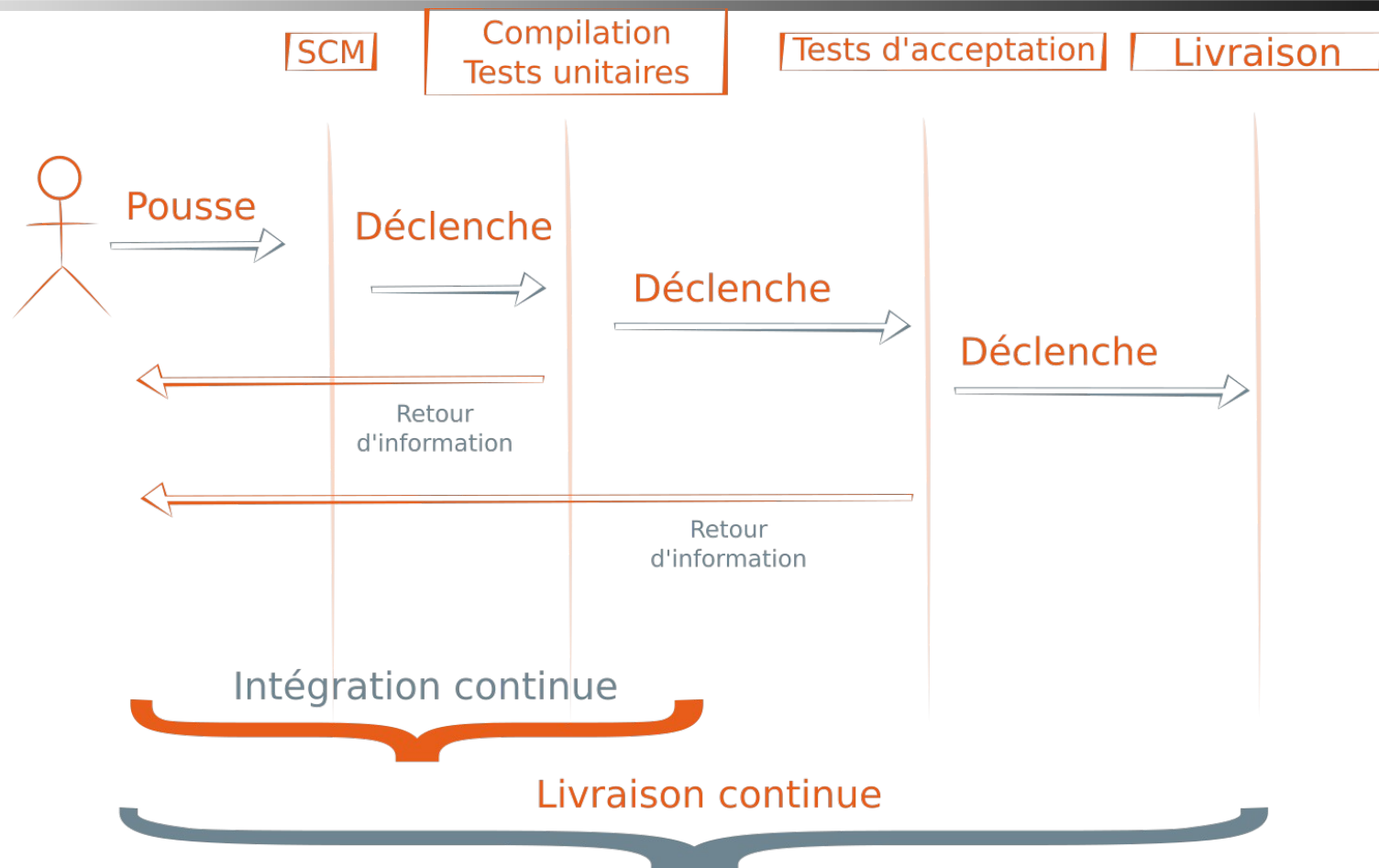


Déploiement continu

Combiné avec des tests d'acceptance, tous les builds réussis peuvent être déployer automatiquement en production sans aucune intervention manuelle

C'est le stade ultime de l'intégration continue appelée **déploiement continu.**

Intégration Continue / Livraison continue





Phases de la mise en place

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

En général, cela passe par plusieurs phases qui chacune améliore l'infrastructure et modifie les pratiques des équipes de développement.

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



Types d'outils

3 groupes d'outils s'intègrent dans le processus de Continuous Delivery :

- Le **SCM** (Source Control Management) qui centralise le code source (Source applicatif, du build, des tests, de l'infrastructure, ...), les branches et les versions
- La **PIC** (Plate-forme d'intégration continue) qui va interfacer le SCM et exécuter une série de tests, bloquer les fusions de branches en cas de problème (et forcer la revue de code), vérifier les conventions de code, et générer des rapports
- La **plate-forme de livraison** qui peut être intégrée à la PIC, elle permet de contrôler une version à livrer et provisionner les cibles de production (déploiement dans un parc informatique, sur un serveur ou sur un cluster)

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

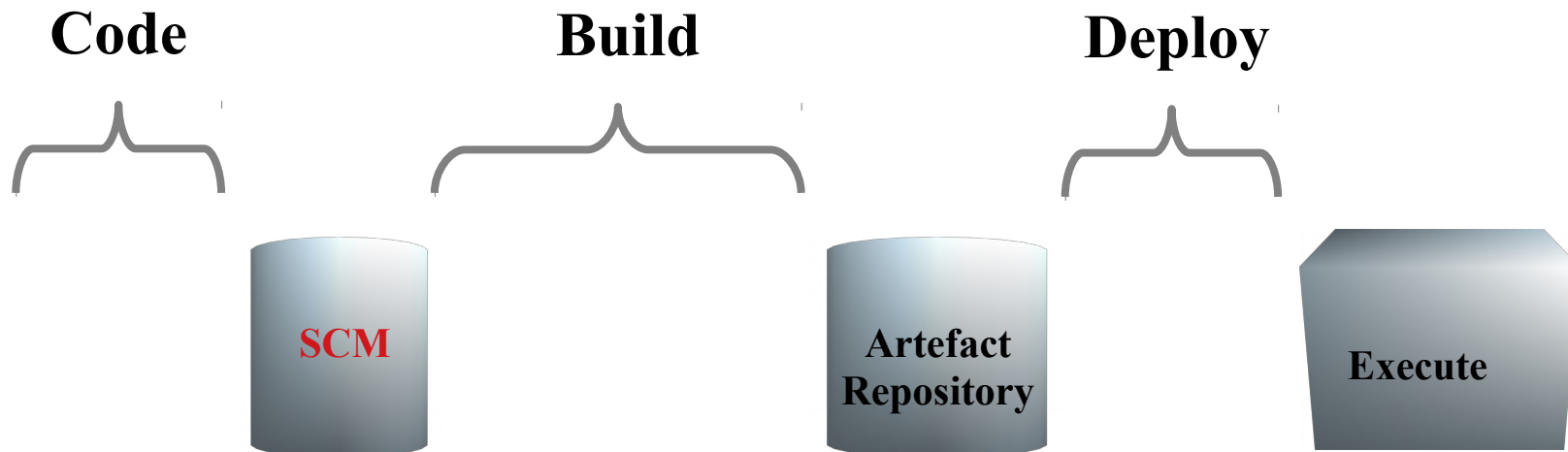
=> Bien définir le périmètre de chaque outil utilisé



Outils et Cycle de vie

Plateforme d'intégration continue

Plateforme de livraison





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 à la branche principale de rester stable (et donc disponible au fonctionnel), lorsque les travaux engagés dans une branche de développement sont terminés, ils sont intégrés dans le tronc commun
- Les **tags/étiquettes** permettent de fixer les versions des sources. Ils correspondent en général à des releases de l'application et servent à identifier les versions en production



PIC et branches

La PIC travaille en continue sur toutes les branches :

- Tronc commun (branche de production)
- Branche d'intégration (Développement de la future version)
- Branche thématique (Expérimentation de code, de fonctionnalité)

Pour toutes ses branches, la PIC essaie de pousser au plus loin le build



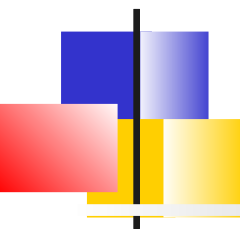
Release et Livraison

Dans le cas d'une distribution de software, on parle de **release**

La release consiste à :

- Tagger le dépôt des sources : SCM
- Produire un artefact et le stocker dans un repository d'artefact

Le tag permettra de faire évoluer la release avec des patches correctifs



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.



Automatisation de provisionnement

D'énormes progrès ont été effectués en peu de temps sur l'automatisation du provisionnement grâce à la virtualisation et la containerisation.

L'infrastructure est alors décrite dans des DSL ou scriptée. Ces fichiers font partie des sources du projets et sont donc présents dans le SCM

- La plateforme de livraison automatise le déploiement en exécutant ces fichiers sources.



Format 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 le cloud

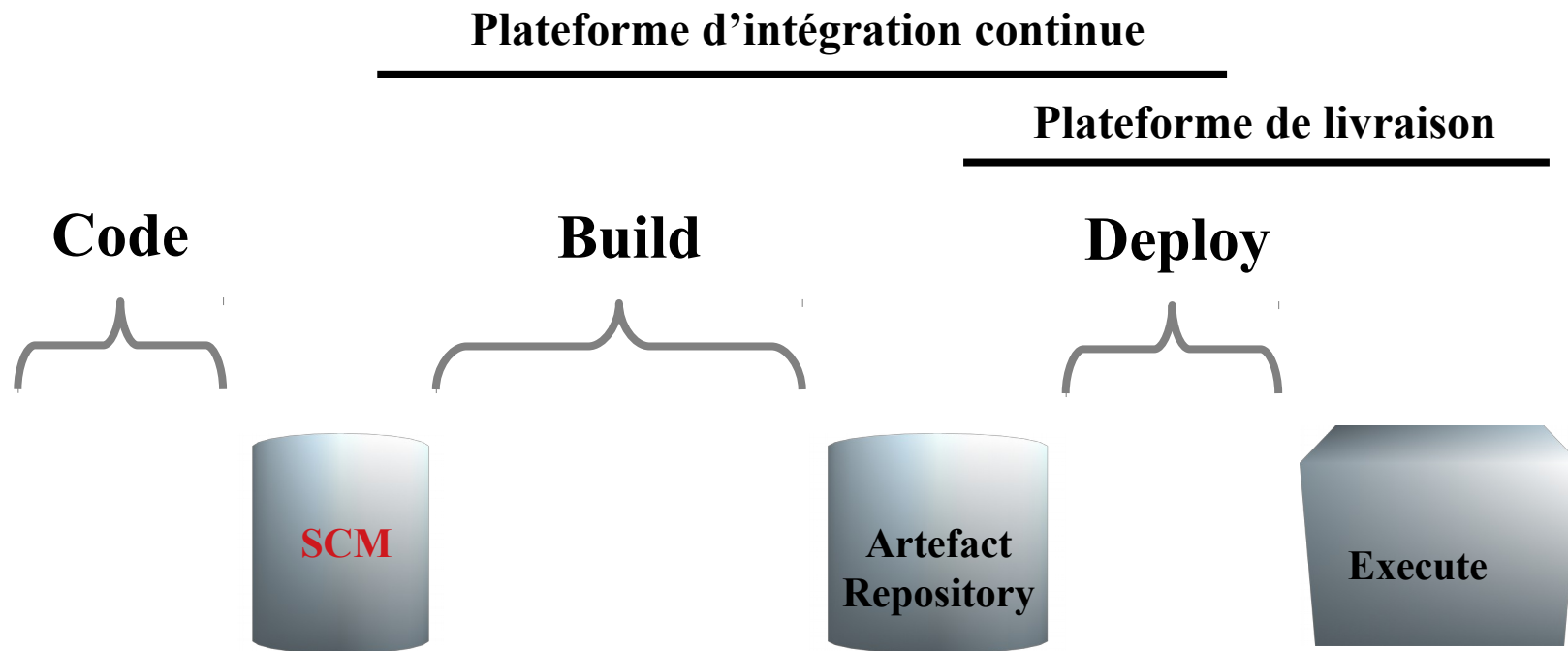
Certaines solutions ont comme vocation de gérer tous ces formats. D'autres sont spécialisés



SCM

Typologie
Workflows de collaboration et usage
des branches
Principales commandes Git

Les SCMS dans le Cycle de vie



Git, Bitkeeper
SVN, CVS
BitBucket, GitHub
GitLab
Solutions Cloud :
AWS, Google, Azure



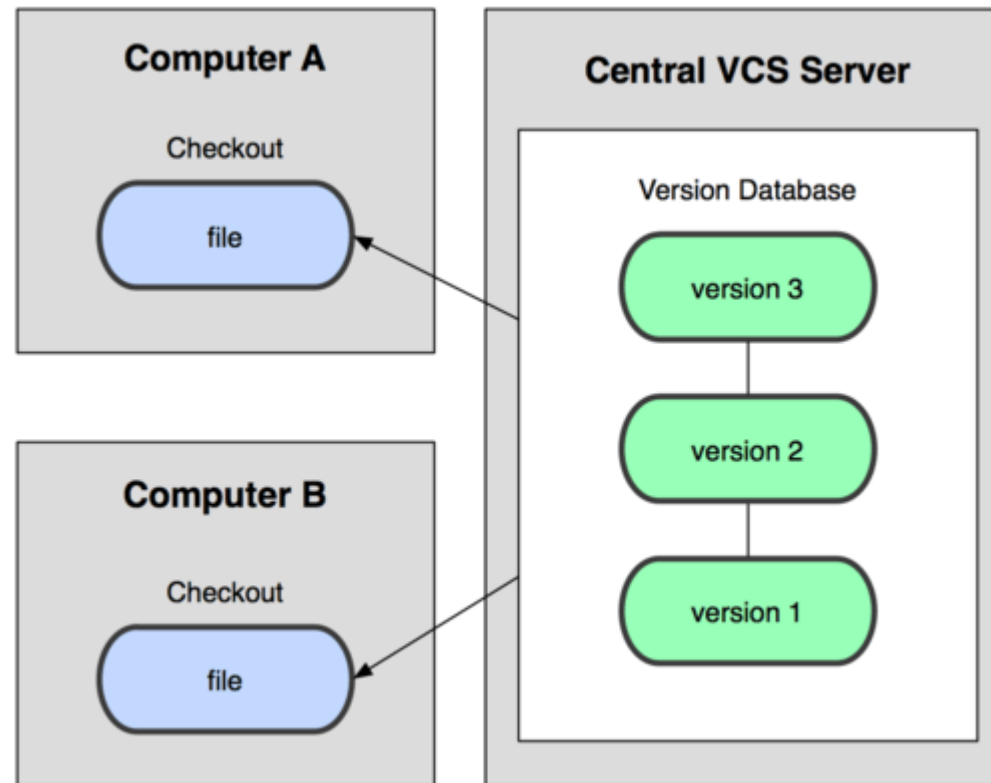
SCM

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

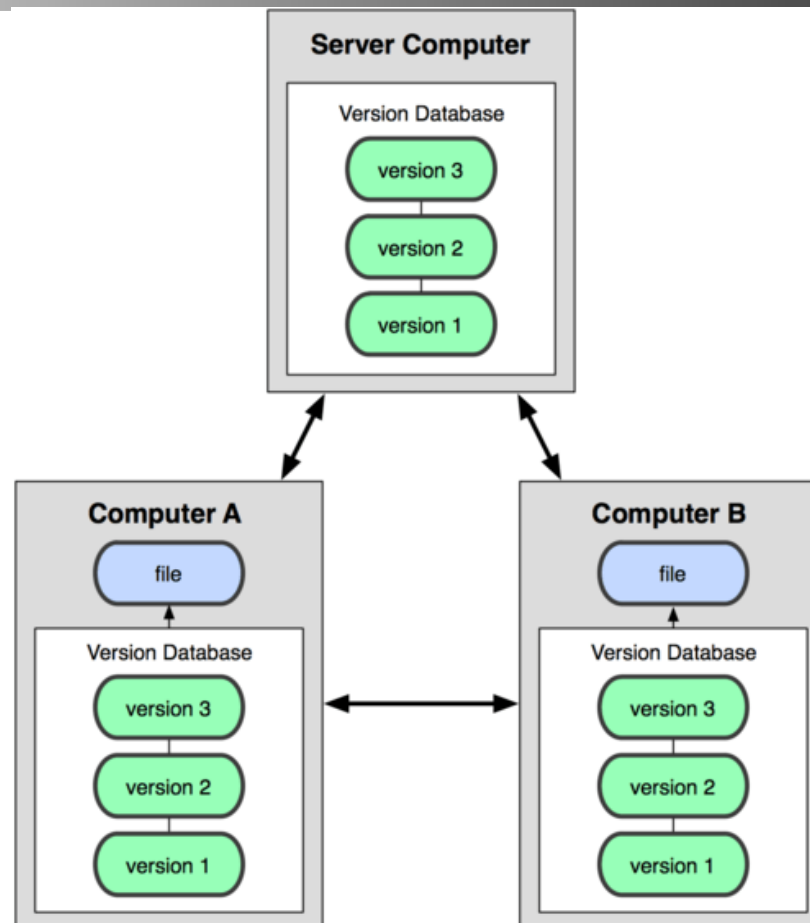
Le système permet :

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

SCM centralisés : SVN, CVS, Perforce



SCM distribués : Git, Bitbucket





Principales opérations

clone, copy : 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



Stockage des données

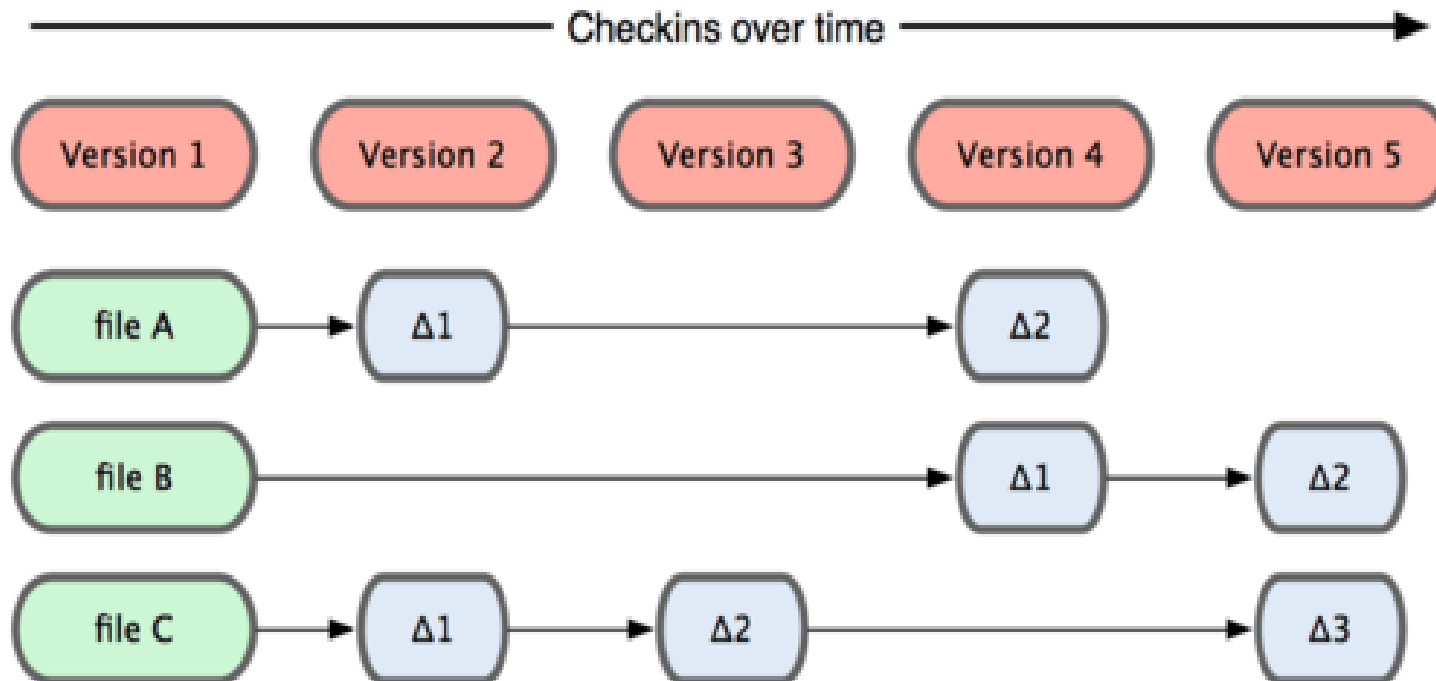
Les systèmes traditionnels stockent en général le fichier original et toutes les modifications qui lui ont été apportés (*patch*)

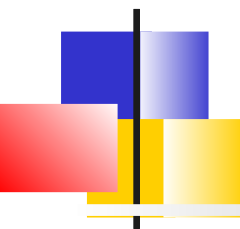
Les nouveaux systèmes stockent des instantanés complets de l'arborescence projet

- Pour être efficace, si un fichier est inchangé, son contenu n'est pas stocké une nouvelle fois mais plutôt une référence au contenu précédent

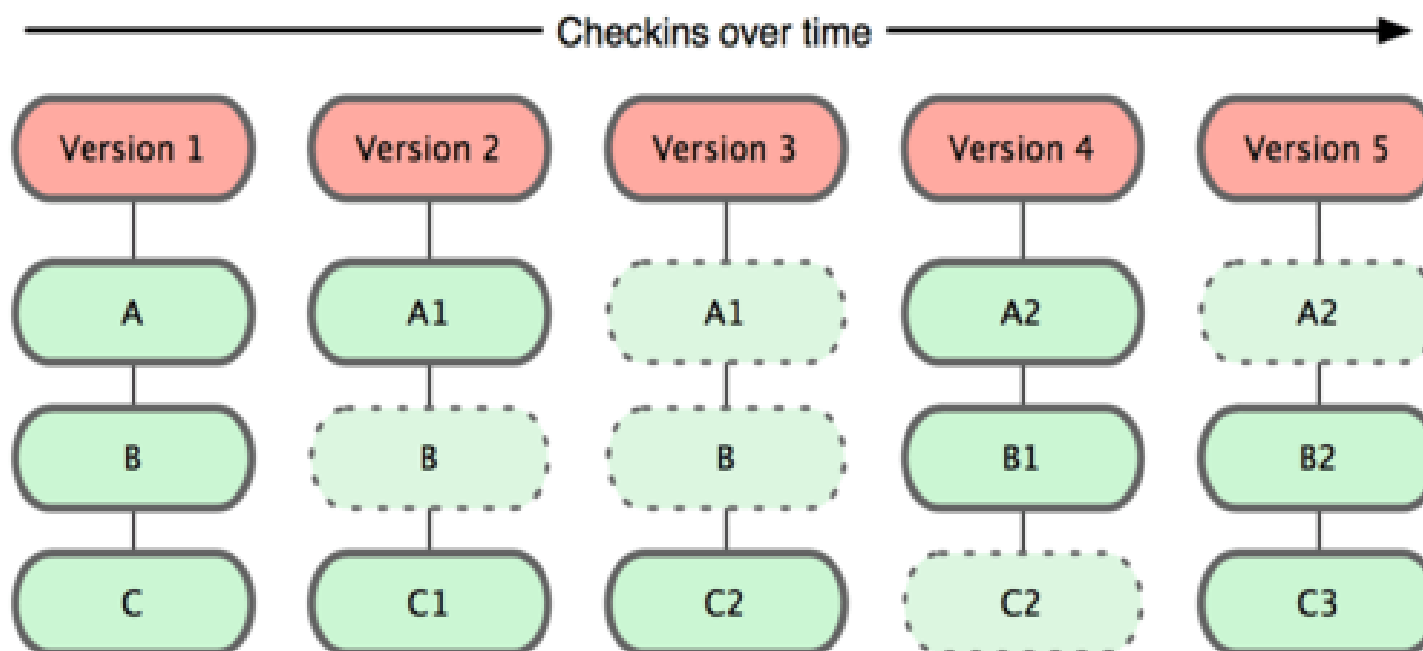


Approche standard





Approche Git





Gestion des branches

Les outils diffèrent également par leur gestion des branches

- Branche par recopie. CVS, SVN
 - => La création de branche est une opération lourde qui a des impacts sur la taille du projet
 - => Dans les SCM centralisés, le dév. n'utilise pas de branches locales
- Branche par pointeurs. Git BitBucket
 - La création de branche n'a pas d'impact sur le projet
 - Le projet peut contenir de nombreuses branches
 - En distribué, les dév. Peuvent utiliser des branches locales pour démarrer tout nouveau travail



Workflows de collaboration et usage des branches



Révision et Clé de Hash

Chaque instantané du projet est identifié dans le dépôt par une clé :

- N° de révision dans CVS, SVN
- Clé de hash dans Git

Une branche ou un tag est une façon de donner un nom à un commit particulier

- Une branche avance avec les commits.
- Un tag est fixe et immuable



Utilité des branches

Une branche est créée lorsque on veut démarrer des développements sans impacter la branche d'origine (master ou autre)

Éventuellement, lorsque les développements sont terminés ; il sont intégrés à la branche d'origine



Usage

Localement, un développeur crée des branches dès qu'il démarre un nouveau travail. Il supprime la branche locale lorsque son travail est terminé.

Sur le serveur de référence, les branches créées servent à la collaboration.

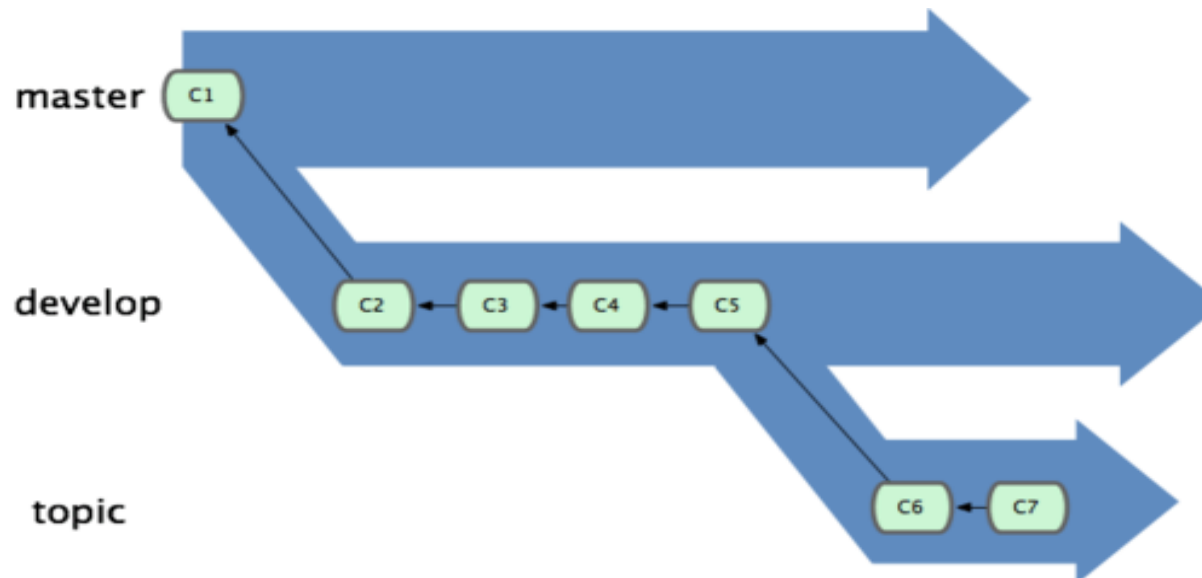
Les branches stables du serveur ont souvent des fonctions différentes (production, intégration)

Branches longues et thématiques

Sur un projet, on a généralement donc plusieurs branches ouvertes correspondantes à des étapes du développement et des niveaux de stabilité

Lorsqu'une branche atteint un niveau plus stable, elle est alors fusionnée avec la branche d'au-dessus.

On distingue les branches longues et les branches de features utilisés que pendant le développement de la fonctionnalité





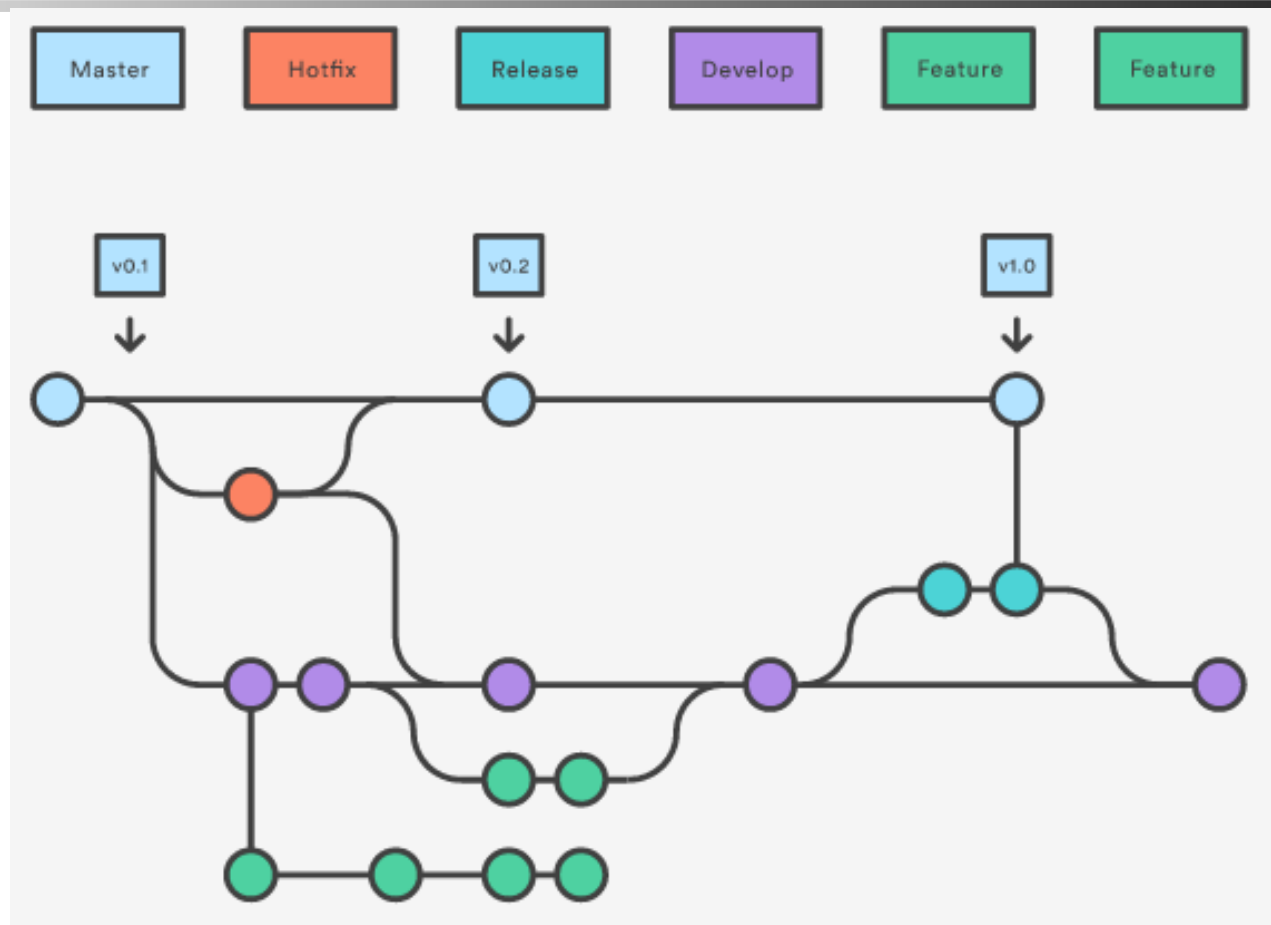
Gitflow

Le workflow **Gitflow** définit un modèle de branches orientées vers la release d'un projet

- Adapté pour la gestion de grands projets
- Il assigne des rôles très spécifiques aux différentes branches et définit quand et comment elles doivent interagir

En plus de la branche longue, il utilise différentes branches pour la préparation, la maintenance et l'enregistrement de releases

Branches Gitflow





Revue de code

En plus des branches de Gitflow, les gros projets utilisent également des branches de revue de code permettant de valider les modifications avant de les intégrer dans la branche supérieure.

Des outils tels que Gerrit, Gitlab, Github permettent la mise en place transparente de ce type de fonctionnement

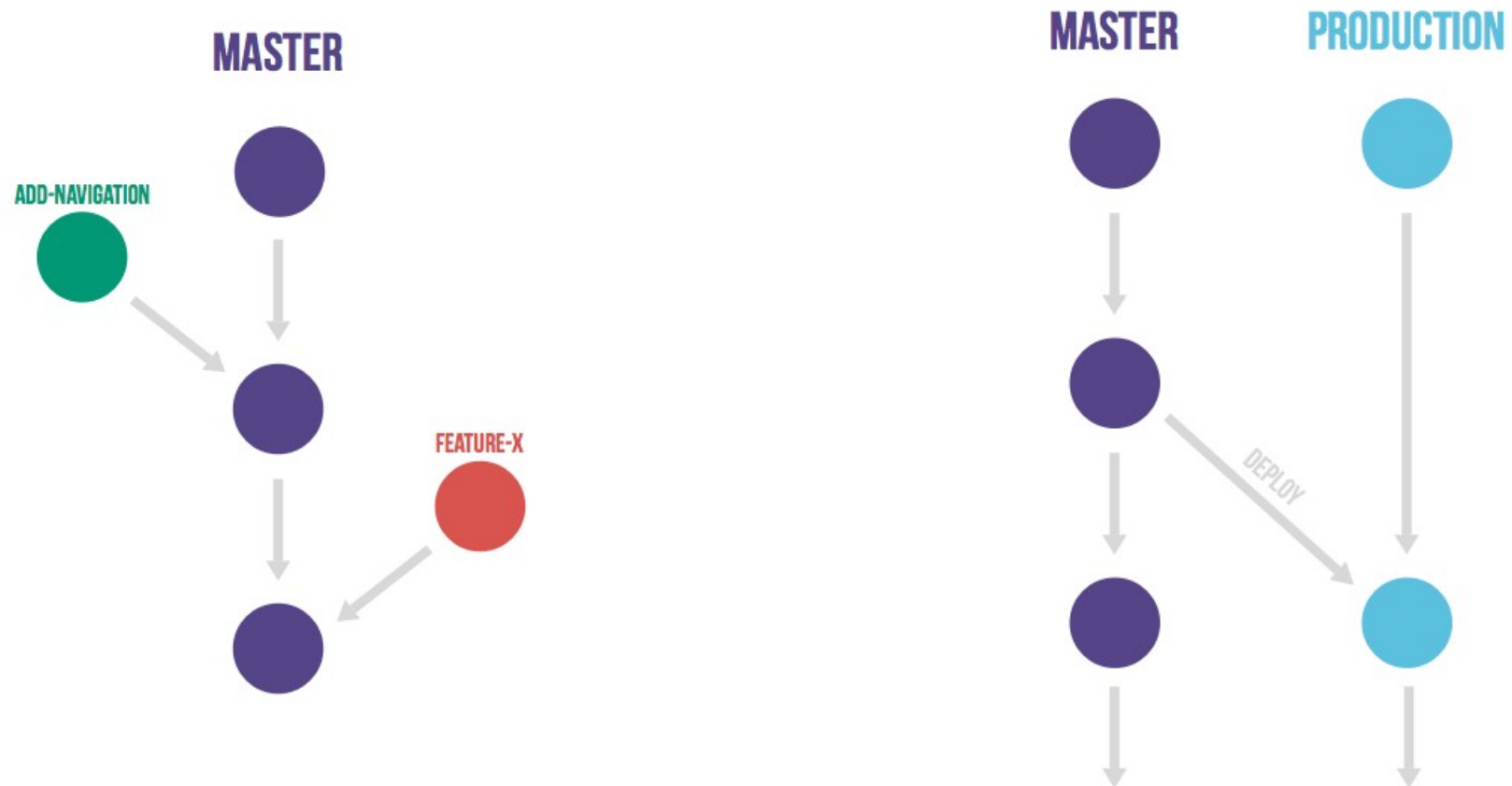


Gitlab Flow

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
- 3) Il est possible d'utiliser d'autres branches :
 - production : Chaque merge est taggée
 - release : Branche de préparation d'une release
- 4) Les Bug fixes/hot fix patches sont repris de master via des cherry-picked

Features, Master and Production





Principales commandes Git



Créer un dépôt

Il y a 2 façons de créer un dépôt :

- **Importer** un projet existant dans Git

```
$ git init
```

- **Cloner** un dépôt d'un autre serveur

```
$ git clone git://github.com/schacon/grit.git
```



Fichiers du répertoire de travail

Chaque fichier du répertoire de travail peut être ***suivi*** ou ***non-suivi***.

Les fichiers non suivis sont indiqués dans ***.gitignore***

Les fichiers suivis peuvent être dans les 3 statuts : ***non-modifié***, ***modifié*** ou ***indexé***

- Lors de l'édition d'un fichier, Git le détecte comme modifié
- Il faut alors les passer dans la zone de staging ou index pour les committer



git status

L'outil principal pour vérifier le statut des fichiers est ***git status***

Par exemple, le résultat de cette commande après un clone :

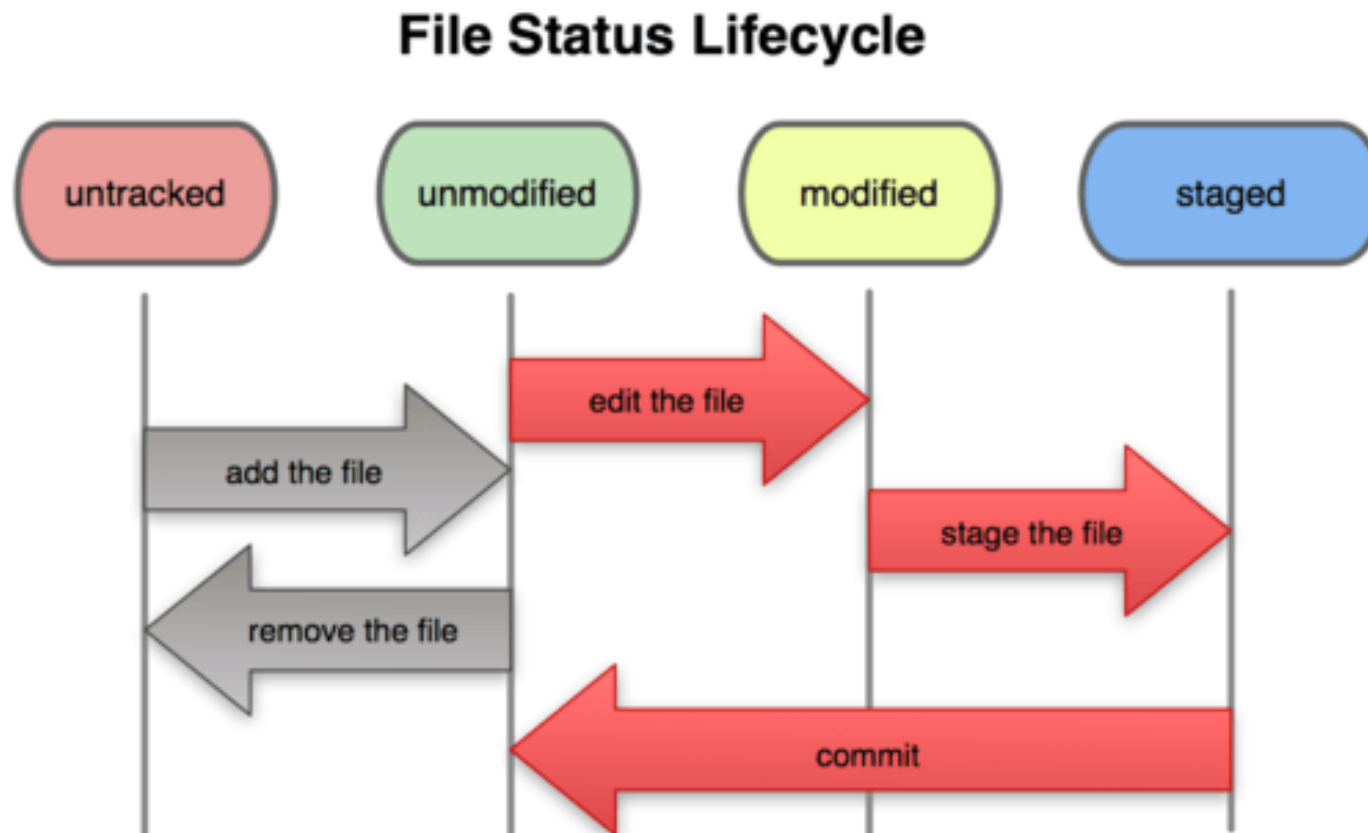
```
$ git status
```

```
On branch master
```

```
nothing to commit, working directory clean
```

=> Aucun fichier suivi n'a été modifié sur la branche par défaut *master*

Statuts des fichiers





Ajouter des fichiers

Pour commencer à suivre un fichier ou l'indexer, il faut donc utiliser la commande ***git add***.

Par exemple :

```
$ git add README
```

```
$ git status
```

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

```
new file:   README
```

La commande *git add* prend en argument un chemin de fichier ou de répertoire (récursif).



Commit

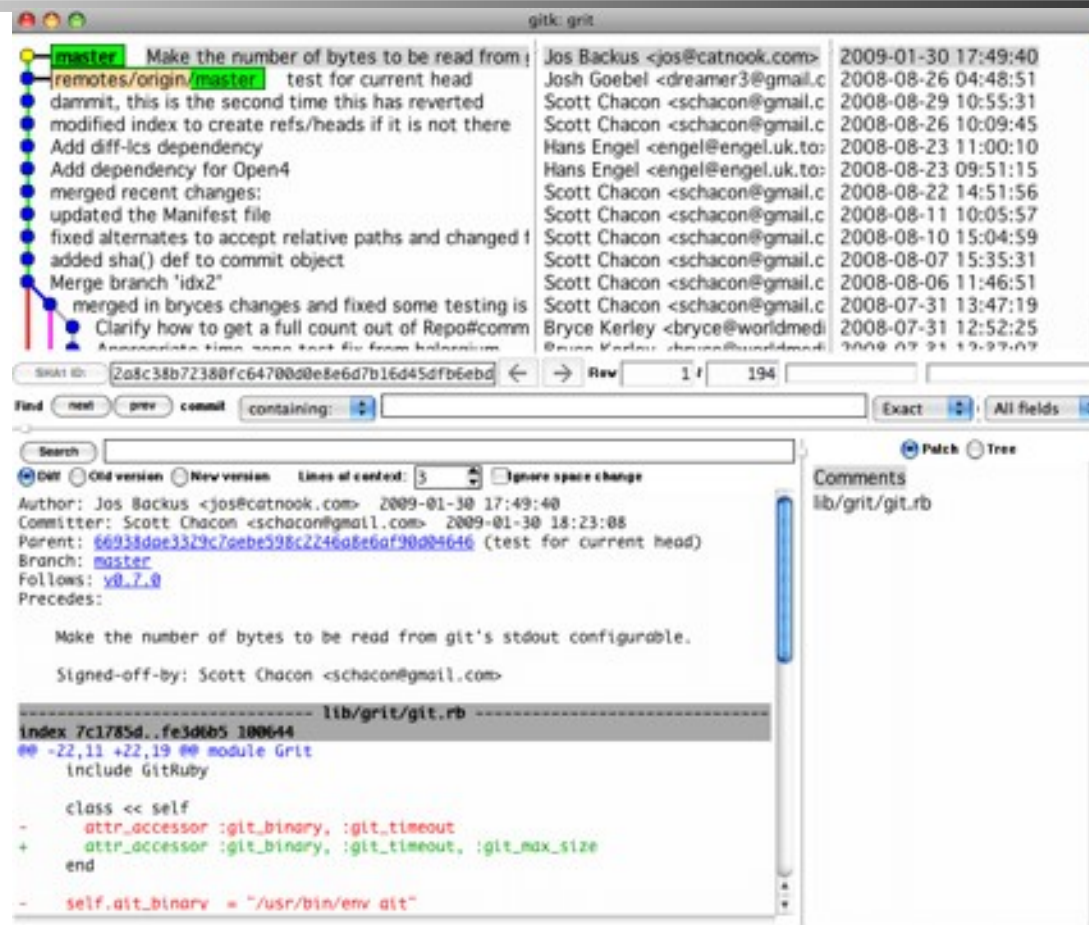
La commande *commit* n'a d'effet que sur l'index.

Tous les fichiers créés ou modifiés qui n'ont pas été ajoutés ne participent pas au commit

git commit démarre l'éditeur spécifié par la variable d'environnement *\$EDITOR*

Le développeur doit fournir un commentaire afin que le commit soit effectif.

Historique : *gitk --all*



The screenshot shows the `gitk` graphical user interface. At the top, a commit history list is visible, showing a sequence of commits with their messages and timestamps. The commit messages include "Make the number of bytes to be read from", "test for current head", "dammit, this is the second time this has reverted", "modified index to create refs/heads if it is not there", "Add diff-libs dependency", "Add dependency for Open4", "merged recent changes:", "updated the Manifest file", "fixed alternates to accept relative paths and changed", "added sha() def to commit object", "Merge branch 'idx2'", "merged in bryces changes and fixed some testing is", and "Clarify how to get a full count out of Repo#comm". The commit list is sorted by time, with the most recent commit at the top.

Below the commit list, the details of the selected commit (SHA1 ID: `2a8c38b72380fc64700d0e8e6d7b16d45dfb6ebd`) are displayed. The commit message is "Make the number of bytes to be read from git's stdout configurable." and the commit is attributed to Scott Chacon. The commit is part of the `master` branch and follows the `v0.7.0` tag. The commit is preceded by the commit with SHA1 ID `7c1785d..fe3d6b5 100644`.

The commit details section shows the commit message, the commit hash, and the commit date. The commit message is "Make the number of bytes to be read from git's stdout configurable." and the commit is attributed to Scott Chacon. The commit is part of the `master` branch and follows the `v0.7.0` tag. The commit is preceded by the commit with SHA1 ID `7c1785d..fe3d6b5 100644`.

The commit details section also shows the commit message, the commit hash, and the commit date. The commit message is "Make the number of bytes to be read from git's stdout configurable." and the commit is attributed to Scott Chacon. The commit is part of the `master` branch and follows the `v0.7.0` tag. The commit is preceded by the commit with SHA1 ID `7c1785d..fe3d6b5 100644`.



Création de branche

Une branche Git est simplement un **pointeur** pouvant se déplacer sur les commits du référentiel.

- La branche par défaut est nommé *master*.

La création de branche se fait par :

```
$ git branch testing
```

Le basculement vers une branche existante :

```
$ git checkout testing
```

Les 2 à la fois

```
$ git checkout -b testing
```



Fusion de la branche de développement

Fusion d'une branche de développement dans *master*

```
$ git checkout master
```

```
$ git merge iss53
```

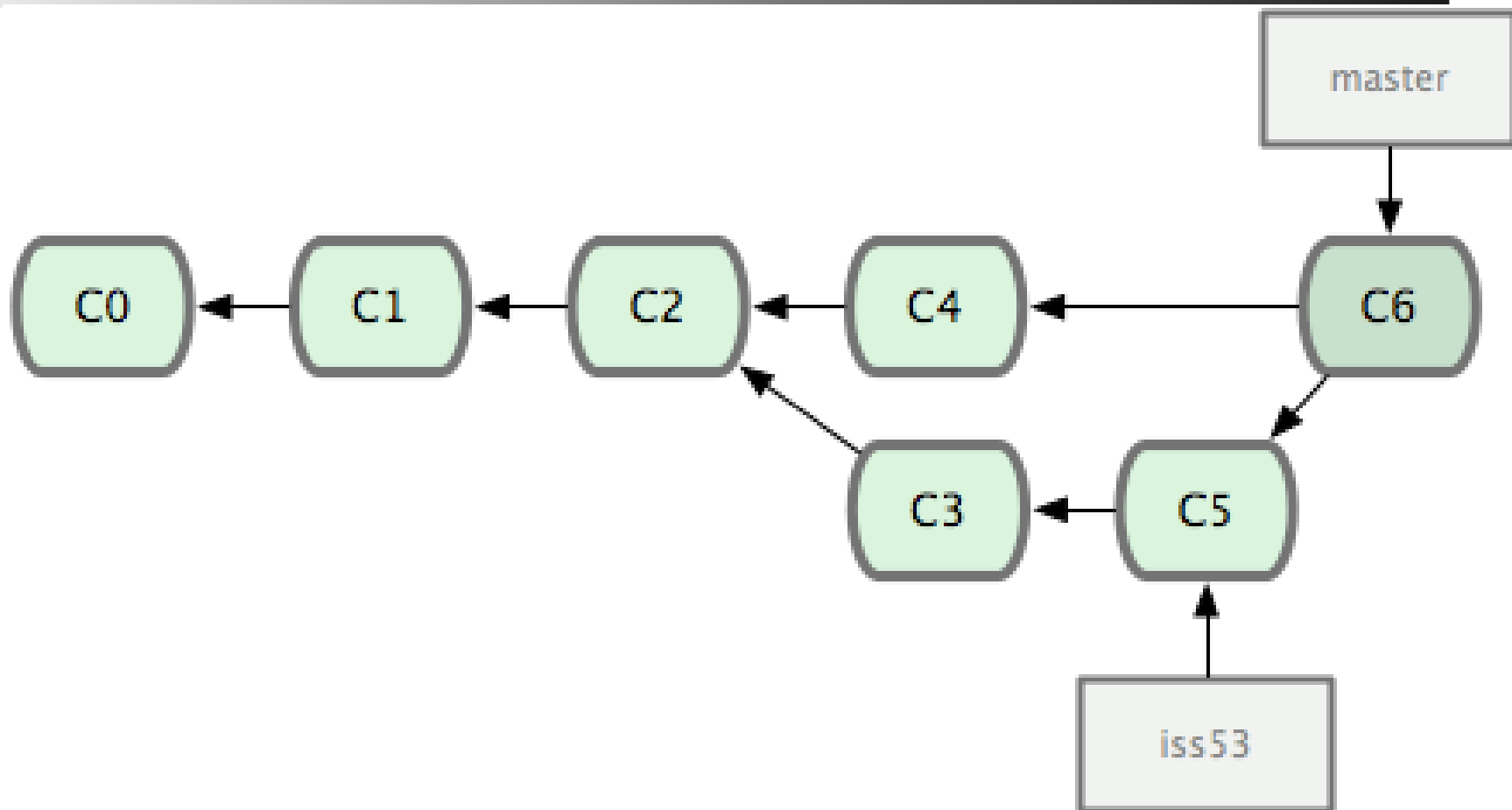
```
Auto-merging README
```

```
Merge made by the 'recursive' strategy.
```

```
README | 1 +
```

```
1 file changed, 1 insertion(+)
```

Résultat de la fusion





Rebase

Avec git, l'intégration de modification peut se faire par l'opération ***rebase***

L'opération consiste à rejouer les patches d'une autre branche sur le patch commun d'une autre branche

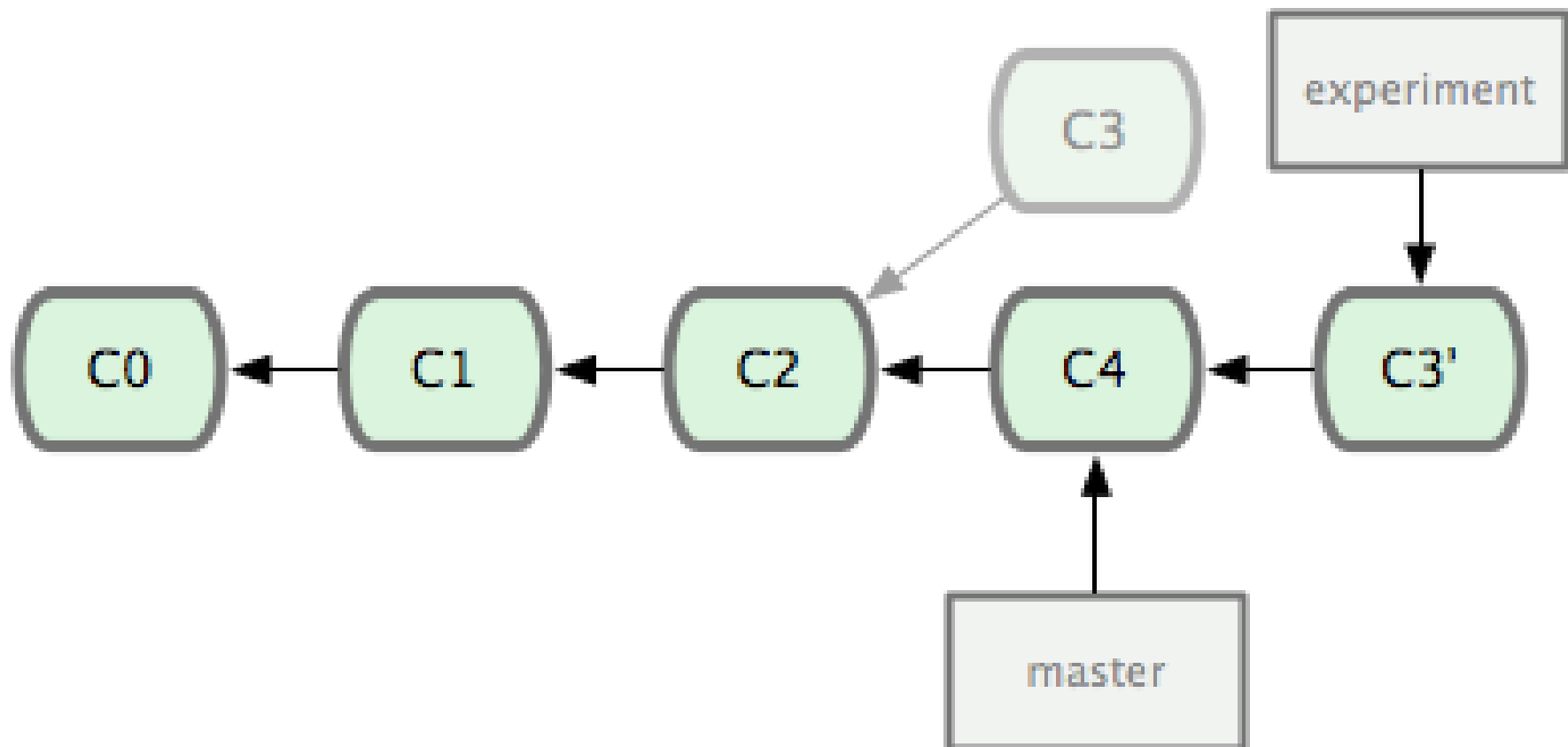
```
$ git checkout experience
```

```
$ git rebase master
```

```
First, rewinding head to replay your work on top of it...
```

```
Applying: added staged command
```


Rebase





Commandes de collaboration

Il y a 4 opérations de synchronisation avec un dépôt distant :

- **clone** : A l'initialisation, récupère l'ensemble du dépôt et extrait la branche master dans le répertoire de travail
- **fetch** : Se synchronise avec le dépôt (récupération des nouvelles infos) sans modifier le répertoire de travail
- **pull** : Se synchronise avec le dépôt et fusionne les modifications avec le répertoire de travail
- **push** : Pousse ses modifications locales vers le dépôt distant. Opération possible seulement si le dépôt local est à jour



Récupérer un dépôt distant

La commande ***git fetch*** permet de récupérer un dépôt distant
La commande se connecte au projet distant et récupère toutes les données que l'on ne possède pas déjà

Cette commande ne modifie pas l'espace de travail courant

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
* [new branch]      master      -> pb/master
* [new branch]      ticgit      -> pb/ticgit
```

=> La branche master de Paul est accessible localement par pb/master. Il est possible de la fusionner avec une de ses branches ou d'effectuer un check out complet.



git pull

A la différence de *git fetch*, la commande ***git pull [remote] [branch]*** récupère et fusionne les données automatiquement dans la branche courante. (comme *git clone* qui permet d'initialiser le dépôt et le répertoire de travail)



Pousser vers un dépôt distant

Lorsque votre projet local a atteint un point de développement à partager, il faut utiliser la commande

git push [remote-name] [branch-name]

```
$ git push origin master
```

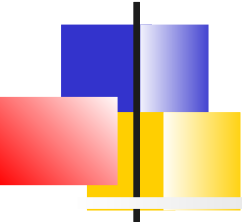
Cette commande est possible seulement si on a les droits d'écriture sur le dépôt distant et si personne n'a poussé de données entre temps

Si une opération *push* a eu lieu auparavant, il faut d'abord récupérer les données et les fusionner avant de pouvoir les pousser vers le dépôt



Outils de build

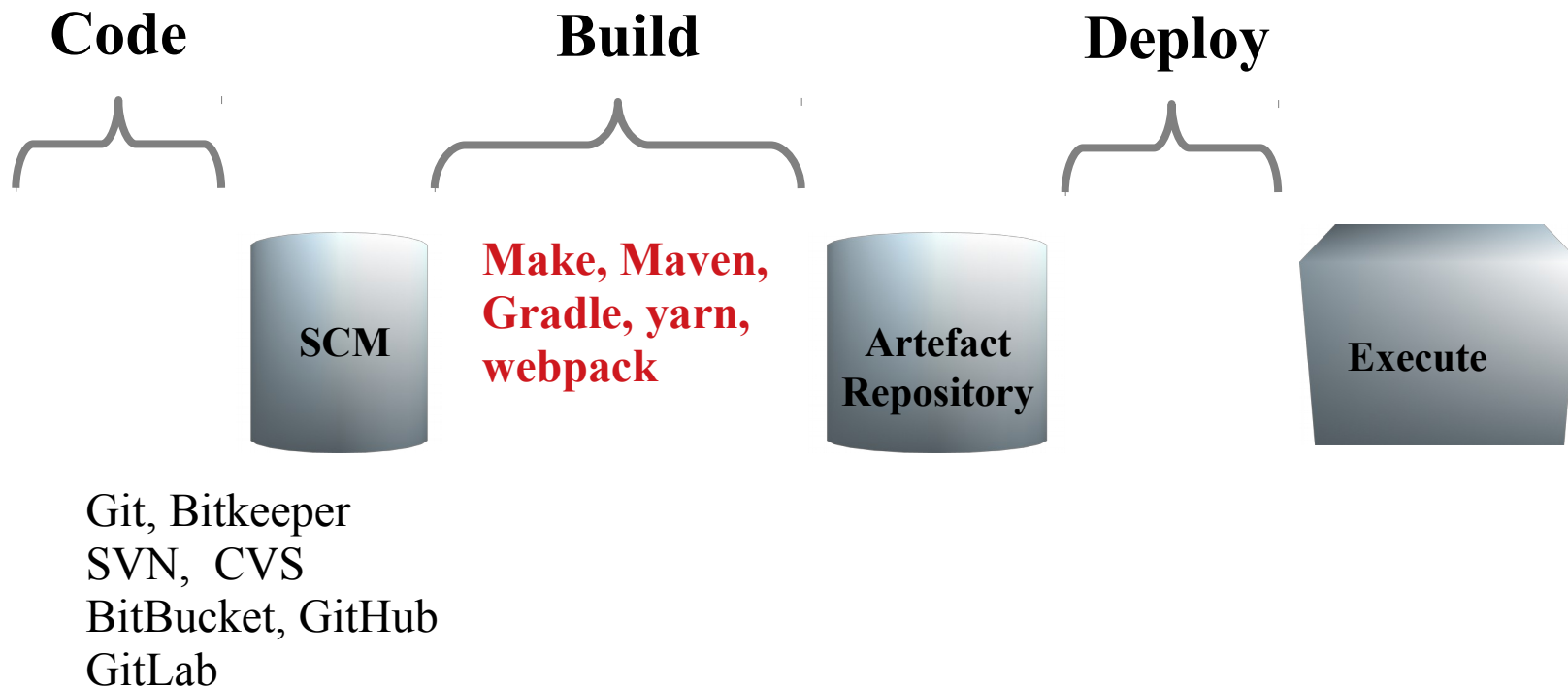
Caractéristiques d'un outil de build
Les tests et la qualité
Release et dépôts d'artefacts



Les outils de build dans le cycle de vie

Plateforme d'intégration continue

Plateforme de livraison





Pré-requis d'un build

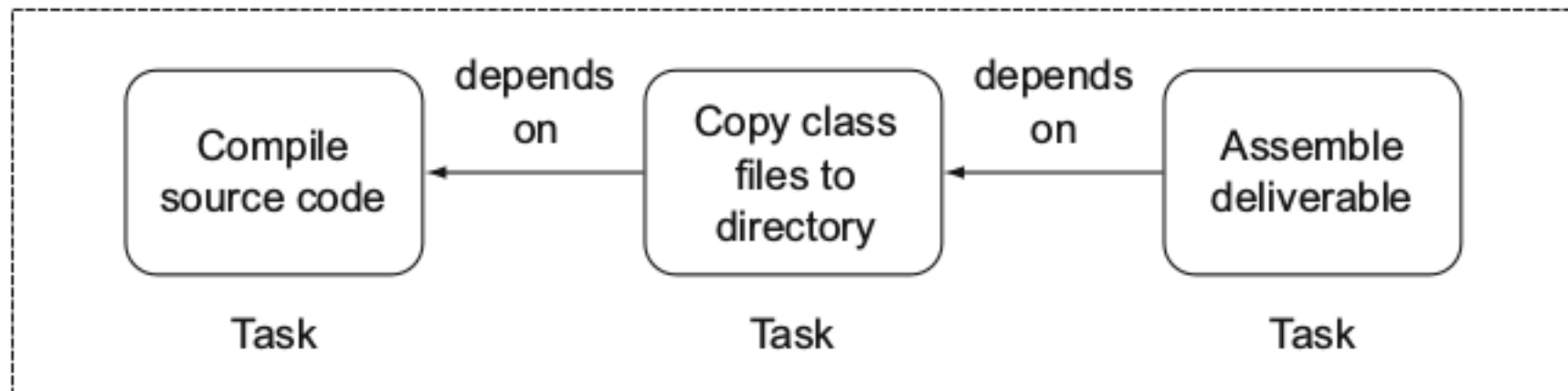
- **Proscrire les interventions manuelles** sujettes à erreur et chronophage
- Créer des builds **reproductibles** : Pour tout le monde qui exécute le build
- **Portable** : Ne doit pas nécessiter un OS ou un IDE particulier, il doit être exécutable en ligne de commande
- **Sûr** : Confiance dans son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



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** :





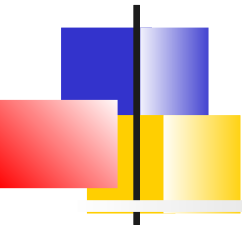
Composants d'un outil de build

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 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 : L'actuel. Gestionnaire de dépendances. Convention plutôt que configuration. Extension par mécanisme de plugin. Verbeux car XML

Gradle : Futur ? Gestionnaire de dépendances. Allie les qualités de Maven et des capacités de codage du build. Concis



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

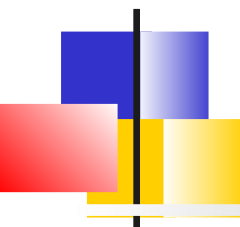
webpack : Création de bundle pour la production

grunt : Automatisation de tâches

gulp : Optimisation, Minification de code

yeoman : Générateur de code

tslint : Analyse de code source, règles de codage

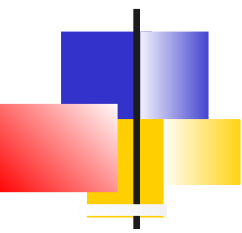


Exemple *package.json*

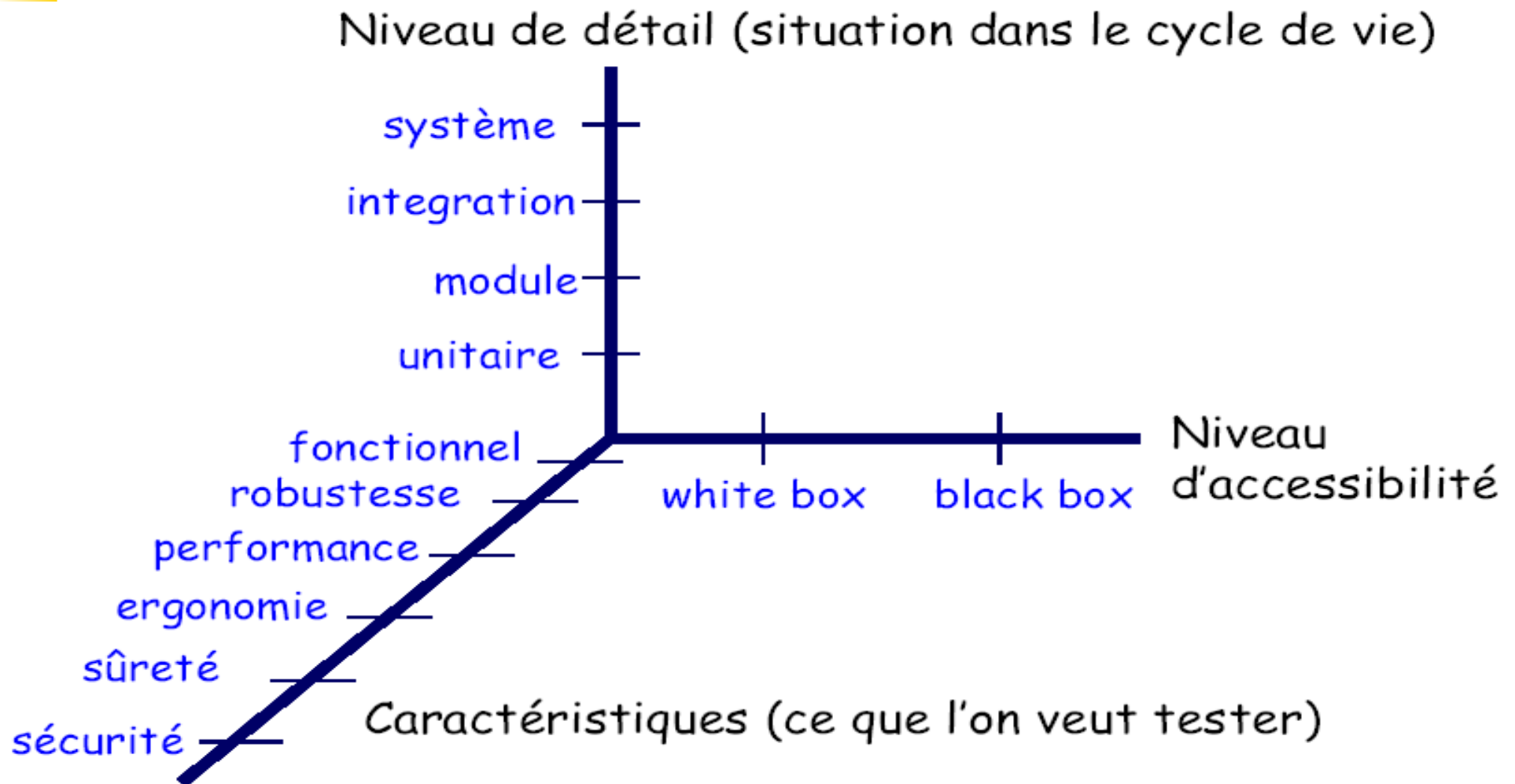
```
{  // Identification projet
  "name": "forum",
  "version": "0.0.0",
  "description": "Description for forum",
  "private": true,
  "license": "UNLICENSED",
  "cacheDirectories": [
    "node_modules"
  ],  // Dépendances
  "dependencies": {
    "@angular/common": "4.3.2",
    "@angular/forms": "4.3.2",
    "@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",
  }
}
```



Tests et qualité



Types de test





Types de test

Test Unitaire :

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

Test d'intégration :

Est-ce que plusieurs classes/couches fonctionnent ensemble ?

Test fonctionnel :

Est-ce que mon application fonctionne ?

Test de performance :

Est-ce que mon application fonctionne bien ?

Test d'acceptance :

Est-ce que mon client aime mon application ?



Le framework : JUnit

Framework fourni par les gourous à l'origine de XP...

Originellement destiné à la plate-forme Java mais depuis porté dans d'autres langages (C/C++/.NET etc..)

- Au départ, uniquement accès sur les tests unitaires...
- Mais aussi les tests d'intégration



Isolation et Mock objects

Les parties à tester doivent être isolées

- Tests peuvent être réalisés même si les parties dont dépend le code ne sont pas encore développées
- Permet d'éviter les effets de bord
- Permet de tester le code lorsque les parties dont il dépend ont des erreurs

=> Utilisation de Mock Objects simulant les interfaces



Exemple *MockBean*

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```



Test d'intégration

Les **tests d'intégration** font participer plusieurs composants du système.

Par exemple, la couche contrôleur avec la couche DAO

– Ils sont généralement plus lourds et plus lents que les tests unitaires car ils nécessitent :

- Une infrastructure plus lourde (Serveur applicatifs, Base de donnée, ...)
- Une préparation généralement plus lourde des données de test

=> Pour les accélérer, on utilise des serveurs ou base de données embarqués ou mockés



Tests fonctionnels

Les **tests fonctionnels** sont des tests en boîte noire qui exécutent des scénarios d'usage de l'application et vérifient leur conformité

- Ils sont fortement dépendants de l'interface utilisateur et de ce fait sont difficiles à automatiser et maintenir
- Dans le cas d'une application web, ils simulent ou pilotent un navigateur et vérifient les réponses fournies par le serveur



Exemple : Selenium Web Driver

```
public class MonTest {
    public static void main(String[] args) {
        // Créer une nouvelle instance de Firefox driver
        WebDriver driver = new FirefoxDriver(); // Utiliser ca pour visiter Google
        driver.get("http://www.google.com");
        // Déterminer le champ dont le name est q
        WebElement element = driver.findElement(By.name("q"));
        element.sendKeys("Selenium"); // Taper le mot à chercher
        element.submit(); // Envoyer la formulaire
        System.out.println("Page title is: " + driver.getTitle()); // Verifier le titre de la page
        // Google fait la recherche dynamique avec JavaScript.
        // Attendre le chargement de la page de 10 secondes
        // Verifiez le titre "Selenium - Recherche Google"
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d)
            {return d.getTitle().toLowerCase().startsWith("selenium");}
        });
        System.out.println("Page title is: " + driver.getTitle());
        //Fermer le navigateur
        driver.quit();
    }
}
```




Tests de performance

Les **tests de performance ou de charge** mesurent les temps de réponse ou débit d'un système en fonction de sa sollicitation.

- Ils sont itératifs et permettent d'optimiser l'usage de l'application.
- Ils nécessitent la mise en place :
 - de bancs de test (Isolation, sondes, instrumentation)
 - de benchmark (Pour comparer les optimisations)
 - la définition d'un modèle de charge (Anticiper les charges en production)
- Outils OpenSource : *Apache JMeter, Gatling*



Tests d'acceptance

Les **tests d'acceptance** ont pour but de valider que la spécification initiale est bien respectée

- Ils sont mis au point avec le client, le testeur et les développeurs
- Dans les méthodes agiles, ils complètent et valident une « User Story »
- Avec l'approche BDD (*Behaviour Driven Development*), l'expression des tests peut être faite en langage naturel.



Exemple Gherkin

Feature: Refund item

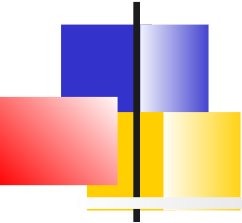
Scenario: Jeff returns a faulty microwave

Given Jeff has bought a microwave for \$100

And he has a receipt

When he returns the microwave

Then Jeff should be refunded \$100



ISO 25010

SOFTWARE PRODUCT QUALITY

Functional Suitability

- Functional Completeness
- Functional Correctness
- Functional Appropriateness

iso25000.com

Performance Efficiency

- Time Behaviour
- Resource Utilization
- Capacity

Compatibility

- Co-existence
- Interoperability

Usability

- Appropriateness
- Recognizability
- Learnability
- Operability
- User Error Protection
- User Interface Aesthetics
- Accessibility

Reliability

- Maturity
- Availability
- Fault Tolerance
- Recoverability

Security

- Confidentiality
- Integrity
- Non-repudiation
- Authenticity
- Accountability

Maintainability

- Modularity
- Reusability
- Analysability
- Modifiability
- Testability

Portability

- Adaptability
- Installability
- Replaceability



Métriques internes et Outils Qualité

SonarQube est la plate-forme qui regroupe tous les outils de calcul de métrique interne d'un logiciel (toute technologie confondue)

Il intègre 2 aspects :

- Détection des transgressions de règles de codage et estimation de la dette technique
- Calculs des métriques internes et définition de porte qualité

Sa mise en place nécessite une adaptation en fonction du projet.



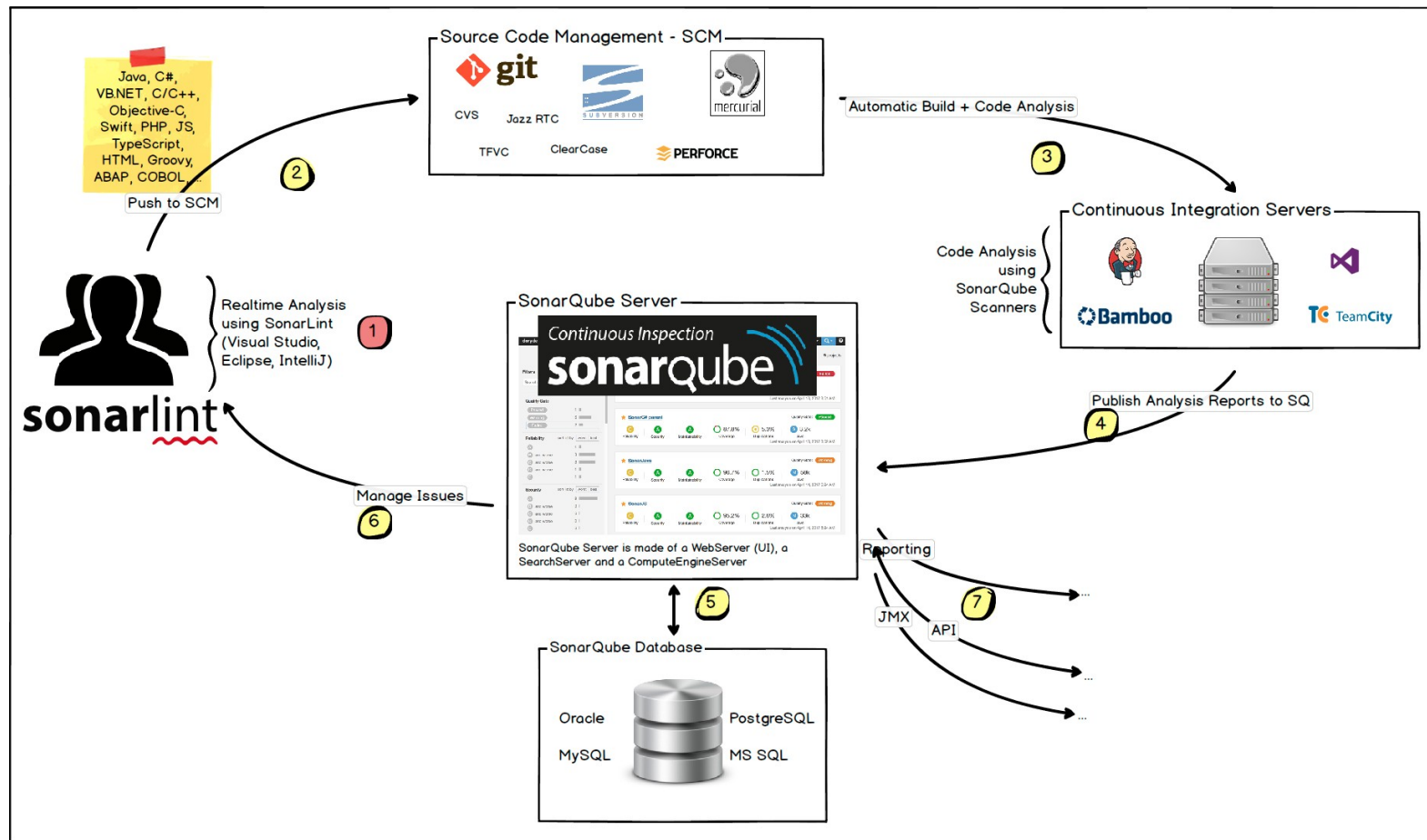
CI et Porte qualité

Les **portes qualité** définissent un ensemble de seuils pour les différents métriques. Le dépassement d'un seuil :

- Déclenche un avertissement
- Empêche la production d'une release.

SonarQube fournit des portes par défaut qui sont adaptées en fonction du projet.

Analyse continue





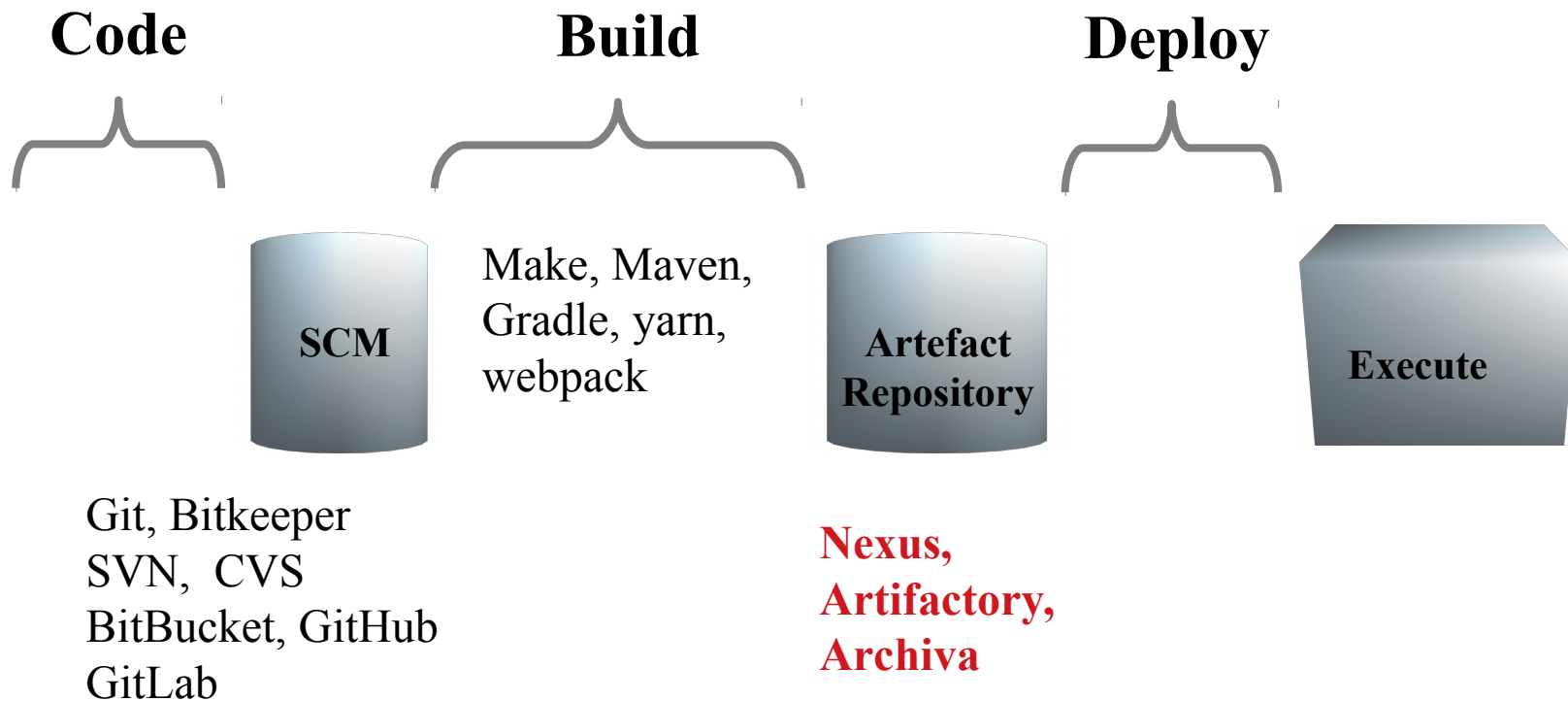
Release et dépôt d'artefacts



Les gestionnaires de dépôts dans le cycle de vie

Plateforme d'intégration continue

Plateforme de livraison





Dépôts d'artefacts

Les artefacts ou packages produits sont stockés dans des dépôts.

- **publics** (Maven, NPM, Aptitude, Yum,) permettent de récupérer des produits Open Source...
- **privés**, propres à une organisation ou entreprise. Ils contiennent les productions de l'entreprise et font souvent office de proxy des dépôts publics



Dépôt privé

La mise en place d'un dépôt interne à l'entreprise apporte :

- De la sécurité, contrôle des codes utilisés
- Optimise la bande passante
- Permet de publier les artefacts produits par la société.

Les outils spécialisés sont *Nexus*, *Artifactory*, *Archiva*. Ils s'intègrent avec différents types de dépôts



La release

La release consiste à permettre que l'artefact généré par un commit particulier puisse aller en production.

En général cela consiste à :

- Générer l'artefact et s'assurer que les tests d'acceptance sont OK
- Intégrer la branche de travail dans la branche de production
- Tagger le commit avec un n° de version
- Déposer l'artefact dans le dépôt d'artefact
- Modifier les sources afin de modifier le n° de version les opérations avec le SCM.



Automatisation

L'automatisation de la release doit être assumée par la plateforme d'intégration continue

La PIC s'appuie souvent sur des fonctionnalités de l'outil de build (Par exemple Maven)



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



Concepts

Nexus manipule des **dépôts** et des **composants**

- Les composants sont des artefacts identifiés par leurs coordonnées. Nexus permet d'y attacher des méta-données Pour être générique envers tous les types de composants. Nexus utilise le terme asset
- Les dépôts peuvent être des dépôts Maven, npm, RubyGems, ...



Plateforme d'intégration continue

Concepts communs
Pipelines typiques
Solutions

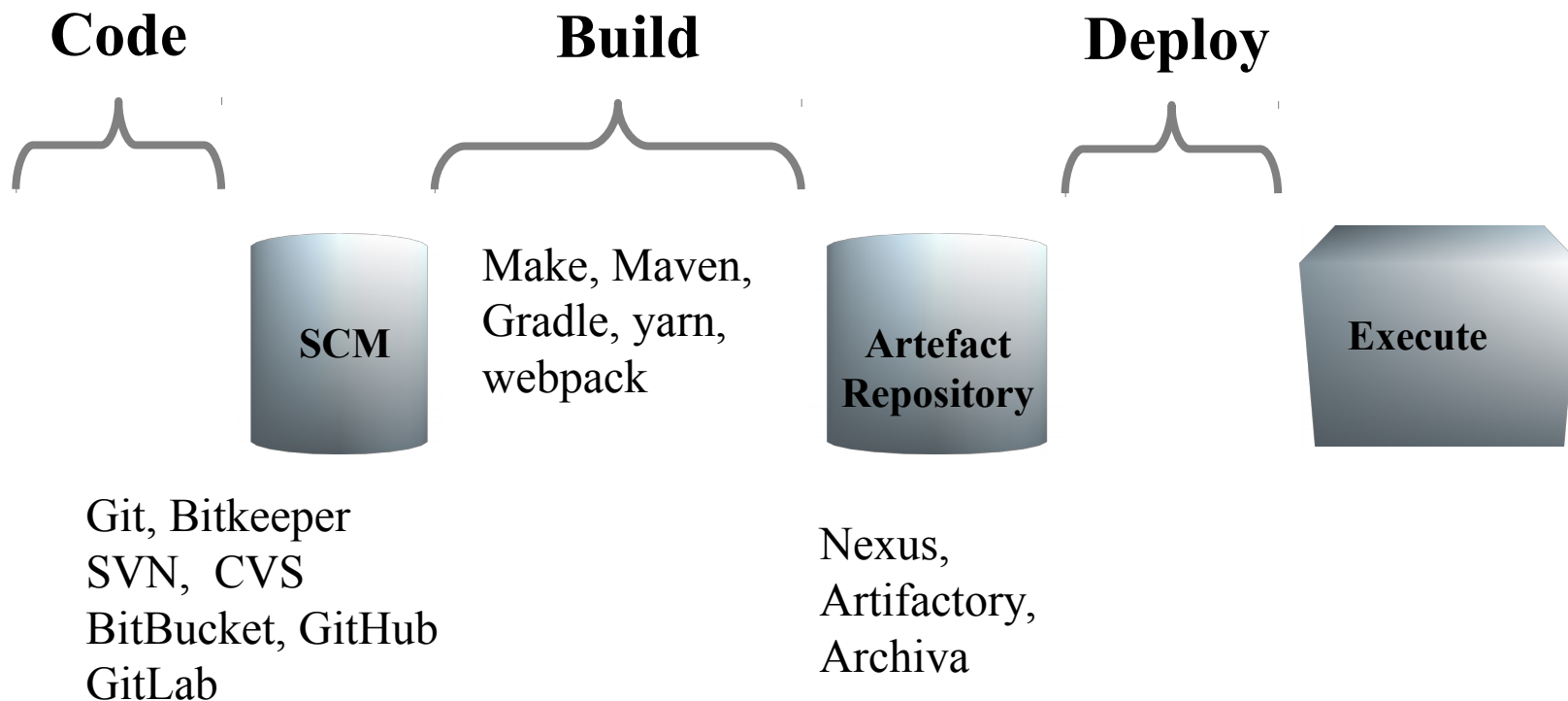


PICs dans le Cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

Plateforme de livraison





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 **machines virtuelles ou conteneurs** qui sont alors provisionnés/exécutés lors du build et qui disparaissent ensuite



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
- Via docker



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 les jobs 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 pire l'infrastructure de PIC nécessaire au projets (plugins et autre) est elle même dans le SCM et peut être automatisé



Pipelines typiques



Principes

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

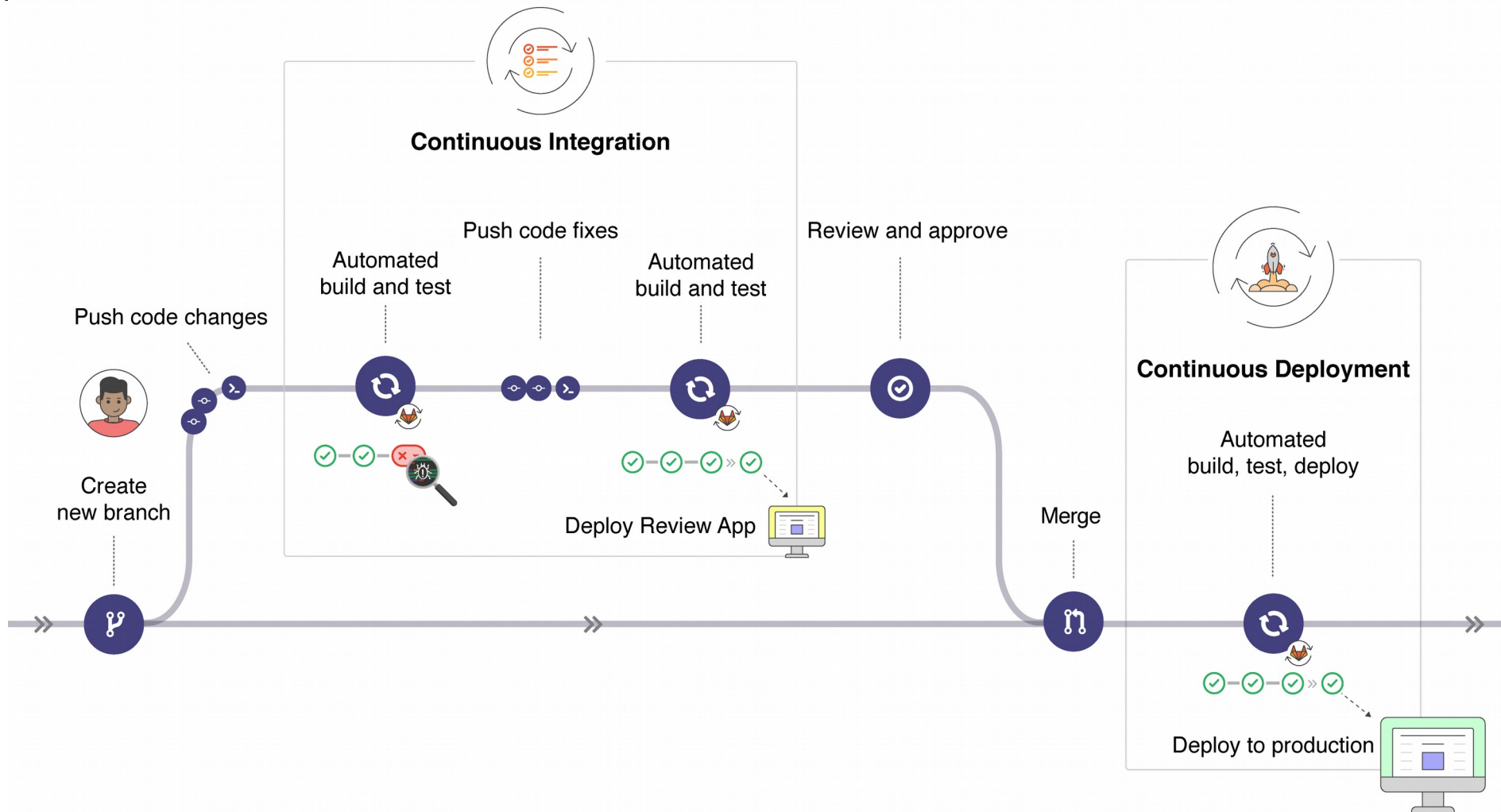
Par exemple : Build, Test, QA, Production

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

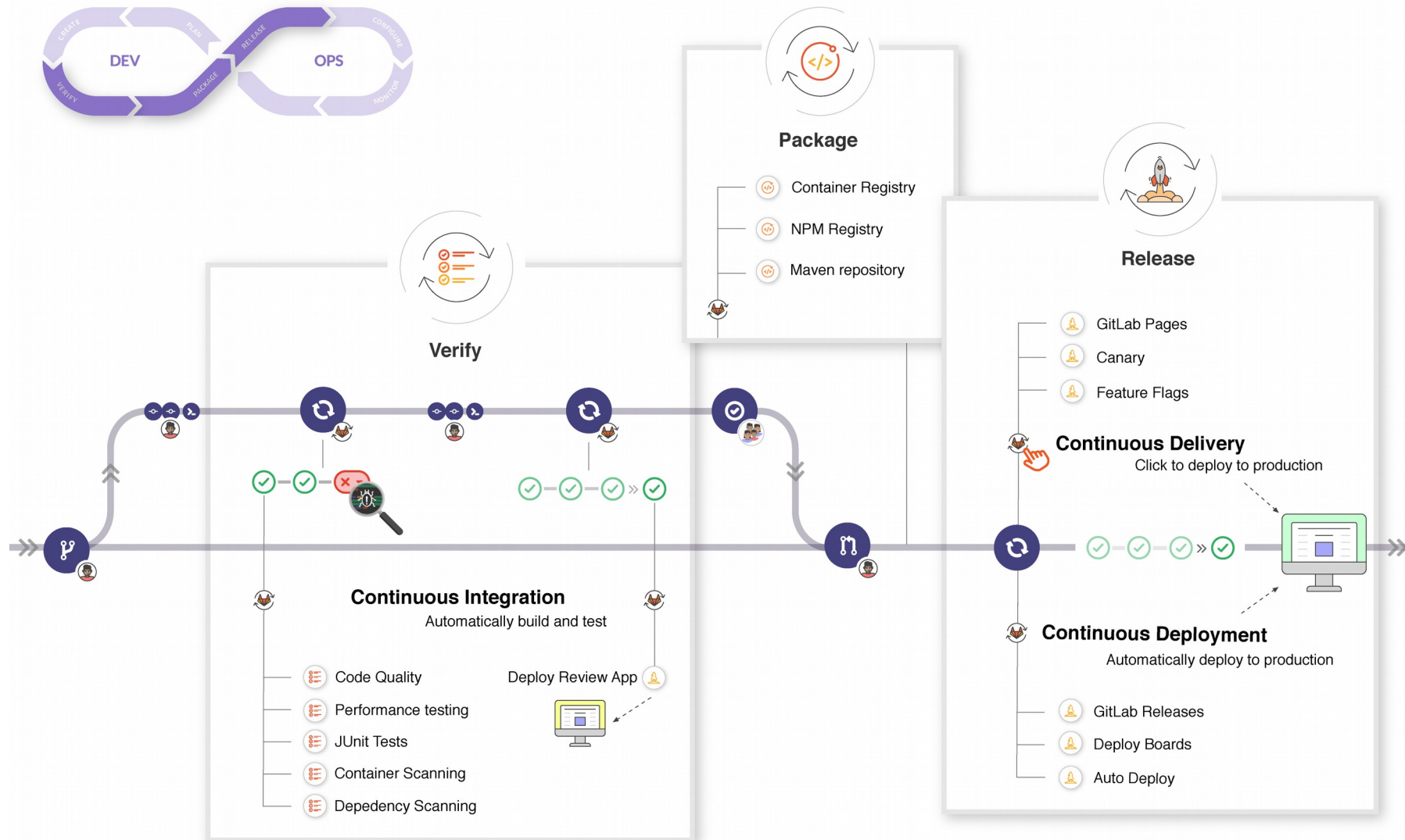
Les jobs exécutent des actions à partir du dépôt ou réutilisent des artefacts précédemment construits

Les jobs publient des rapports résultats

Exemple Gitlab CI



En plus détaillé





Solutions



Gitlab

GitLab est l'interface de gestion des projets Git (utilisateurs, projets, pull-requests, forks...)

Elle propose aussi une gestion de rapport d'anomalie qui réagit aux messages de commit

Gitlab est Disponible en version communautaire et opensource



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 pipeline. Les outils nécessaires pour le build sont pré-installés ou des images Docker sont utilisés.
- Les pipeline 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



Strider

Strider est développé en NodeJS et utilise MongoDB pour la persistance.

Des plugins lui permettent de s'intégrer à BitBucket , GitHub , ou des dépôts privés.

Son interface est légère et fluide

Adapté aux petites équipes développant en *NodeJS*, tout comme en Python, Go, Java et Ruby.

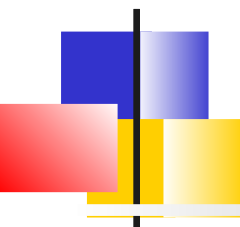


Plugins



































Strider est personnalisable via des plugins

Un plugin peut :

- Ajouter des hooks pour exécuter des actions durant le build.
- Modifier le schéma de la base pour y ajouter des champs
- Enregistrer des nouvelles routes HTTP.
- S'abonner à ou émettre des événements socket.
- Créer ou modifier l'interface Strider.



Strider

Strider				
Dashboard Projects Account Admin				
Latest Builds:				
Job	Project		Commit	Duration
 51f458a7		jaredly/codetalker		 135s <div><div></div></div>
 51f45893	 	notablemind/note		 5s 2 minutes ago
  51f45498	 	Strider-CD/strider	 jshint fixes	 481s 11 minutes ago
 51f2db75	 	Strider-CD/strider-simple-worker	 Merge pull request #15 from Strider-CD/c	 35s a day ago
 51f2b384	 	Strider-CD/strider-travis		 6s a day ago
 51eec859	 	jaredly/org-lite		 1s 4 days ago
 51eec4e5	 	jaredly/search		 607s 4 days ago
 51eb90b1	 	notablemind/socketio-monitor		 1s 7 days ago



Travis CI

Travis CI fournit un service en ligne utilisé pour compiler, tester et déployer le code source déposé sur GitHub

La configuration s'effectue via ***travis.yml***, qui spécifie le langage de programmation et l'environnement de build et de test

Dashboard

Travis CI

About UsBlogStatusHelp

Sam Iamm

Search all repositories

My Repositories

✓ one-fish/two-fish # 2686

Duration: 33 min 46 sec

Finished: 30 minutes ago

✓ hop-on/pop # 7001

Duration: 22 min 54 sec

Finished: about an hour ago

✓ horton-hears/awho # 209

Duration: 53 sec

Finished: about 2 hours ago

✓ green-eggs/ham # 209

Duration: 53 sec

Finished: about 2 hours ago

✓ onthe/places-youllgo # 778

green-eggs / ham

build passing

CurrentBranchesBuild HistoryPull Requests

More options

✓ master adding in Oh the places you'll go!

~ #209 passed

Restart build

You'll be on your way up!

You'll be seeing great sights!

Commit abc123

Branch master

Sven Fuchs authored and committed

Ran for 53 sec

about 2 hours ago

Job log

View config

Remove log

Raw log

1 Worker information

6 mode of '/usr/local/clang-5.0.0/bin' changed from 0777 (rwxrwxrwx) to 0775 (rwxrwxr-x)

7 Build system information

8 Build language: node_js

9 Build group: stable

10 Build dist: trusty

11 Build id: 345296935

12 Build id: 345296935



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

```
#!/groovy
stage('Init') {
    node {
        checkout scm
        gitCommit = sh(returnStdout: true, script: 'git rev-parse HEAD').trim()
    }
}
stage('Build') {
    parallel frontend : {
        node {
            checkout([$class: 'GitSCM', branches: [[name: gitCommit ]],
                dir("plbsi-front") {
                    sh 'npm install' sh './node_modules/.bin/ng build -e prod'
                    stash includes: 'dist/**', name: 'front'
                }
            ] } },
        backend : {
            node {
                checkout([$class: 'GitSCM', branches: [[name: gitCommit ]],
                    sh 'mvn clean test'
                ] } }
        }
    }
stage('Integration') {
    node {
        dir("plbsi-rest/src/main/resources/static") { unstash 'front' }
        sh 'sudo cp plbsi-rest/target/plbsi-rest-4.0.0-SNAPSHOT.jar /home/plbsi/bin/plbsi-rest.jar'
        sh 'sudo /etc/init.d/plbsi-rest.sh start'
    }
}
```




Virtualisation et gestion de conf.

Solutions de virtualisation

Provisionnement, outils de base

Solutions de gestion de configuration

Offre de Cloud

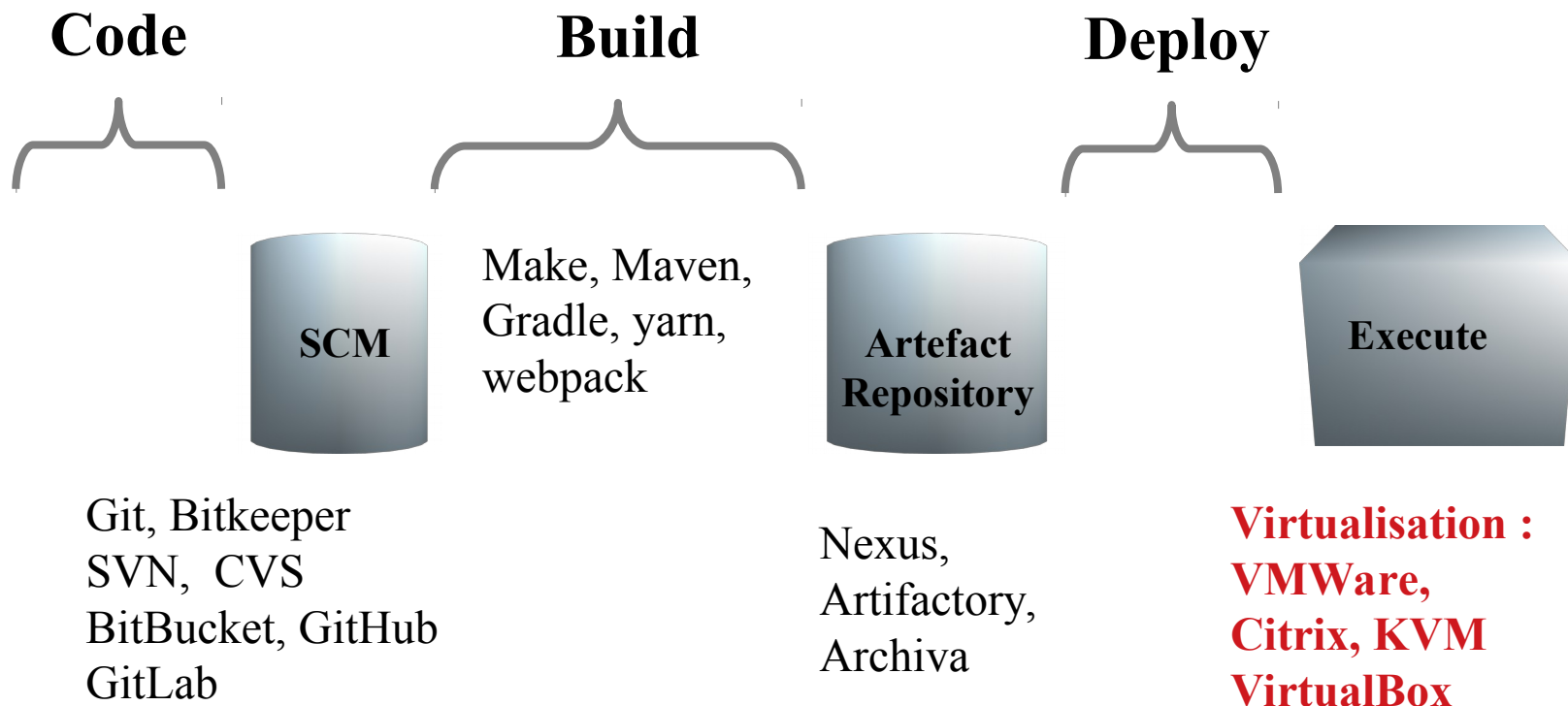


La virtualisation dans le cycle de vie

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

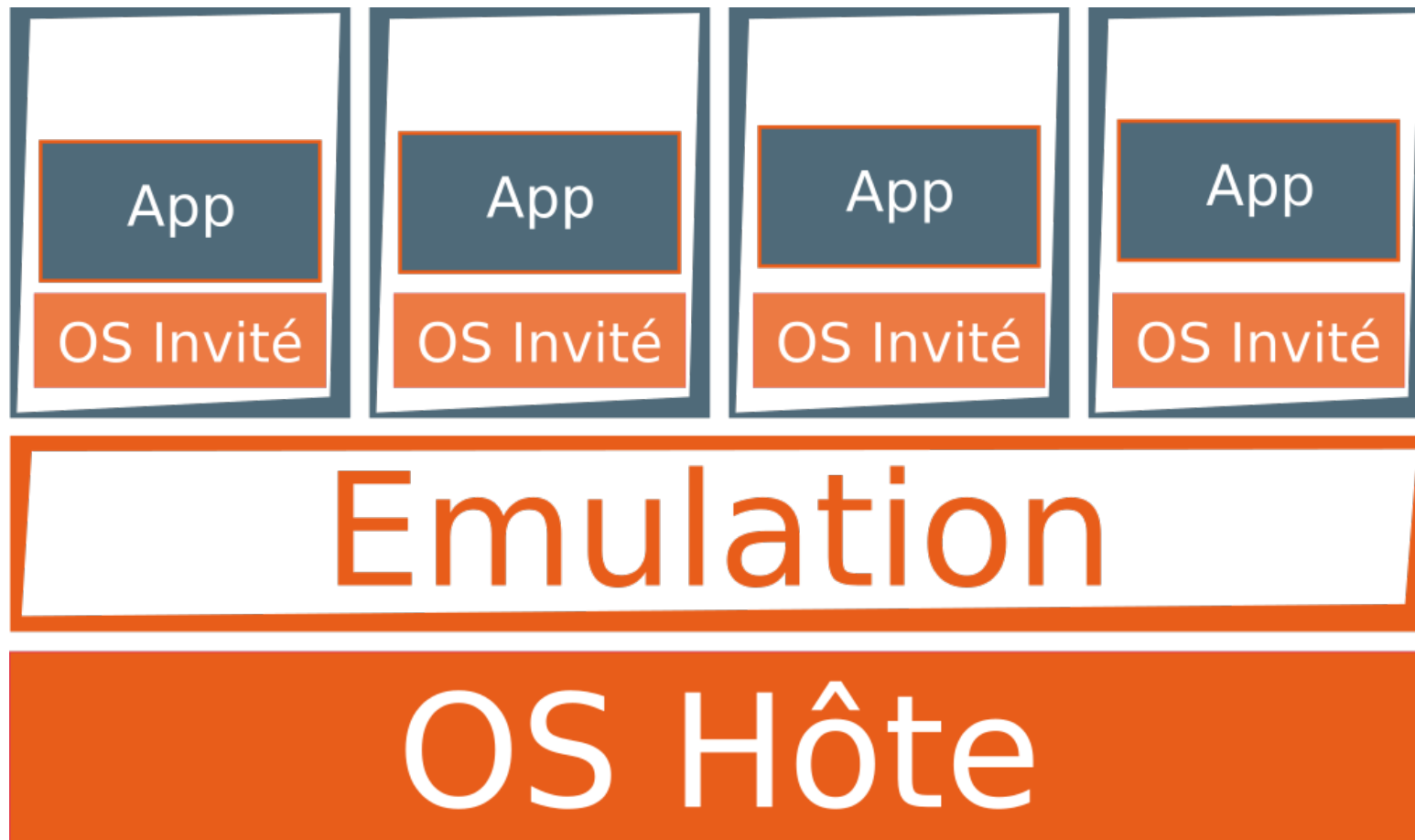
Pour virtualiser un environnement complet le serveur doit être capable

- 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



Fonctionnalités Hyper-V

Virtualisation imbriquée

Affectation individuelle de device

Sauvegarde dans le cloud

Supporte la containerisation

Quality of service (QoS) pour les réseaux et le stockage

Redimensionnement à chaud des disques, CPU, mémoire

Migration à chaud

Réplication

Sécurité avec Windows Active Directory

Supporte Windows PowerShell)



Composants VMWare

vCenter Server: Outil de gestion centralisé

vSphere Client: Accès au serveur à partir d'une station de travail

vSphere SDKs: Interfaces pour des solutions tierces vSphere.

VM File System: Système de fichiers en cluster pour les VMs.

Virtual SMP: Permet à une VM d'utiliser plusieurs processeurs physiques en même temps.

VMotion: Permet une migration à chaud en garantissant une intégrité transactionnelle .

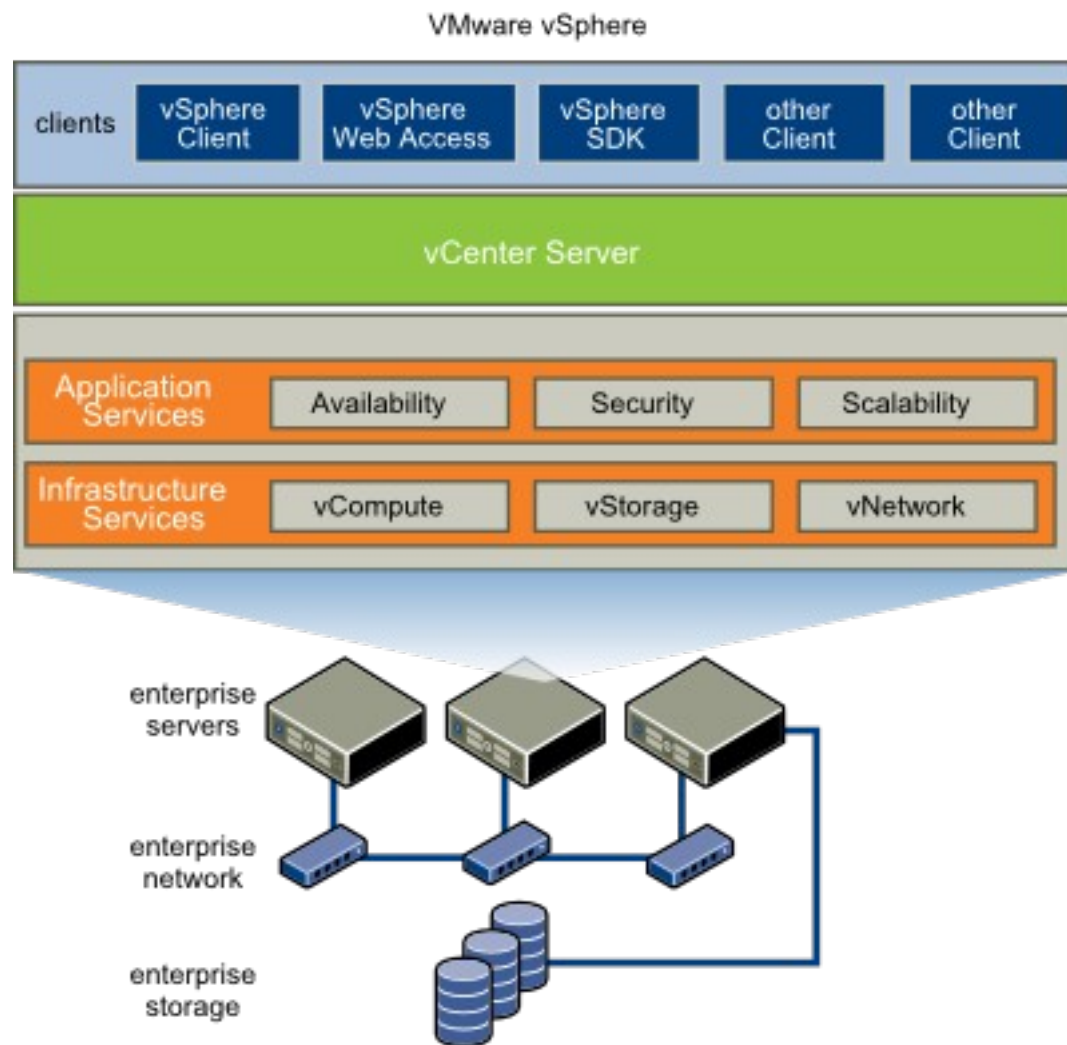
Storage vMotion: Permet la migration de fichiers d'une VM sans interruption de service.

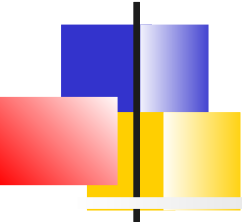
Haute Disponibilité: Si un serveur défaille, la VM est déplacé sur un autre serveur sans interruption de service

Distributed Resource Scheduler (DRS): Assigne et répartit les traitements sur les ressources matérielles disponibles .

Fault Tolerance: Génère une copie de la VM primaire

VMWare vSphere





Caractéristiques Citrix Zen Server

Procédure de recovery de site

Protection contre les défaillances de serveur

Multi-server management

Contrôle dynamique de la mémoire

Intégration avec Active Directory

Administration basée sur des rôles (RBAC)

Partage des ressources CPU

Cache mémoire

Migration à chaud des VM et du Stockage (XenMotion)



KVM

Support pour les conteneurs

Scalabilité

Surbooking automatique des ressources => gain de performance

Contrôle des accès disque d'une VM

Solution Low cost

API de programmation

Migration à chaud (VM et stockage)

Affectation de device PCI aux VMs

Intégration à Red Hat Satellite (outil de gestion de qualité et sécurité)

Support pour le recovery



Gestion de configuration et provisionnement

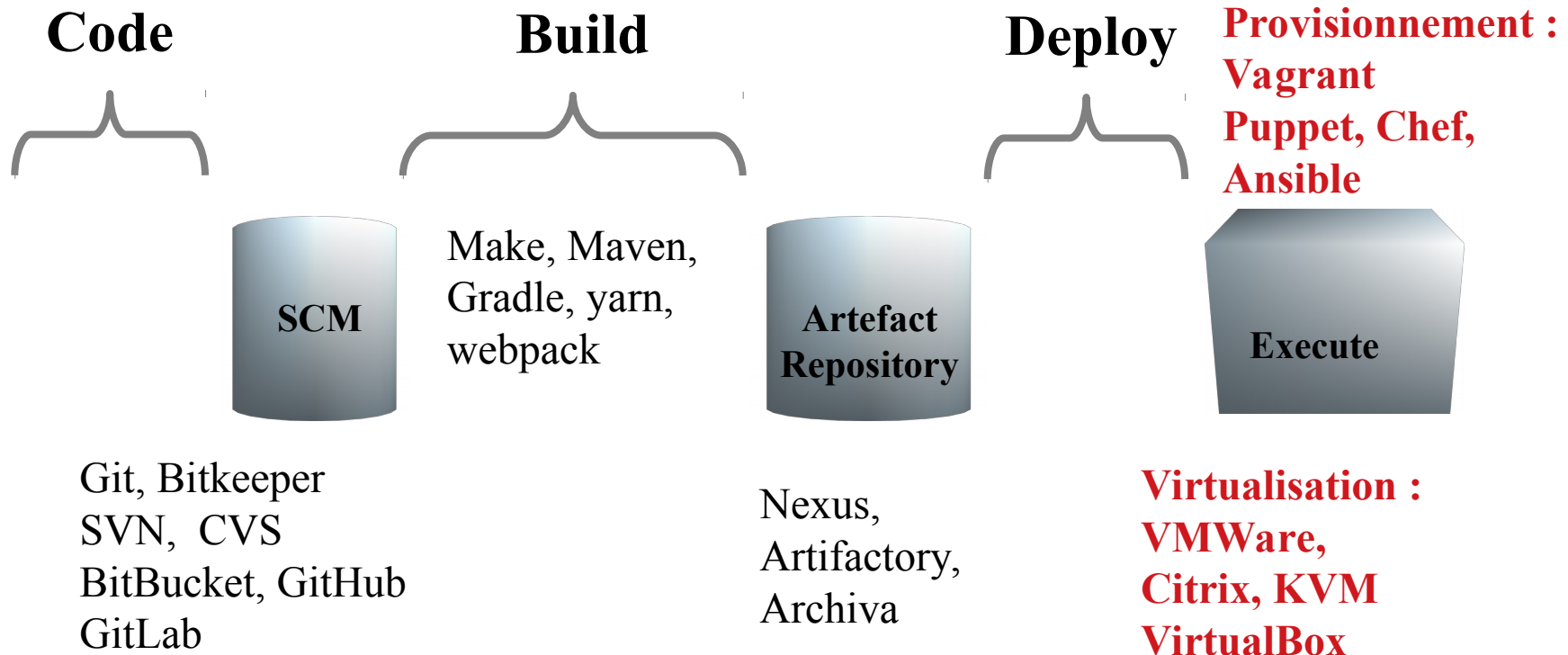


Provisionnement dans le cycle de vie

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



Outils de base

SaltStack est un logiciel de gestion de configuration écrit en Python, fonctionnant sur le principe client-serveur. Il utilise le format YAML et Jinja2

Fabric est un outil et une librairie Python qui permet le déploiement et la gestion de tâches administratives locales ou via SSH sur différents serveurs.



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



Vagrant boxes

Plutôt que de redéfinir intégralement une machine virtuelle, il est possible de partir d'une image de base téléchargeable facilement. Les ***vagrant boxes***

A partir d'une image de base, on peut provisionner la machine avec un script shell séparé contenant toutes les installations.



Isolation de la machine

Par défaut, un répertoire sur la machine virtuelle est montée sur un disque du hôte

Des directives de configuration permettent d'associer des ports du hôte au port la machine virtuelle, on peut également lui affecter une IP fixe, ou la raccorder à une réseau existant



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

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



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 centralisée 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**"

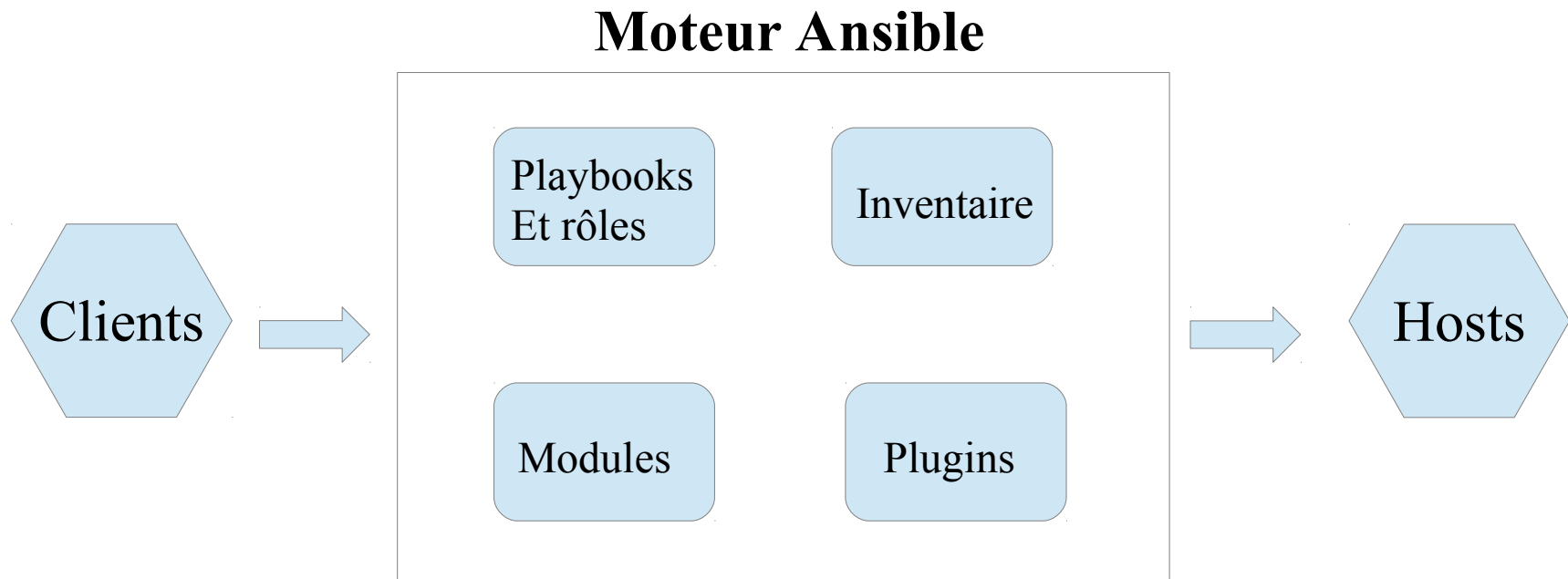
- Ansible exécute les modules et les enlève une fois l'exécution terminée
- Les modules Ansible peuvent être écrits dans n'importe quel langage du moment qu'ils retournent du JSON. Ils sont idempotents.

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.



Example

```
---
- 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
```




Roles

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

Un rôle peut contenir :

- **files** : Les fichiers à copier sur la cible
- **handlers**
- **meta** : Dépendances
- **template**
- **variables**
- **tasks**



Arborescence classique

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
```



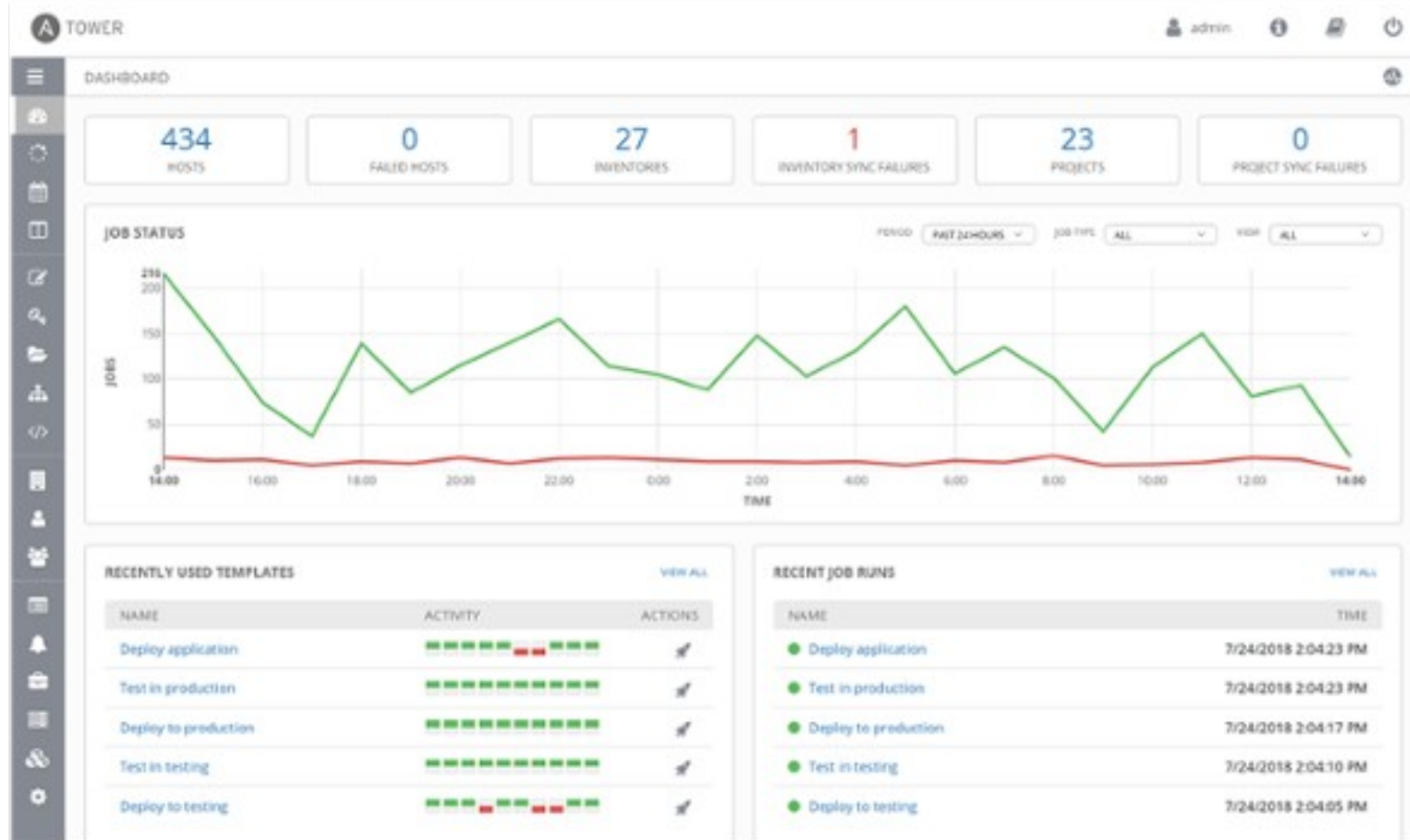
Ansible Tower

Ansible a été racheté par RedHat qui offre des produits d'entreprise autour de la solution

Ansible Tower par exemple offre

- une interface Web permettant de contrôler l'infrastructure et les déploiements
- Planification de job
- Notifications
- API Rest et CLI pour l'automatisation

Snapshot





Containerisation

Avantages de la containerisation
Architecture docker et principales
commande
Isolation et Docker-compose
Intégration PIC et Docker



Introduction

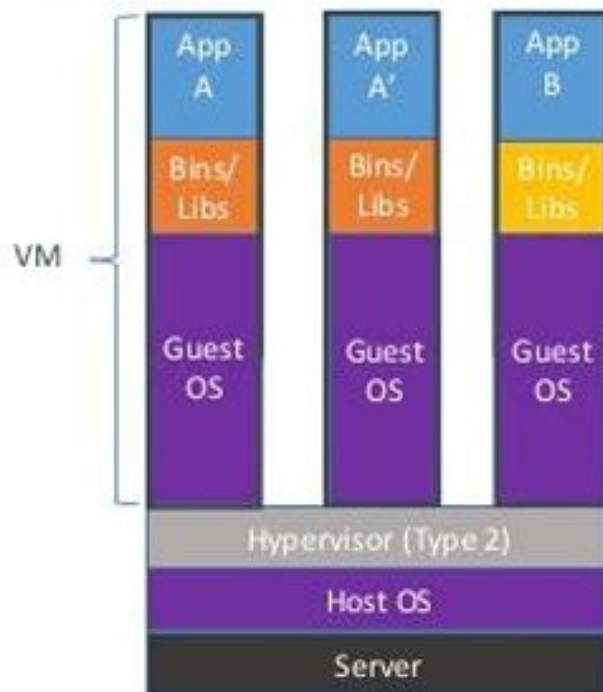
Plutôt que de virtualiser une machine complète, il n'est nécessaire que de créer un environnement d'exécution dans lequel un service est démarré pour fournir l'application.

Le but des conteneurs applicatifs est donc d'isoler un processus dans un système de fichiers à part entière - il n'est plus question de simuler le matériel et les services d'initialisation du système d'exploitation sont ignorés.

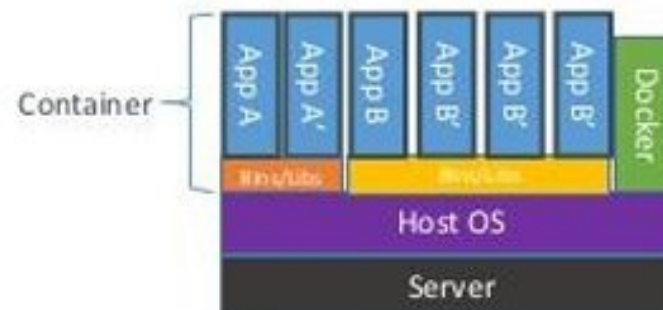
Seul le strict nécessaire réside dans le conteneur, à savoir l'application cible et quelques dépendances.

Containers vs VMs

Containers vs. VMs



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





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.



Avantages de la containerisation

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

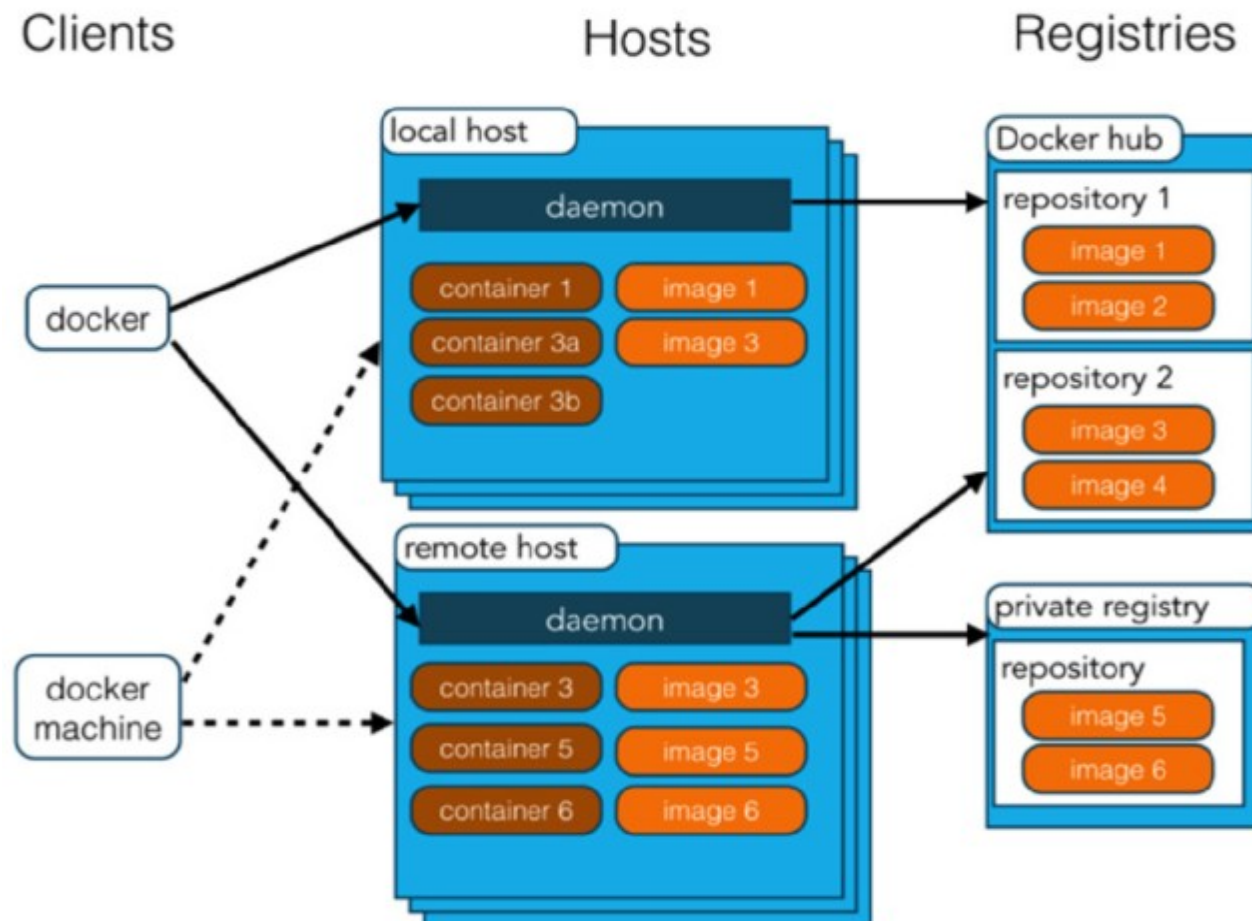
Chargement du container 50 fois plus rapide que le démarrage d'une VM

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

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

Nécessite une approche DevOps

Docker architecture





Commandes Docker

#Récupération d'une image

```
docker pull ubuntu
```

#Récupération et instanciation

```
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"]



Isolation et communications du conteneur



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
- Des autres containers



Communication réseau avec la machine hôte

Pour communiquer à la machine hôte, en général l'image déclare les ports utilisés dans son fichier *Dockerfile* (en fait ce n'est qu'informatif)

EXPOSE 8080

Ensuite au démarrage d'un conteneur on associe le port exposé à un port disponible de la machine hôte.

```
docker run -p 80:8080 myImage
```



Partage de fichiers avec la machine hôte

De la même façon, il est possible de partager des répertoires entre l'hôte et le conteneur.

```
docker run -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.

Docker s'installe séparément sur Linux et est inclus dans Docker pour les distributions Mac ou Windows

Orienté au départ pour faciliter le développement et l'intégration ; il intègre de plus en plus des fonctionnalités pour la production



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

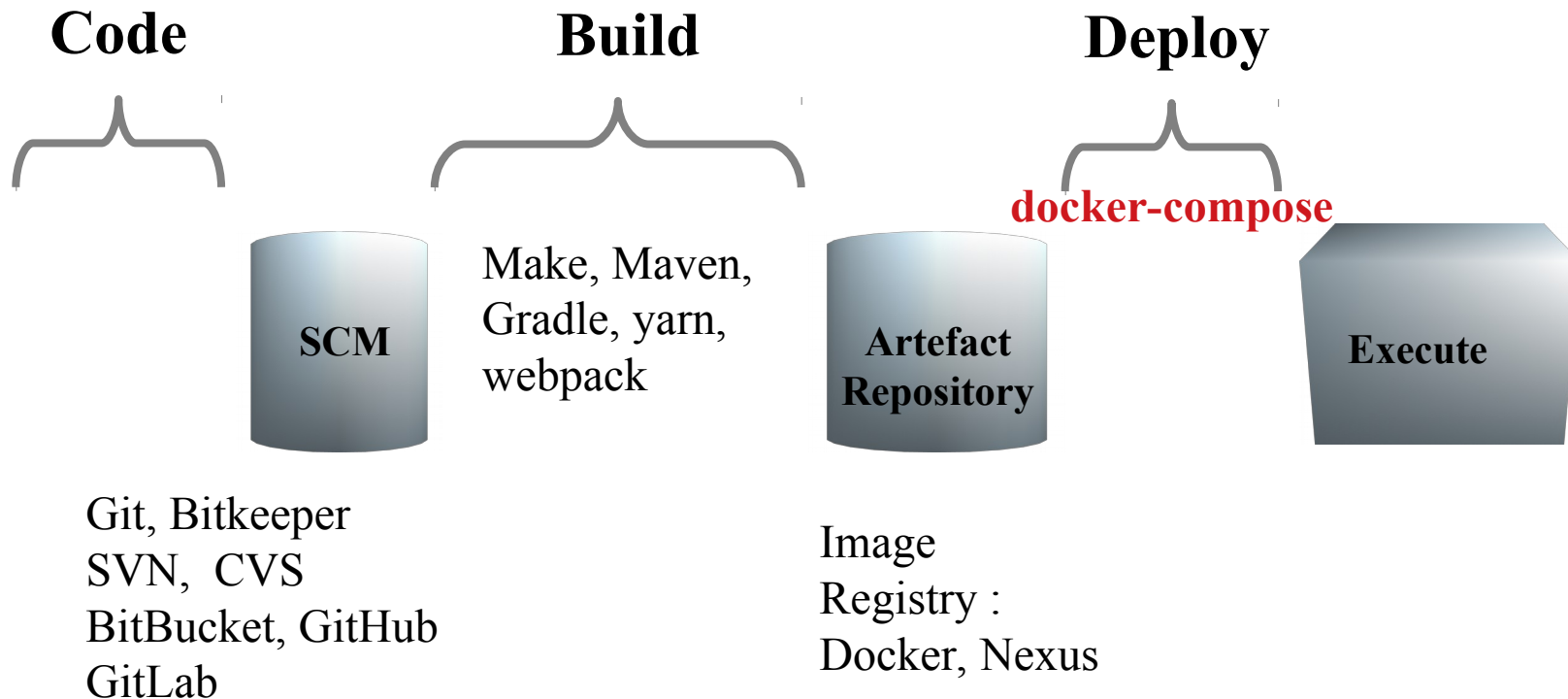


Paradigme docker

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

Plateforme de livraison



Artefacts :
Image Docker



CI/CD et Docker



PIC et Docker

Une PIC utilise docker de plusieurs façons :

- Les workers utilise des images pour exécuter des tâches du build ou pour exécuter des services utilisés par le build
- La pipeline a pour but de construire une image et de la pousser dans un dépôt.



Techniques d'intégration

L'intégration peut nécessiter :

- Préinstaller docker sur les runners
- Déclarer des registres d'image et les moyens d'y accéder
- Permettre du docker in docker. (Une container de build démarre un autre container). Utilisation de l'image dind



Exemple Jenkins

```
// Declarative //  
pipeline {  
  agent {  
    docker { image 'node:7-alpine' }  
  }  
  stages {  
    stage('Test') {  
      steps { sh 'node --version' }  
    }  
  }  
}
```




Gestion de cache

Les outils build téléchargent généralement les dépendances externes et les stockent localement pour les réutiliser.

Le fait que les containers sont créés avec des systèmes de fichier vierge ralentit l'exécution des pipelines.

=> Les plateformes de CI/CD proposent d'utiliser des volumes docker pour cacher les données entre 2 builds.



Exemple Jenkins

```
// Declarative //
pipeline {
  agent {
    docker {
      image 'maven:3-alpine'
      args '-v $HOME/.m2:/root/.m2'
    }
  }
  stages {
    stage('Build') {
      Steps { sh 'mvn -B'}
    }
  }
}

// Script //
node {
  docker.image('maven:3-alpine').inside('-v $HOME/.m2:/root/.m2') {
    stage('Build') { sh 'mvn -B' }
  }
}
```



Dockerfile

Pipeline permet également de construire des images à partir d'un Dockerfile du dépôt de source.

Il faut utiliser la syntaxe déclarative :

```
agent { dockerfile true }
```

```
// Declarative //
```

```
pipeline {  
  agent { dockerfile true }  
  stages {  
    stage('Test') { steps {sh 'node --version'} }  
  }  
}
```



Sidecar pattern

sidecar pattern : Exécuter un conteneur en service pendant que les jobs de build l'utilisent. Ex : base de données de test

Exemple GitlabCI

```
image: ruby:2.2
```

```
services:
```

```
- postgres:9.3
```

```
before_script:
```

```
- bundle install
```

```
test:
```

```
script:
```

```
- bundle exec rake spec
```



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

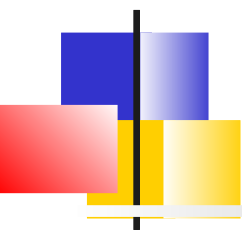
```
#!/usr/bin/env groovy

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'
        }
    }
}
```




Orchestrateur de conteneurs

Orchestrateur, scalabilité et
déploiement

Solutions, exemple Kubernetes

Intégration dans une pipeline CI/CD

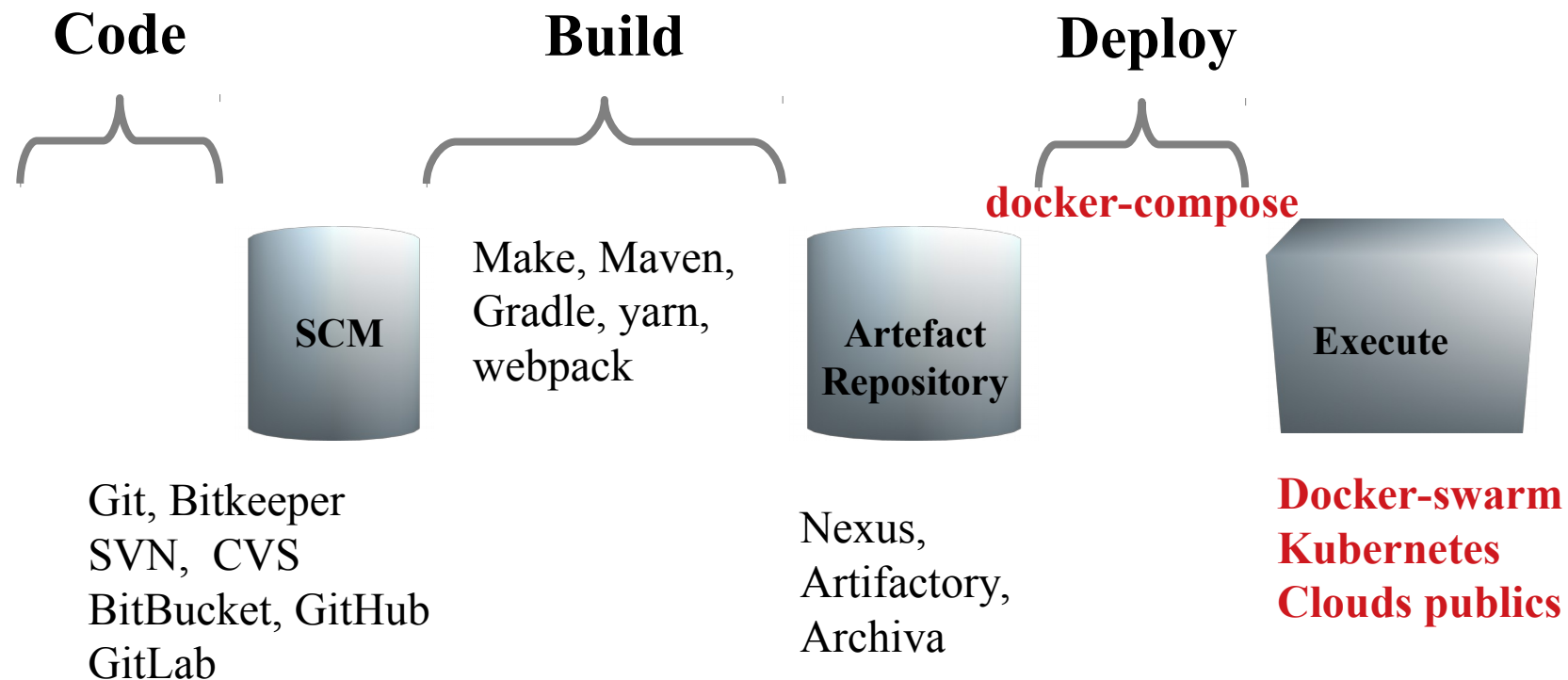


Cloud dans le cycle de vie

Plateforme d'intégration continue

Jenkins, Gitlab CI, Travis CI, Strider

Plateforme de livraison



Artefacts :
Image Docker



Orchestration de conteneurs

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

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

Il va permettre également la "mise à l'échelle" (ou scaling) en fonction de la charge courante.



Orchestrateur

Gère un pool de ressources

Il connaît la topologie du cluster

- Ressources disponibles
- Applications déployées

Il connaît l'état de santé ...

- ...des services
- ...de la plateforme

C'est une évolution des outils

Manuel (Scripts) ® Automatisé (Conformité) ®
Délégué (Orchestration)



Introduction

Une des contraintes des approches DevOps est la scalabilité

Les services applicatifs déployés doivent pouvoir être répliqués à la demande afin de :

- Répondre à la charge
- Résister aux défaillances

Mais également fiabiliser les déploiements

- Déploiement blue-green
- Canary testing



Services de discovery

Les contraintes de réplication impliquent qu'il n'est plus possible d'affecter de façon statique des adresses réseaux aux services applicatifs déployés.

La localisation des services doit être dynamiques et s'appuie sur des services de discovery qui apporte 2 principales fonctions :

- Service registration : Un service qui démarre sur un container ou pod s'enregistre auprès du service de discovery
- Service lookup : Un client voulant accéder à un service s'appuie sur le service de discovery pour localiser une réplique disponible



Réplication du service de discovery

Le service de discovery étant critique à l'ensemble des services déployés, il doit être lui-même répliqué.

Typiquement, il y a un service de discovery par data-center et les services applicatifs ont dans leur configuration le service de discovery préféré.



Solutions

Solutions de Key/Value store distribués => Nécessite de gérer soit même l'enregistrement du services :

– **Zookeeper** (Apache) :

- Un des plus vieux projet de ce type. Implémentation Java.
- Cluster de nœud dont un leader coordonnant les écritures
- Données arborescentes partagées sur tous les nœuds

– **etcd** :

- Même fonctionnalités que Zookeeper mais plus simple à mettre en place
- API http
- Utilisé dans Kubernetes

Enregistrement :

– **Registrar** :

- Enregistre automatiquement les containers Docker
- Intégration avec etcd, Consul et SkyDNS2

Service de discovery complet :

- **Eureka** (Netflix) : Application SpringBoot, healthcheck via heartbeat
- **Consul** : Plus robuste, adapté au cluster, healthcheck via protocol de gossip réduisant le trafic réseau



Service de proxy

Un service **proxy** est un service qui sert d'intermédiaire entre les clients effectuant des requêtes et les serveurs qui leur répondent :

- Un client envoie une demande au service proxy qui, à son tour, redirige cette demande au service de destination
- Simplifie et contrôle la complexité de l'architecture distribuée



Apports du proxy

Point d'entrée unique des API publiques exposées

Routage : répartition de charge, canary testing

Sécurité :

- Seul les ports 80 et SSL sont exposés
- Mécanismes d'authentification du client (oAuth, ...)
- Gestion des entêtes de sécurité http

Cache : Cache du résultats des requêtes, gestion des dates d'expiration

Compression : Compression des données échangées



Solutions

nginx :

- Modèle asynchrone, non-blocking, event-driven (opposé aux pools de process ou de threads d'Apache)
- Supporte énormément de requêtes concurrentes
- Utilisation pour les cluster Kubernetes via Ingress

HAProxy :

- Haute-disponibilité, Répartition de charge
- Supporte d'énormes volume de charge

Zuul Proxy :

- Application SpringBoot

...



Déploiement blue-green

Le déploiement blue-green a pour but que le service soit toujours disponible même pendant le déploiement d'une nouvelle version

Le processus est le suivant :

- La version n est déployée : Toutes les requêtes accédant au proxy sont vers les services exécutant la version n
- La version $n+1$ est déployée et s'exécutent en même temps que la version n . Certaines requêtes sont dirigées vers la nouvelle version pour tester.
 - Tests automatisés d'une pipeline CI
 - Tests manuels via machine interne au réseau
 - Canary testing : Une partie des utilisateurs sont dirigés vers la $n+1$
- La version $n+1$ est considéré comme complètement opérationnelle. Lorsque, la version n a répondu à toutes les requêtes en cours, elle est supprimée de l'environnement de production



Systemes auto-correctifs

Les systèmes auto-correctifs sont des systèmes qui vérifient et optimisent continuellement leur état pour s'adapter aux conditions changeantes.

Ces fonctionnalités peuvent s'appliquer au niveau :

- Applicatif :
Ex : Pattern disjoncteur des micro-services (Hystrix)
- Système :
Healthcheck des services de discovery
- Matériel



Surveillance et logs centralisés

Les architectures *DevOps* étant fortement distribuées. Pour la surveillance, il est nécessaire de centraliser les informations :

- Logs
- Métriques des nœuds
- Paquets réseau

Solutions :

- *ElasticStack, Grafana*



Solutions, Focus sur Kubernetes

Swarm mode

Introduit avec la v1.12

Remplace Docker Swarm

Client Docker

Proxy des requêtes vers *docker-engine*

Pas de GUI par défaut (portainer, rancher, ...)

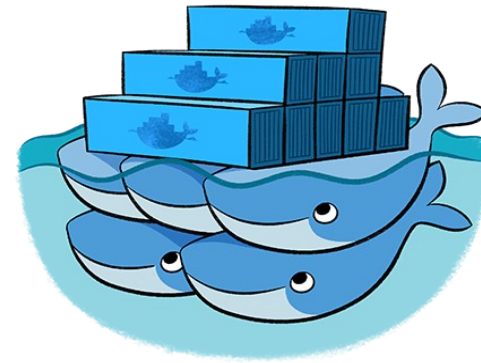
Vocabulaire:

Node: Docker engine membre du cluster swarm de type Manager (Membership, Délégation) ou Worker

Service: Ensemble de tâches (Répliques, Réseau, Stockage)

Tâches: Un conteneur et sa commande géré par un Manager

Pile de service : ~ Docker-compose



Docker Swarm mode

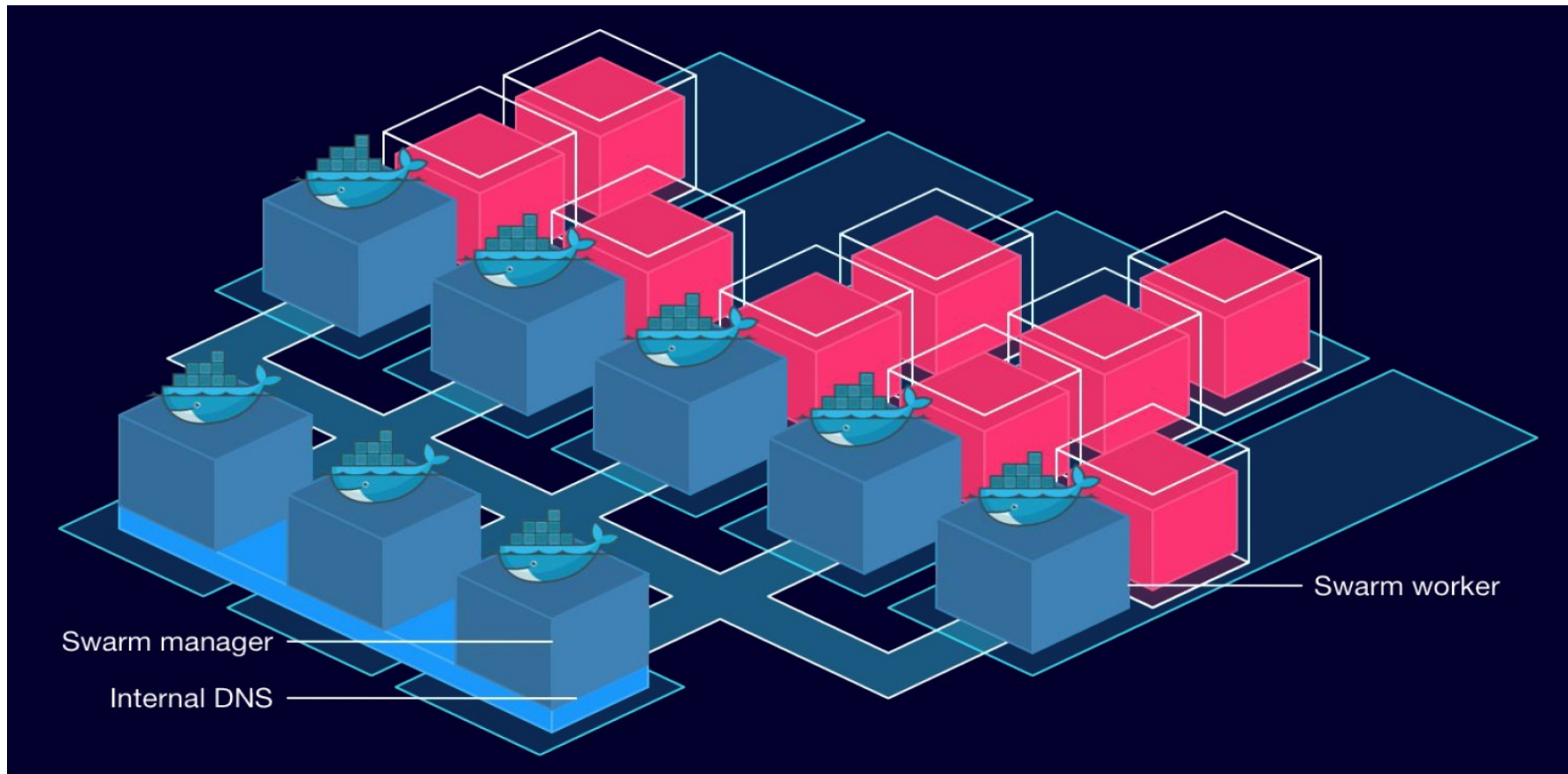
3500 commits

166 contributeurs

Développé en Go

Géré par Docker Inc

Architecture





Docker Machine

Déploiement et administration des hôtes de conteneur

- Provisions de machines virtuelles
- Démarrage, Arrêt, Suppression de machines virtuelles
- Installation / Configuration du moteur Docker
- Administration basique de l'hôte
- Provision de nœud Swarm

Installation et configuration de docker sur plusieurs systèmes

- Une quinzaine de pilotes disponibles : VirtualBox, VMware, Hyper-Azure, AWS, DigitalOcean, GCE, etc



Provisionnement

- Vérification / Installation de l'hyperviseur
Hyper-V, VirtualBox, Azure, ...
- Téléchargement du système d'exploitation hôte
Boot2Docker, RancherOS, Ubuntu, Debian, ...
- Déploiement de la machine virtuelle
Configuration mémoire, CPU, ...
- Création et déploiement d'un jeu de clé SSH
Magasin local et de l'hôte déployé
- Installation du moteur Docker
Choix de la version
- Intégration à un cluster SWARM
Optionnel



Exemples d'utilisation

```
docker-machine create -d azure -azure-subscription-id=<sub-id> -azure-subscription-cert=<cert.pem> node1
```

Déploiement d'une machine dans le cloud

```
docker-machine ls
```

Liste des machines attachés à docker-machine

```
docker-machine ssh node1
```

Connexion à une machine en SSH

```
docker-machine env mycontainer
```

Liste des variables d'environnement d'un conteneur

Kubernetes

OpenSource : Provient des projets Google's Borg and 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 : 71000 commits
+2000 contributeurs

49000* sur GitHub

Géré par la Cloud Native
Computing Foundation
(Groupe neutre)



Fonctionnalités

Connu comme le noyau linux des systèmes distribués

Fournit une abstraction du matériel sous-jacent (les nœuds) et fournit une API afin que des charges de travail soit consommées par un pool de ressources partagé

Agit comme un moteur qui fait converger l'état courant d'un système vers un système désiré

Tous les services gérés sont clusterisés et load-balancé par nature : Permet de scaler dynamiquement



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 de Workloads
- 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

pod

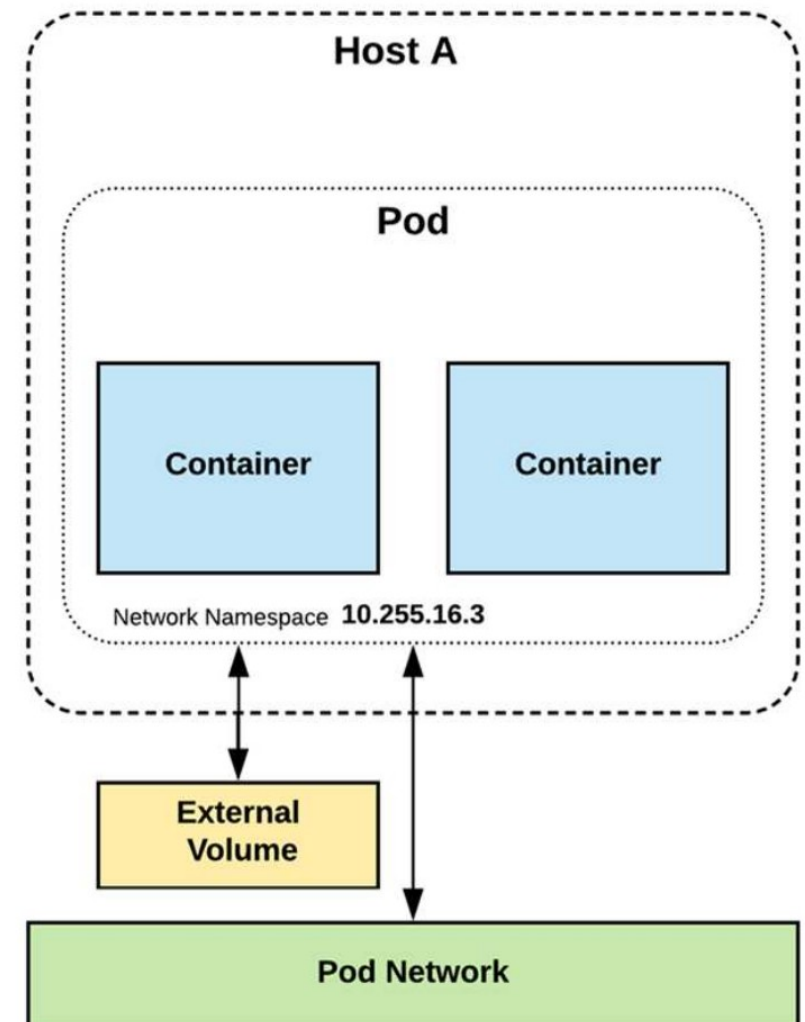
Un **pod** est la plus petite unité de travail de Kubernetes

Un pod regroupe un ou plusieurs conteneurs qui partagent :

- Un espace de nom réseau
- Les mêmes volumes (système de fichier)
- Appartenant à un seul contexte

Les pods sont éphémères. Ils disparaissent lorsqu'ils :

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





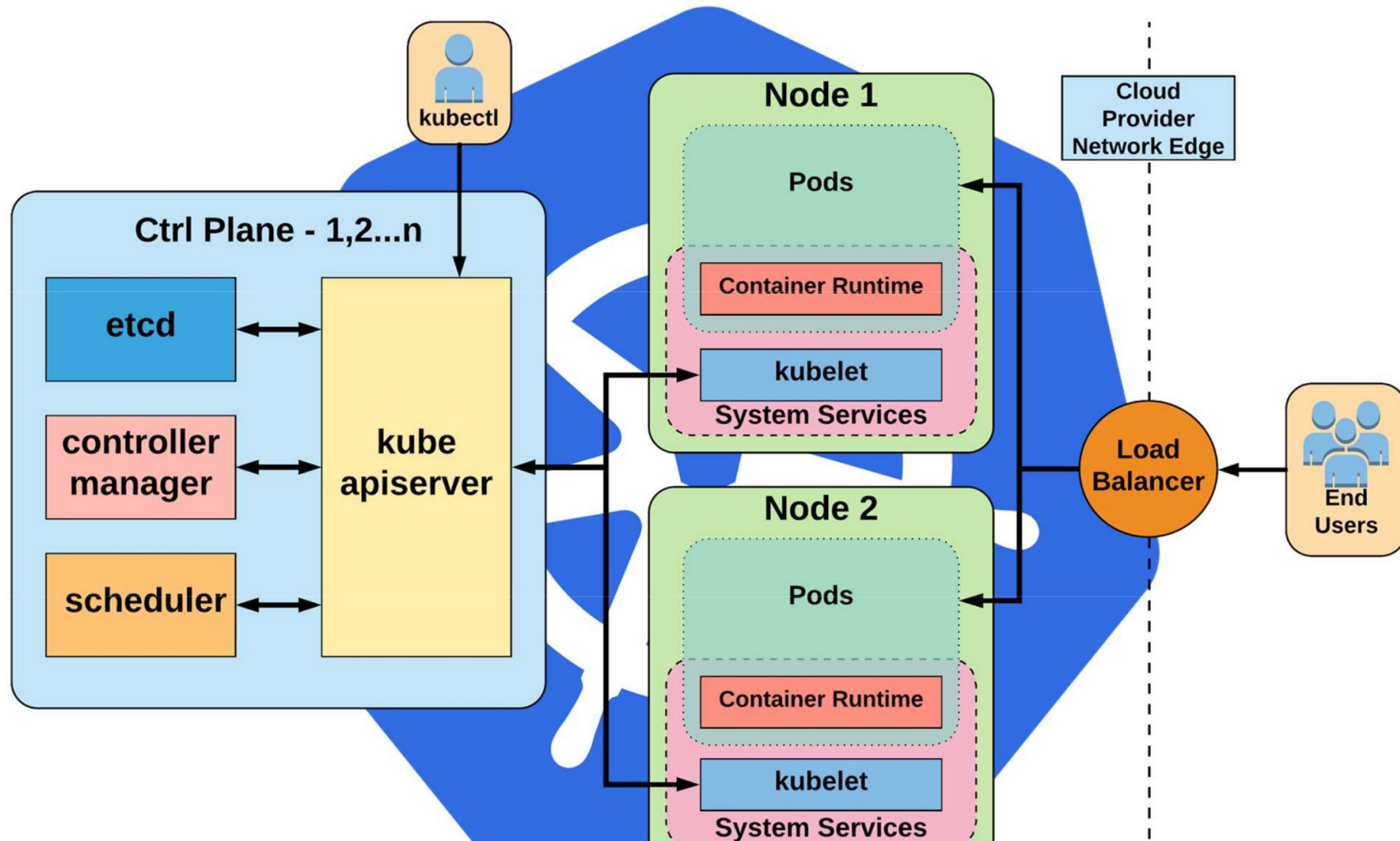
Services

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)

Architecture





Composants de contrôle

etcd : Cluster datastore stockant des clé/valeurs.

controller-manager : Service. Amène le cluster vers son état souhaité

scheduler : en fonction de règles, évalue les contraintes des workloads afin de les placer sur des ressources correspondantes.

api-server : Donne accès aux précédents composants via API Rest. Authentification, autorisation, ...

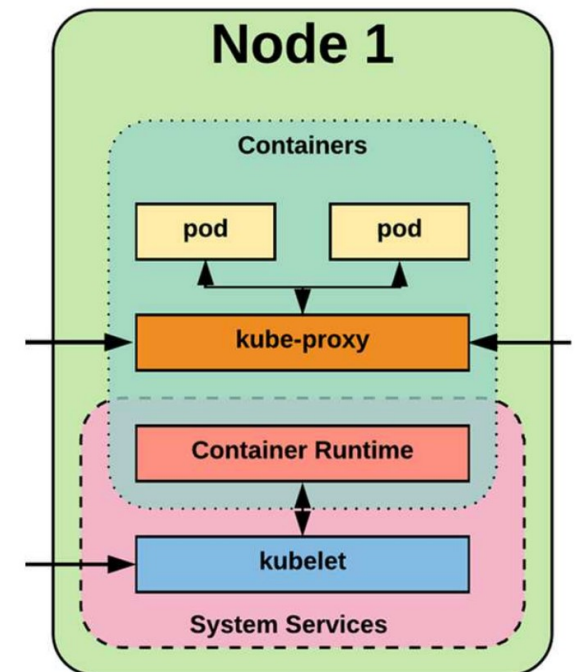
kubectl : CLI client

Composants sur le nœud

kubelet : Agent présent sur chaque nœud gérant le cycle de vie des pods, comprend les manifestes YAML des conteneurs

kube-proxy : Gère les règles réseau sur chaque nœud. Forward et load-balance vers les pods

container-runtime : Exécute les containers (docker, cri-o, Rkt, Kata, Virtlet)





Services optionnels

Cloud-controller-manager : Intégration à un service cloud spécifique

Cluster DNS : DNS pour les services Kubernetes

Kube dashboard : Tableau de bord web de monitoring

API Metrics : Fourniture de metrics



Règles réseau

Tous les conteneurs d'un pod peuvent communiquer entre eux sans entrave.

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.



API

Format:

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

Examples:

`/apis/apps/v1/deployments`

`/apis/batch/v1beta1/cronjobs`



Modèle Objet

- ***apiVersion***: version API de Kubernetes de l'objet
- ***kind***: Type d'objet Kubernetes
- ***metadata.name***: nom unique de l'objet
- ***metadata.namespace***: Espace de nom de l'objet
- ***metadata.uid***: UID (généré) d'un objet.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  namespace: default
  uid: f8798d82-1185-11e8-94ce-080027b3c7a6
```



Workload

Les objets liés à la charge de travail ont deux champs imbriqués supplémentaires.

- ***spec*** : Décrit la configuration d'état souhaitée de l'objet à créer.
- ***status*** : Est géré par Kubernetes et décrit l'état actuel de l'objet et son historique.



Example

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
```



Déploiement

Kubernetes propose l'objet ***deployment*** permettant :

- De créer des pods gérés par un replicaset
- Déclarer un nouvel état des pods
- Revenir à un ancien déploiement
- Scaler
- Mettre en pause
- ...



Autres concepts

namespace : Permet de partitionner le cluster.

labels : Paires clé-valeur utilisées pour décrire et regrouper des ensembles d'objets ou de ressources

sélecteurs : Utilise les labels pour filtrer ou sélectionner des objets.



Quelques commandes *kubectl*

<code>kubectl config view</code>	<code># Affiche les paramètres fusionnés de kubeconfig</code>
<code>kubectl create -f ./my-manifest.yaml</code>	<code># crée une ou plusieurs ressources</code>
<code>kubectl get services</code>	<code># Liste tous les services d'un namespace</code>
<code>kubectl get pods --all-namespaces</code>	<code># Liste tous les pods de tous les namespaces</code>
<code>kubectl describe pods my-pod</code>	<code># Description complète d'un pod</code>
<code>kubectl set image deployment/frontend www=image:v2</code>	<code># Mise à jour</code>
<code>kubectl rollout undo deployment/frontend</code>	<code># Rollback du déploiement précédent</code>
<code>kubectl scale --replicas=3 rs/foo</code>	<code># Scale un replicaset nommé 'foo' à 3</code>
<code>kubectl delete pod,service baz</code>	<code># Supprime les pods et services ayant le noms "baz"</code>
<code>kubectl logs my-pod</code>	<code># Affiche les logs du pod (stdout)</code>
<code>kubectl attach my-pod -i</code>	<code># Attache à un conteneur en cours d'exécution</code>
<code>kubectl exec my-pod -- ls /</code>	<code># Exécute une commande dans un pod existant (un seul conteneur)</code>
<code>kubectl port-forward my-pod 5000:6000</code>	<code># Écoute le port 5000 de la machine locale et</code>
<code>forwarde vers le port 6000 de my-pod</code>	



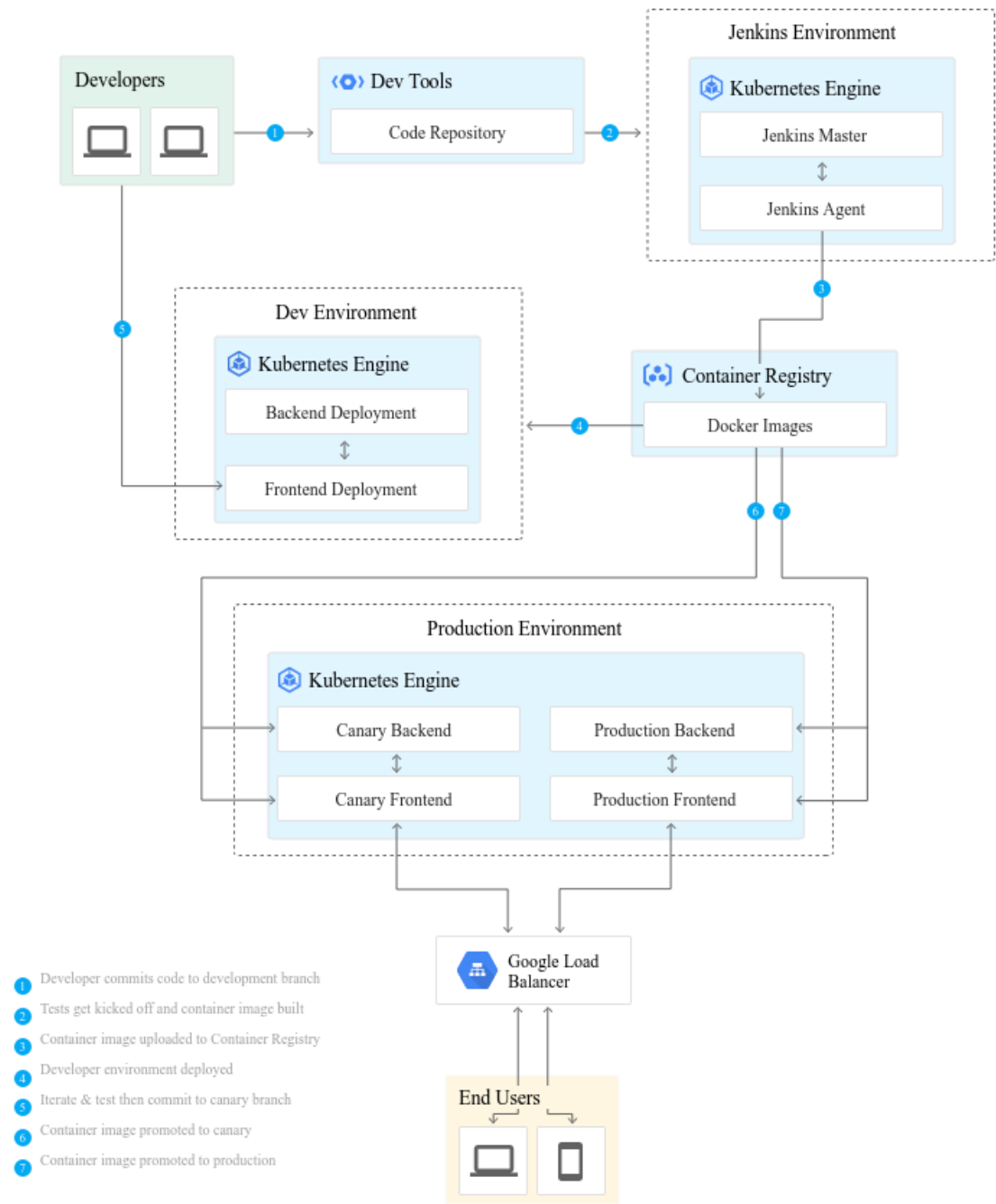
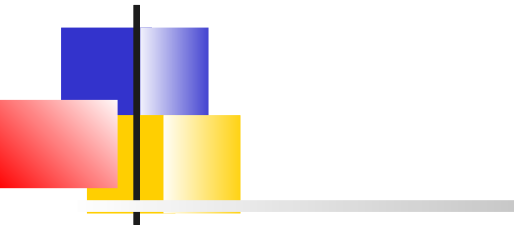
Kubernetes et le cloud

Kubernetes est proposée par les acteurs du cloud

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



Intégration dans la pipeline CI/CD





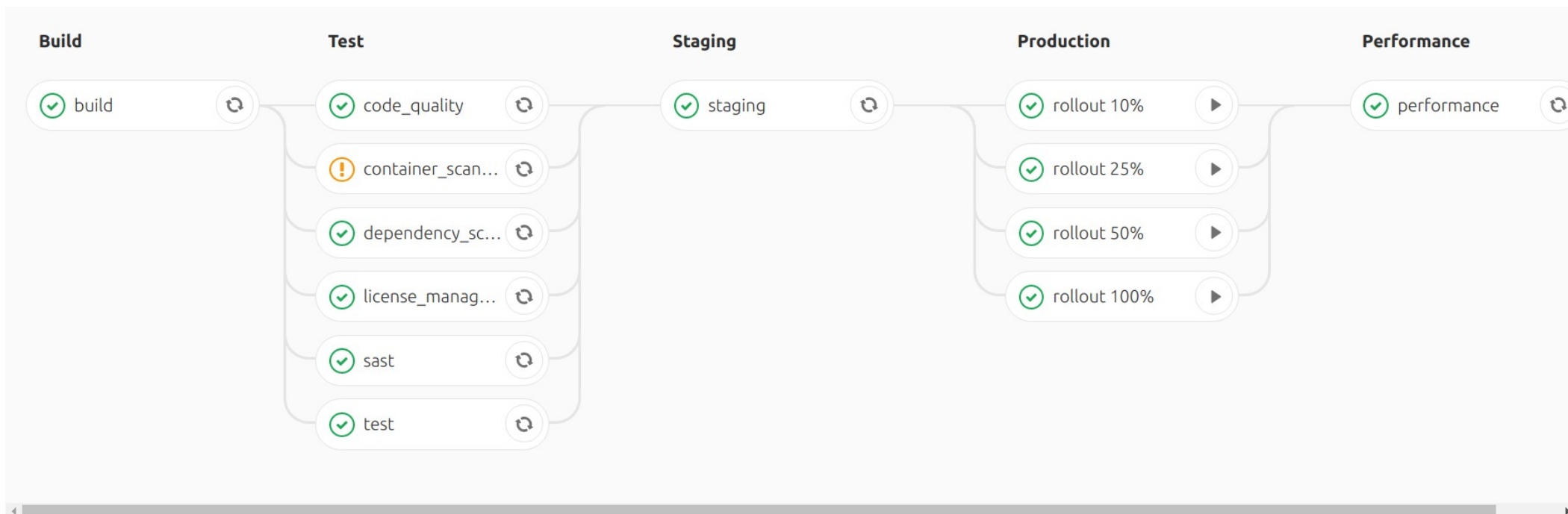
Exemple AutoDevOps GitlabCI

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





Annexes



Chef

Chef est un outil d'automatisation d'infrastructure écrit en Ruby

Il est multi-plateforme et permet entre autres choses :

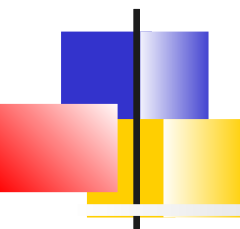
- le déploiement de configurations
- l'analyse et les rapports d'états de déploiement



Composants principaux

Chef se découpe en plusieurs parties :

- ***Chef-Server*** qui est le serveur central d'approvisionnement
- ***Chef-Client*** qui permet d'exécuter des "recettes" sur les nœuds
- ***Chef-Automate*** qui permet l'automatisation de recettes en fonction de "workflow" ou "pipeline"



Concepts

Resource: Le composant basique d'une recipe. Décrit l'état attendu d'un élément de configuration comme un package, un service, un utilisateur, un groupe

Recipe : Ensemble de ressources formant une configuration qui *marche*

Cookbook : Ensemble de *recipes* formant la configuration finale d'une infrastructure.
Uploadé sur le Chef server.

Les cookbooks peuvent également utiliser des Attributs, Fichiers, Gabarits, Méta-données qui peuvent personnaliser les recipes et ressources



ChefDK

ChefDK inclut 2 principales commandes en ligne :

- ***chef***: Permet de gérer un repository *Chef* où se trouvent les recettes, cookbooks, gabarits, etc. Et permet de pousser vers le serveur
- ***knife***: Permettant d'interagir avec les nœuds gérés par le serveur



Puppet

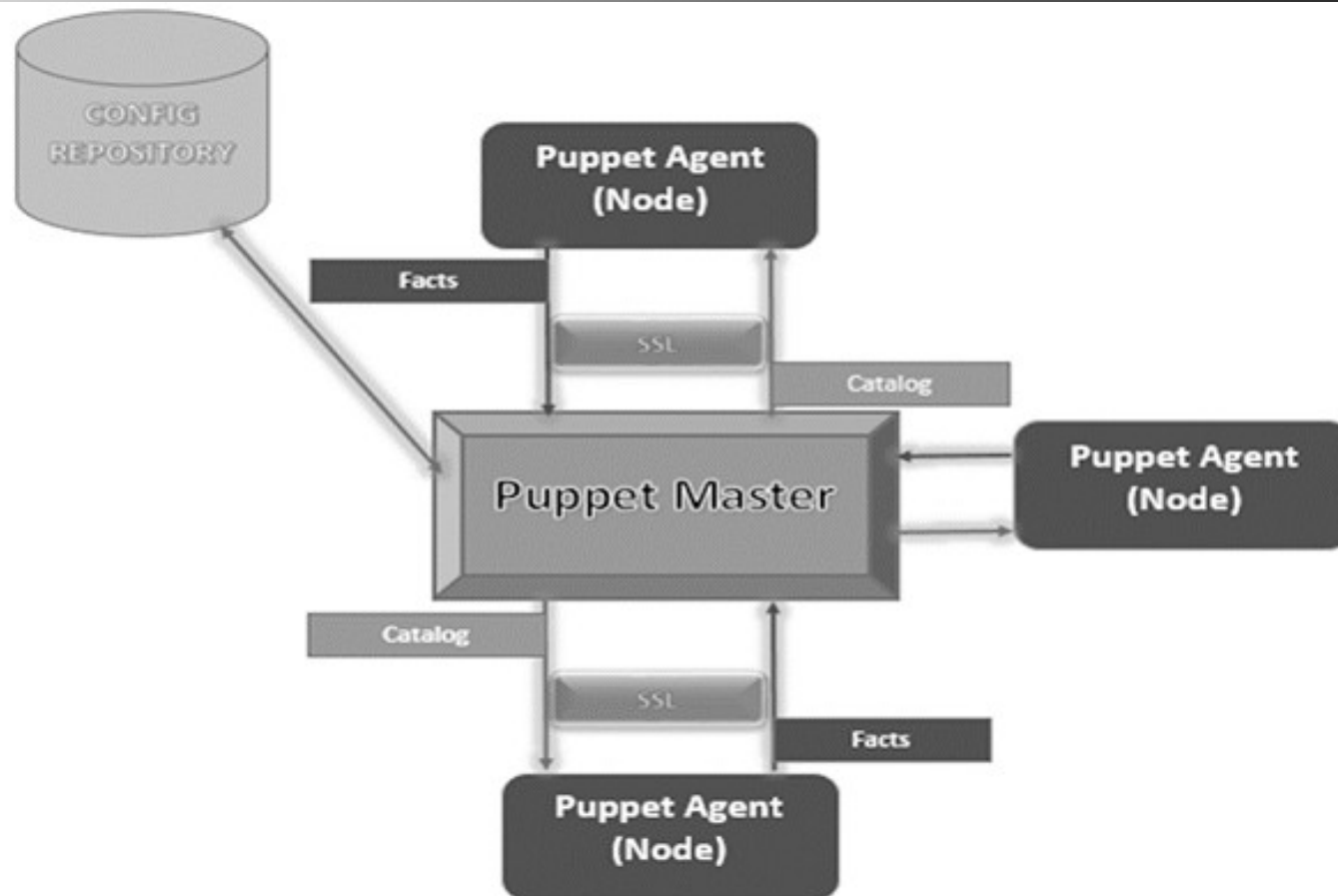
Puppet réalisé en Ruby

Version communautaire est disponible sous licence GPL, version pro.

Puppet utilise un langage déclaratif

- L'administrateur saisit l'état qu'il souhaite obtenir (permissions souhaitées, fichiers et logiciels à installer, configurations à appliquer),
- Puppet se charge automatiquement d'amener le système dans l'état spécifié quel que soit son état de départ.

Architecture





Exemple

```
class serveur_web
{
  package {'httpd' :
    ensure => 'present'
  }
  service { 'httpd':
    ensure => 'running',
    enable => true
  }
  file { '/var/www/myapp':
    ensure => 'directory',
  }
  file { "/etc/httpd/conf.d/myvhost.conf":
    ensure => file,
    owner => 'root',
    group => 'root',
    content => epp('web/myvhost.conf ', {
      server_name => 'myserver.exemple.com',
      www_root => '/var/www/myapp'
    })
  },
  notify => Service['httpd'],
  require => Package["httpd"]
}
```