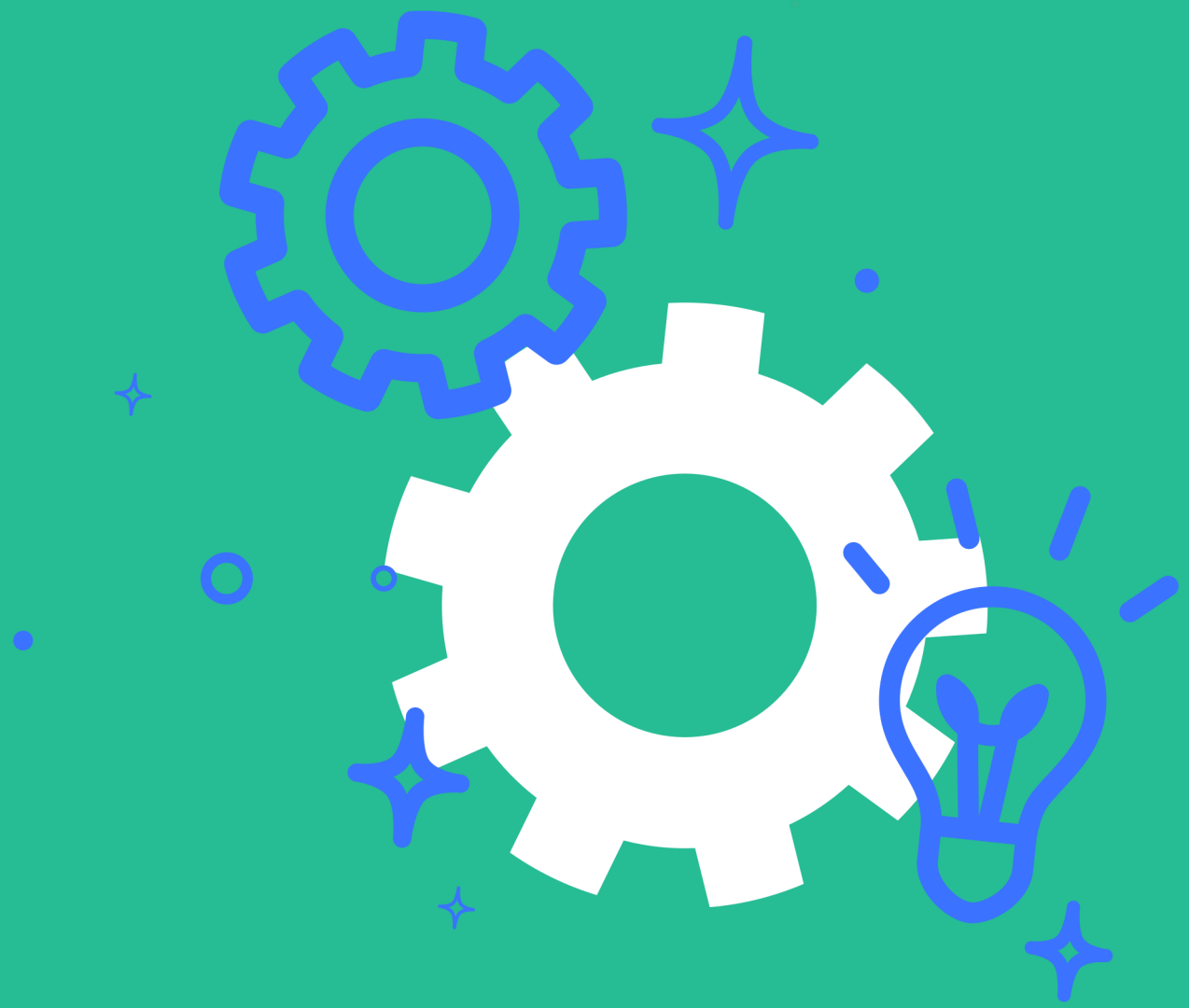
DEVOPS & INDUSTRIALISATION

Les solutions open source



SMILE

I.T IS OPEN

Table des matières

Edition 2017

Préambule

Smile

Smile est une société d'ingénieurs experts dans la mise en œuvre de solutions open source et l'intégration de systèmes appuyés sur l'open source.

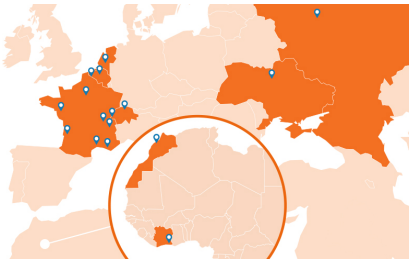
Smile compte 1000 collaborateurs dans le monde, dont plus de 600 en France, ce qui en fait la première société en France et en Europe spécialisée dans l'open source. Depuis 2000, Smile mène une action active de veille technologique qui lui permet de découvrir les produits les plus prometteurs de l'open source, de les qualifier, de les évaluer, puis de les déployer, de manière à proposer à ses clients les produits les plus aboutis, les plus robustes et les plus pérennes. Cette démarche a donné lieu à toute une gamme de livres blancs couvrant différents domaines d'application. La gestion de contenus, les portails, le décisionnel, les frameworks PHP, la virtualisation, la Gestion Electronique de Documents, les ERP, le big data…

Chacun de ces ouvrages présente une sélection des meilleures solutions open source dans le domaine considéré, leurs qualités respectives, ainsi que des retours d'expérience opérationnels.

Au fur et à mesure que des solutions open source solides gagnent de nouveaux domaines, Smile est et sera présent pour proposer à ses clients d'en bénéficier sans risque. Smile

apparaît dans le paysage informatique français et européen comme le prestataire intégrateur de choix pour accompagner les plus grandes entreprises dans l'adoption des meilleures solutions open source. Ces dernières années, Smile a également étendu la gamme des services proposés. Depuis 2005, un département consulting accompagne nos clients, tant dans les phases d'avant-projet, en recherche de solutions, qu'en accompagnement de projet. Depuis 2000, Smile dispose d'une Agence Interactive, proposant outre la création graphique, une expertise e-marketing, éditoriale, et interfaces riches. Smile dispose aussi d'une agence spécialisée dans la Tierce Maintenance Applicative, l'infogérance et l'exploitation des applications.

Enfin, Smile est implanté à Paris, Lyon, Nantes, Bordeaux, Lille, Marseille, Montpellier et Toulouse. Et présent également en Suisse, en Ukraine, aux Pays-Bas, en Belgique, au Maroc ainsi qu'en Côte d'Ivoire.



En parallèle à ces publications, pour bien comprendre la révolution en marche de l'open source, Smile a publié plusieurs livres blancs expliquant les spécificités, les modèles économiques, les sous-jacents ainsi que les atouts de l'open source.

Ce livre blanc

Avec les exigences de planning "Time-to-market" et le développement des méthodologies agiles, les besoins d'industrialiser les tests, de fluidifier les déploiements en production et de rapprocher les équipes IT se sont progressivement affirmés.

Les projets évoluant dans des configurations logicielles et d'infrastructure de plus en plus complexes, générant des risques opérationnels et de planning, imposent alors d'aller jusqu'à ***l'automatisation des déploiements*** .

Les solutions permettant de répondre à ces enjeux sont portées par la ***dynamique open source*** , apportant un foisonnement d'outils innovants, divers mais ***interopérables*** , et surtout ***accessibles*** à toutes les entreprises.

Si les démarches agiles veulent rapprocher les métiers des équipes de développement, ***le DevOps a vocation à rapprocher les fonctions de développement (Dev) et d'exploitation (Ops)*** de l'IT. Ces démarches, et l'alignement entre les différents acteurs qu'elles entraînent, sont des inducteurs de travail collaboratif, d'efficacité, de qualité et de mise à disposition au plus vite de produit ou service IT.

Les concepts et solutions décrits dans ce livre blanc éclairent sur les pistes pour avancer dans une démarche DevOps, portée vers l'excellence opérationnelle et le renfort de collaboration entre les équipes.

Au travers de notre expérience en intégration de solutions de bout en bout dont l'infogérance, et notre veille permanente sur les solutions Open Source, nous avons souhaité partager avec vous ces concepts et solutions pour faciliter la réussite de vos projets et opérations IT.

Auteurs

Ce livre blanc a été rédigé par Patrice Ferlet, avec la contribution de Badr Chentouf, Florent Béranger et Hubert Fongarnand.

NB : Cet ouvrage a été rédigé en LaTeX [1] sur l'interface ShareLaTeX [2], deux logiciels open source.

1. http://www.latex-project.org/
2. https://fr.sharelatex.com/

Principes

Les notions de base

Les termes ***Continuous Delivery*** (CD) et ***Continuous Integration*** (CI), respectivement en français Déploiement continu et Intégration continue sont très présents depuis quelques années. Dans la méthodologie ***AGILE*** , ces deux concepts s'intègrent de manière quasi-naturelle et bien souvent les équipes construisent leurs logiciels en suivant ces principes sans en avoir pleinement conscience.

Le but est de permettre l'évolution de projets de manière fluide entre la production de code, et le déploiement final.

La majeure partie des projets est généralement soumise à la vue de différentes équipes :

- les développeurs, techniciens intégrateurs de solutions
- les équipes fonctionnelles qui délivrent des spécifications d'intégration et qui, souvent, sont aussi testeurs de nouvelles fonctionnalités
- les opérateurs, administrateurs système, qui vont mettre en production les versions finales

Ces 3 équipes sont disséminées dans l'entreprise, parfois géographiquement, et il n'est pas rare que la communication entre elles ne soit ni claire ni efficace. L'une des principales raisons de cette fracture est l'absence d'outils pour leur permettre de communiquer efficacement.

Le ***DevOps*** est un mouvement qui est de plus en plus adopté par les équipes techniques dans les entreprises éditrices de logiciels ou les intégrateurs de solutions (Web, mobile, vidéo, etc...). Ce principe propose de mettre en place une jonction entre le développeur et l'opérateur (ainsi que, si possible, les intermédiaires) pour réaliser efficacement et de manière maîtrisée le produit final.

Pour résumer, nous pouvons définir 3 groupes d'outils qui s'intègrent dans le processus de Continuous Delivery :

- **Le SCM** ou "Source Control Management" pour "Gestion de contrôle de Sources" qui permet de rassembler le code source, les branches et les versions
- **la PIC** pour "Plate-forme d'intégration continue" qui va interfacer le VCS 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)

Tous ces groupes peuvent être un ensemble d'outils. Et c'est sur ce point que peut aussi régner la confusion. Bien que ces trois groupes soient clairement identifiés, les outils qui les portent sont parfois capables de gérer plusieurs couches du processus.

Prenons par exemple ***Jenkins*** qui est un formidable ***outil d'intégration continue*** , il est aussi très souvent utilisé comme outil de déploiement et peut très bien faire appel à ***Ansible*** ou ***Puppet*** pour provisionner les serveurs de production, après avoir créé l'objet de livraison. Dans ce cas, Jenkins va jouer sur plusieurs tableaux et prendre en charge plusieurs aspects du CI/CD.

C'est pour cela qu'il faut avant tout ***bien identifier quel outil fourni quel service*** , et avoir connaissance de la frontière jusqu'au concept suivant.

> La ***PIC*** va être l'élément central dans un flux de livraison continue puisqu'elle est l'élément qui va ***déclencher une procédure de déploiement*** automatisée, un refus de fusion et alerter les équipes.

Intégration Continue

L'Intégration Continue est en quelque sorte le socle du Déploiement Continu. Pour résumer le principe, chaque modification poussée par un développeur dans le service de gestion de sources va être automatiquement testés pour s'intégrer dans le tronc de développement [1] - ce qui induit qu'à tout moment l'application est considérée comme étant potentiellement livrable.

1. ou bien dans la branche de maintenance adéquate

Ce qui fait office de limite par rapport au Déploiement Continu est que ce tronc n'est pas mis en production sans l'aval d'une opération des équipes opérationnelles. Les raisons qui poussent à ne pas déployer ces versions automatiquement sont nombreuses, notons par exemple les différences d'environnement entre le poste de développeur, la plate-forme d'intégration continue et l'environnement de production. Mais pour autant, l'intégration continue est bel et bien l'une des briques les plus importantes à mettre en place pour réussir à adopter le Déploiement Continu.

La mise en place d'une plate-forme d'intégration continue (PIC) est indispensable à la mise en œuvre d'une démarche de "Continuous Delivery", la PIC permettant alors d'assurer la qualité logicielle en continu. En effet, une PIC permet de s'assurer qu'à chaque itération de création de valeur logicielle, un ensemble socle de tests, qui peut être de différentes natures, est effectué.

Les résultats des tests de la PIC pouvant alors déclencher ou non le déploiement sur un environnement de production, il est clair que la qualité et la complétude des tests est cruciale. Cette complétude, et le coût en temps associé sont à prendre en compte avant d'opter pour une méthode de livraisons en production vraiment automatisées.

Le sujet de l'intégration continue, ou "Continuous Integration", est très bien couvert par les solutions open source comme Jenkins, sans oublier Git ou SVN pour la gestion des sources. Les outils d'intégration continue sont décrits dans notre livre blanc "***Gestion du Cycle de vie des Applications (ALM)*** " que nous vous invitons à télécharger et lire.

Livraison Continue

La démarche DevOps se base sur le fait que le développement logiciel et son aspect production, opérationnel, sont fortement imbriqués, et qu'ils doivent par conséquent être gérés ensemble, et ce, ***dès le début d'un projet*** .

Cette orientation nous vient des leaders du web - Google, Amazon, ... qui ont un besoin extrêmement fort de livrer en continu des améliorations de leurs services sur internet.

> Issue du ***DevOps*** , le ***Continuous Delivery*** est une orientation du développement logiciel consistant à livrer régulièrement et de manière automatisée

Bien sûr, ce n'est pas si récent que de vouloir livrer ses développements de manière itérative. Dans le cadre d'une méthodologie agile, il faut par exemple livrer de manière régulière, après chaque sprint.

Mais d'une part les besoins actuels de ***time-to-market*** [2] imposent un nouveau rythme, bien supérieur, nécessitant industrialisation et automatisation des processus. Aussi Google, Facebook, Amazon, Twitter et autres, mettent en production plusieurs fois par semaine, ou même plusieurs fois par jour !

D'autre part, par rapport à une méthodologie agile classique, il ne s'agit pas uniquement de livraisons pour test, mais bien de livraison en production, données comprises, sans impact, régression ou incident !

Dans ce contexte, on ne peut pas imaginer rester avec des mises en production traditionnelles, nécessitant une intervention manuelle et une supervision humaine dédiée pendant et après la mise en production, avec aussi la recommandation de ne pas faire de mise en production le vendredi ... Au contraire, avec les outils de Continuous Delivery, ***chaque livraison en production doit devenir un non-évènement*** , ne nécessitant pas de surveillance particulière.

Pour simplifier le concept, il suffit d'imaginer que chaque niveau opérationnel de l'application est testé de manière automatique et/ou manuelle. En règle générale, les tests automatisés (tests unitaires, tests end-to-end) sont en première ligne. Lorsque l'acceptation de tests est validé à ce niveau, l'application passe aux mains de tests utilisateurs. Quand l'utilisateur valide ces tests, une "release" est effectuée.

2. facteur délais nécessaire pour le développement et la mise au point d'un produit avant sa mise en place sur le marché

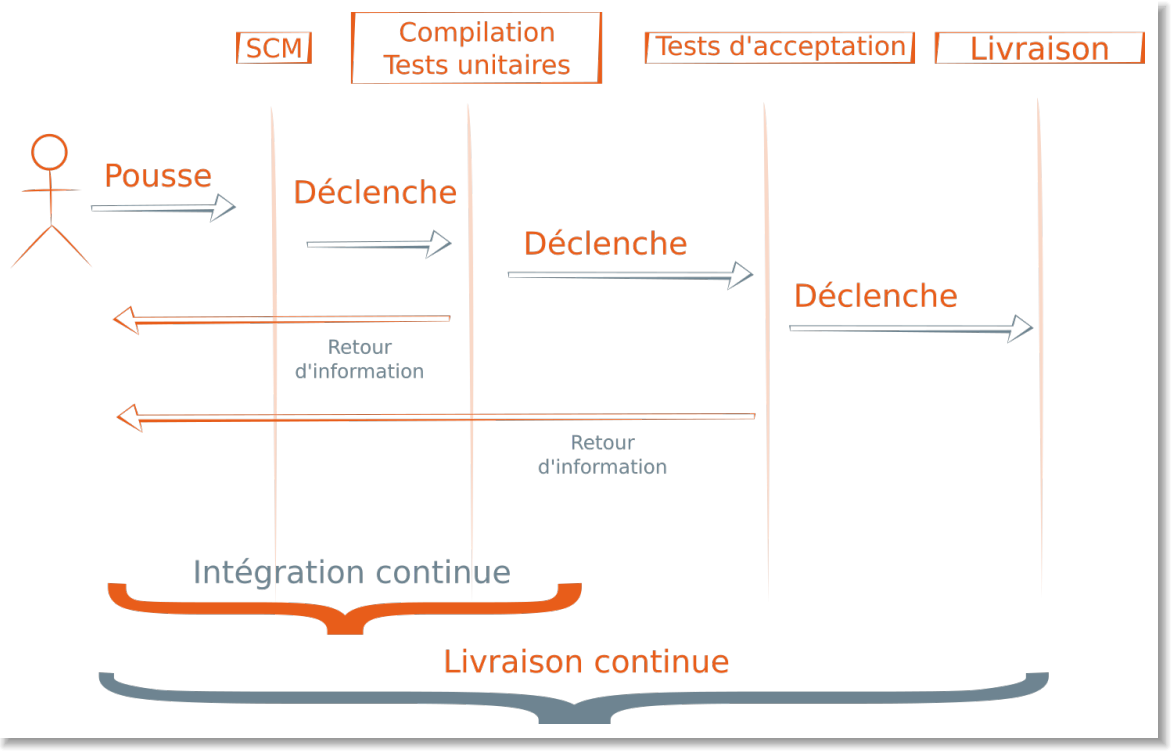


Figure 2.1 – *Séquence de livraison continue*

Il n'y a pas de règles strictes ni de services clairement séparés, dans le diagramme 2.2 les blocs "Compilation et Tests unitaires" et "Tests d'acceptation" qui peuvent éventuellement être intégrés par le même service (par exemple Jenkins est capable de prendre en charge ces opérations, en utilisant des outils d' approvisionnement tels que Ansible, Puppet ou simplement avec un script Shell).

Le principe reste tout de même évident : le logiciel est testé, construits et déployé de manière fluide.

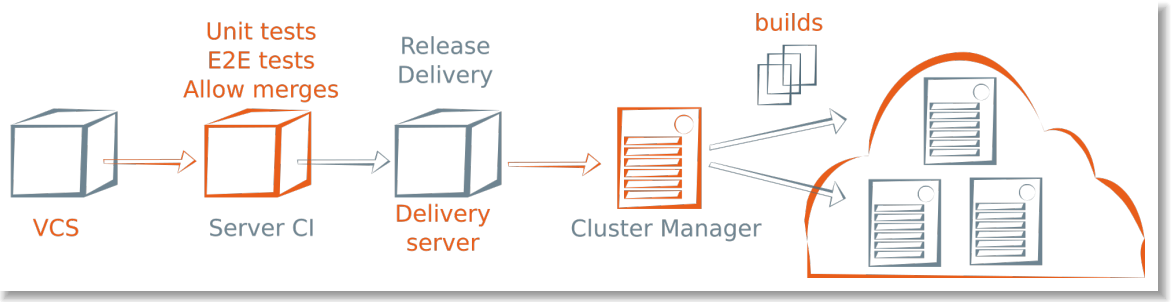


Figure 2.2 – *Représentation d'un flux CI/CD*

Dans le diagramme 2.2, il est bon de noter que les blocs en forme de cube sont les éléments intrinsèques du CI/CD. Le gestionnaire de cluster n'est pas, à proprement parler, un bloc de CI/CD mais il est généralement tout aussi important et doit être pris en compte. C'est pour cela que nous devons, nous aussi, prendre en considération cet élément dans notre ouvrage.

La crainte, tout à fait légitime, est de délivrer une version en production en ne faisant confiance qu'au processus de "Livraison en continu".

Faut-il faire confiance aux tests implémentés ? est-ce que ces tests suffisent ?

Pour faire un parallèle, on peut se remémorer ce qui était en vigueur pour la validation de contenu dans les outils de gestion de contenus il y a encore quelques années. Le processus était relativement lourd et n'induisait pas de souplesse. Sans impacter l'importance des phases d'édition, de validation et de mise en ligne, la latence entre ses actions engendrait une forte réduction de la dynamique de contenu.

Aujourd'hui, la philosophie a largement évolué, notamment grâce à des outils de vérification, l'utilisation de formats d'édition légers (laissant à charge la mise en forme automatisée), les anti-spam (pour les commentaires), la classification automatique, etc. Finalement, les phases du workflow existent encore mais sont fortement assistées par les outils qui permettent de soulager ces processus.

> Dans un sens, le "Continuous Delivery" s'inscrit dans cette tendance visant à automatiser les contrôles, réduire les contraintes et donc à fluidifier et maîtriser le processus d'intégration. À charge de l'ensemble de l'équipe, développeurs et agents de production, de faire en sorte, ensemble, que le processus soit viable et optimal.

La livraison continue apporte également d'autres natures de gain au processus de livraison, comme la reproductibilité des opérations de déploiement.

Les gestionnaires de versions

La base d'un flux de livraison continue repose en général sur ce type de service. Le gestionnaire de version, connue par son acronyme anglais *"VCS"* (Version Control Service) ou SCM (Source Control Management) fournit la possibilité à une équipe de développeurs

d'injecter du code et de le corriger sans pour autant perdre les corrections des autres développeurs de l'équipe. Il permet aussi l'accès à l'historique, la gestion de conflits, et majoritairement de créer des branches et des étiquettes (tag) permettant d'identifier des livraisons.

Les branches et tags sont essentiels.

Lorsqu'il est nécessaire de développer une version qui propose une nouvelle fonctionnalité ou encore pour une version de maintenance (corrective), il faut impérativement être capable de ne pas impacter le "tronc" qui représente le développement de la future version.

Ainsi, il est possible de créer une branche à partir d'une version précise, de développer et créer des étiquettes (tags) qui font office de repère de version, et livrer ces derniers sans rapatrier les développements en cours non validés. En bref, un VCS est une nécessité pour appréhender correctement le Continuous Delivery.

Aujourd'hui, la quasi-totalité des équipes de développement en entreprise (ESN, éditeurs, équipes internes...) utilisent un gestionnaire de concurrence de versions. Que ce soit l'ancestral ***CVS*** ou encore ***Subversion*** , sans compter l'incontournable ***Git*** . Cela veut dire que si une équipe utilise un de ces outils, elle a déjà un pied dans le CI et le CD.

L'étape qui suivra l'utilisation d'un VCS est la mise en place d'une interface à une plateforme d'intégration continue (PIC).

Les conteneurs et les machines virtuelles

Livrer des applications est un sujet à part entière, mais l'environnement de développement est un thème primordial.

Prenons un projet web développé en PHP. Traditionnellement le déploiement de ce type d'application était réalisé sur des serveurs physiques dédiés sur lesquels tournent un service HTTP. Que ce soit Apache httpd, Apache Tomcat, ou encore Nginx, les sources du site étaient déposées sur le serveur, dans un répertoire précis. Le moteur de base de données était bien souvent installé sur le même serveur, parfois sur une autre machine. Le coût d'exploitation et la complexité d'infrastructure était une

> Dans l'univers DevOps et Continuous Delivery il faut faire en sorte que la couche entre production et déploiement s'affine.

contrainte pour le développeur qui devait gérer différentes configurations applicatives, et pour les exploitants qui devaient s'assurer de la bonne gestion du système et du réseau.

Les hyperviseurs sont essentiels. Une machine nue [3] est aujourd'hui capable de superviser des dizaines de machines virtuelles, isolant ainsi les déploiements sur des environnements dédiés. Il existe d'excellents hyperviseurs en open source comme KVM ou Xen, qui une fois orchestrés par OpenStack ou Mesos sont capable de rivaliser avec des VMWare, Azure, Amazon...

Bien que le serveur soit virtualisé, le principe reste équivalent à celui d'un serveur dédié. L'autre revers est que pour virtualiser un environnement complet (un système d'exploitation installé et viable), le serveur doit être capable de supporter la charge de son OS hôte ainsi que les OS invités, sans compter l'émulation du matériel (CPU, mémoire, carte vidéo...) et la couche réseau associée. En d'autres termes, les ressources nécessaires pour exécuter une Machine Virtuelle sont importantes.

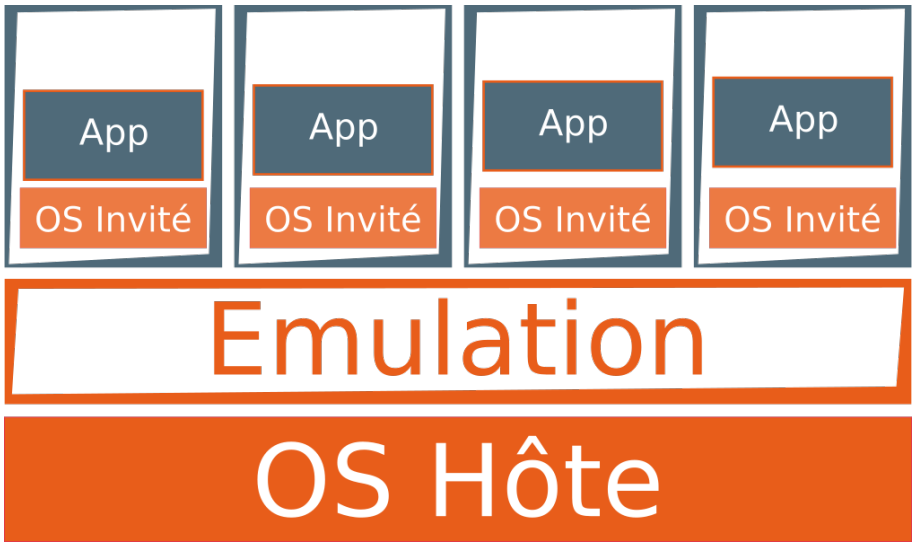


Figure 2.3 – *Représentation du principe de virtualisation "hôte/invité"*

Du point de vue développeur, son poste de développement est largement capable de supporter une machine virtuelle pour travailler avec un environnement proche de celui de production.

C'est sans compter sur l'émergence des ***conteneurs applicatifs*** qui a relativement modifié le visage des principes de déploiement et de services.

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.

3. souvent appelée *Bare Metal* ou *Bare Machine*

Avant d'aller plus loin sur la conteneurisation d'application, il faut rappeler que la technique d'isolation de processus dans un environnement fermé existe depuis des années : le "chroot" [4] permettait déjà de réduire l'impact d'une installation de service dans un répertoire dédié et de plus ou moins l'isoler du reste de l'OS hôte.

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 [5] 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.

Dans la suite de ce livre, nous parlerons de "conteneurs" sans spécifier "applicatifs". De fait, une solution de conteneur telle que OpenVZ, nspawn ou encore LXC peut tout à fait faire office de conteneur applicatif. Il est en effet possible de ne pas démarrer un OS via un "init" et donc de reproduire le comportement que nous connaissons avec Docker ou Rkt. Cependant, dans une approche "micro-service", nous serons plutôt voués à parler de conteneurs applicatifs, et plus particulièrement de Docker.

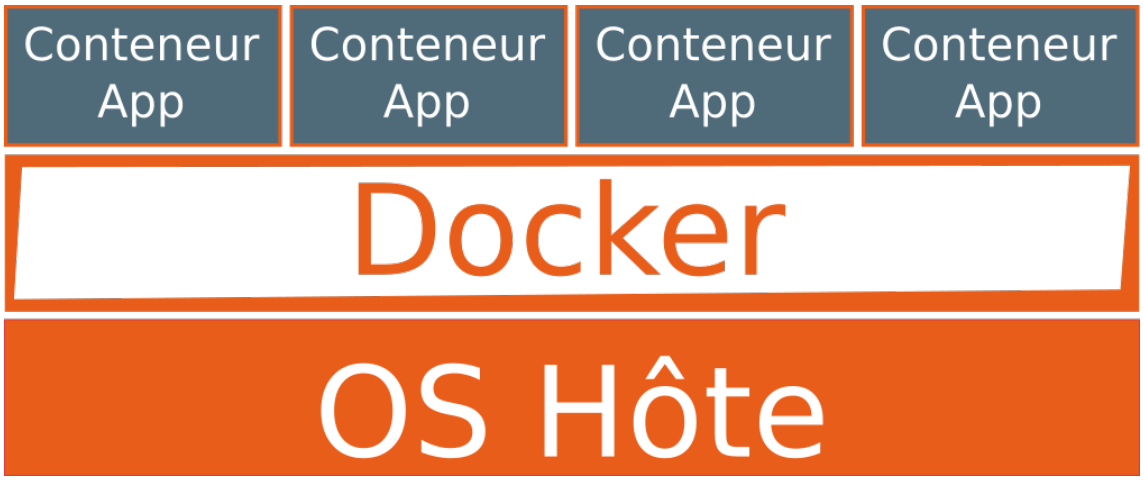


Figure 2.4 – *Réprésentation d'application conteneurisées via Docker*

Le revers de la médaille est que la garantie que le développement fonctionne sur l'environnement de production repose sur le fait que les serveurs de production soient en mesure de faire tourner des conteneurs. Si le système n'utilise pas de gestion de conteneurs, l'application sera tout de même capable de tourner et de répondre, mais le paramétrage s'éloigne de celui utilisé par les développeurs.

Certaines solutions de conteneur permettent un démarrage "quasi-complet", par exemple ***LXC*** va effectuer une séquence de boot et permet de faire tourner l'ensemble

4. qui signifie "change root", ou "changement de racine" en français
5. l'interface réseau étant généralement simulée via un pont

des services nécessaires à une application (telle qu'une base de données et le serveur HTTP) - mais d'autres, comme ***Docker*** demandent à ne faire tourner qu'un seul service. Bien que la mauvaise pratique d'utiliser un processus englobant les services tels que "supervisor" est souvent rencontrée, il est fortement recommandé d'utiliser ***un conteneur par service*** et de ***lier*** les conteneurs entre eux. Cette opération étant simplifiée par ***Docker-compose*** par exemple mais aussi dans les solutions de clusterisation tels que ***Kubernetes*** .

Nous approchons donc d'un mode de fonctionnement applicatif dont le terme est aujourd'hui bien connu :***le micro-service*** .

Notons bien l'un des intérêts qui fait souvent office de fer de lance : ***les conteneurs peuvent être exactement identiques à ceux utilisés en production*** . Moyennant, bien entendu, un travail en amont de la part des développeurs et des agents de production. ***Nous sommes donc bel et bien dans le monde du DevOps*** , une fois de plus.

> Utiliser des conteneurs impose ***une réflexion en profondeur de plusieurs aspects*** tels que la solution de conteneur, la gestion de micro-services sur les postes de développement, la solution utilisée en production et les outils de déploiement

Le principe de micro-service permet aussi de soulager les environnements car il n'est plus nécessaire de démarrer un OS complet (voir figures 2.3 et 2.4). Cela simplifie grandement le paramétrage de l'environnement de travail et sa réutilisation, nous sommes donc aussi dans une logique d'***industrialisation*** . Aussi, il sera aisé de tester plusieurs versions de services puisque ce dernier est un simple conteneur éphémère.

Docker, nativement supporté par Linux, est en passe de permettre aussi son exécution sur OSX. Windows est en reste à cette heure, mais Microsoft défend sa solution Azure et intègre déjà des solutions pour réussir à faire tourner Docker sur sa plateforme.

Sur un environnement de production, nous pouvons alors imaginer plusieurs manières d'exploiter une application :

- les applications sont déployées sur une VM proche de celle des développeurs
- les conteneurs sont déployés sur des environnements préparés
- un gestionnaire de conteneur tels que Kubernetes peut aussi être utilisé.

Et pour nous aider, nous pouvons compter sur OpenShift, OpenStack, Mesos, et tout un panel d'autres solutions qui ont émergé ces dernières années.

Pour faire simple : la production et le développement informatique évoluent dans le même sens - c'est la clef du paradigme de DevOps.

Du développement à la production

Le Continuous Delivery suppose la création d'environnements "iso" à partir du développement, jusqu'à la production, et aussi en sens inverse, en intégrant les contraintes actualisées de la production jusqu'au développement. Pour cela, les outils open source fleurissent et se complètent : Docker, Jenkins, Ansible, Puppet, Chef,...

Si l'environnement de déploiement est clairement la cible, il est difficile pour un environnement de développeur (poste de travail, EDI, environnement système...) d'arriver à reproduire un environnement de production ou s'y approchant suffisamment. Car la grande crainte est de développer un logiciel, quel que soit le type, qui n'arrive pas à s'intégrer sur les plateformes de production.

C'est donc dans ce domaine que les machines virtuelles et les conteneurs ont réussi à changer la donne.

Historiquement, ce sont les machines virtuelles qui ont véritablement donné un nouveau souffle aux développeurs. Car ces dernières proposent de faire fonctionner une machine proche, voire identique, à la machine de production.

Puis les conteneurs ont ajouté une pierre à l'édifice. Ces derniers conquièrent doucement le cœur des administrateurs systèmes et des développeurs pour différentes raisons. Isolation, légèreté, simplicité à intégrer, à *scaler*, etc. Depuis 2013, c'est un sujet qui ne cesse de prendre de l'ampleur de par la forte progression d'intégration de la solution Docker qui a largement contribué à l'adoption de conteneurs. Depuis, d'autres ont emboîté le pas.

Docker permet de créer des conteneurs qui peuvent embarquer les logiciels souhaités. Une des révolutions apportées par Docker est qu'il permet de créer très rapidement un micro-service isolé du système hôte sans utiliser de virtualisation à proprement parler. Cette révolution n'est pas une invention de Docker, ce n'est que la résultante d'une longue liste de techniques déjà éprouvées telles que les *chroot* ou LXC.

Pour autant, il n'y a pas que Docker, et d'autres solutions similaires existent telles que ***RkT (par CoreOS)*** ou ***Systemd Nspawn*** , sans compter sur la venue de ***LXD*** , lui-même proche de ***LXC*** déjà fortement utilisé.

Mais la virtualisation n'a pas disparu pour autant, car elle apporte d'autres aspects plus élémentaires, notamment en termes de gestion de services ou en terme d'isolation. On citera ***Vagrant*** qui permet le démarrage de machines virtuelles (VirtualBox, VMWare, ...) en les initialisant selon une configuration voulue.

Les conteneurs et leur légèreté comblent les développeurs qui peuvent démarrer les services, tester différentes versions de framework ou de logiciels tels que les serveurs de bases de données, en quelques secondes. Les machines virtuelles plaisent aussi par leur approche "packagée" qui permet un contrôle plus rigoureux du fonctionnement du système.

Après le passage en tests, les sources sont donc versionnées et livrées par la PIC. Cette dernière effectue des passes de tests, contrôle la qualité, rapporte et alerte le personnel compétent. Elle va ensuite permettre une livraison automatisée vers un environnement de production.

La PIC doit être en mesure de tester l'application dans les mêmes conditions que les développeurs et que la production. Machine virtuelle ou conteneur, voilà une fois de plus un intérêt majeur de ces technologies. Car une fois de plus, si un problème survient, les équipes de développement vont pouvoir reproduire l'erreur rapidement sans avoir à étudier les différences entre l'environnement de tests et le leur.

La production est aussi quasi-assurée que le logiciel fonctionne sur deux environnements différents sans problème (environnement de développement et PIC).

La production va permettre l'exécution du projet dans les mêmes conditions que celles utilisées par les développeurs de l'application. Si des conteneurs sont utilisés, alors l'image va être identique et, à quelques paramètres près, l'application se retrouve exactement dans le même état d'isolation, de fichier et d'environnement que sur les postes de développement.

Et dans le cas d'une utilisation de machine virtuelle bien contrôlée, une fois de plus, l'application va démarrer et servir sa solution dans une condition testée et approuvée.

Pour ces machines virtuelles, les hyperviseurs sont des sujets essentiels à étudier en détail. L'open source offre aujourd'hui des solutions fiables, éprouvées et de très haute qualité. Notons par exemple ***OpenStack*** et ***Mesos*** (et les frameworks associés), combinés à ***KVM*** en hyperviseur, qui permettent une gestion de machines virtuelles dans des conditions optimales. Que ce soit un IaaS [6] ou un PaaS [7], l'intégration sera contrôlée et rapidement récupérée en cas de panne ou d'erreur.

Concernant la gestion des conteneurs, elle peut devenir subtile dès lors que plusieurs conteneurs doivent fonctionner ensemble, tout en garantissant l'intégrité du fonctionnement en production lors des déploiements. Les développeurs utilisent régulièrement des outils comme ***Docker Compose*** pour lier les conteneurs facilement. La production va utiliser ***Kubernetes*** ou encore ***Panamax*** qui sont des solutions avec une gestion de conteneurs plus élaborée et un niveau d'abstraction plus haut. Il faut donc permettre

6. Infrastructure As A Service
7. Platform As A Service

aux développeurs de tester un déploiement sur un espace de nom isolé, ou laisser la PIC effectuer ce test d'intégration.

Ces conteneurs et machines virtuelles ont besoin d'être déployés et orchestrés afin de garantir la répartition de charge ou encore le paramétrage. Il faut donc un logiciel d'approvisionnement [8].

Ansible , par exemple, permet de configurer ces environnements à base de composants, et de les déployer dans des environnements cibles, du développement à la production. Solution open source déjà retenue par différents grands acteurs internet, son positionnement est *"Ansible is the best way to manage Docker"*. Indéniablement porté par la vague Docker, Ansible reste cependant cohérent et pertinent dans un environnement non dockerisé.

Avec une approche technique différente, ***Puppet*** permet lui aussi de gérer des configurations et de déployer des solutions sur différents environnements. De même que ***Chef*** , relativement similaire. Ces deux derniers produits ont pour eux une maturité plus forte (2-3 ans tout de même, une éternité aujourd'hui !), une diffusion déjà plus large auprès des développeurs ou sysadmin, et des fonctionnalités avancées autour de la gestion des droits de déploiement, répondant à la problématique de sécurité associée.

Un des éléments non adressés par ces solutions porte sur la base de données, sur sa structure et ses données, sa modification dans le temps et sa compatibilité en version avec les différents environnements. Ceci est pour le moment porté par les logiciels applicatifs et pas par les outils devops, en tout cas pour le moment.

Avantages de l'Open Source

Toutes ces solutions sont distribuées sous licence open source. Certains ont des fonctionnalités complémentaires en version entreprise, mais l'essentiel est là, en open source, librement modifiable et diffusable.

Même ThoughtWorks qui a fait son succès sur la solution propriétaire GoCD l'a reversée en open source, certainement en réponse à la montée en puissance de tous les autres outils open source. La suite nous dira si cette reconversion open source tardive vient à point, ou bien trop tard ...

Quoi qu'il en soit, chacun pourra constater que ***l'open source est encore l'écosystème qui porte l'innovation*** , avec une émulation impressionnante, réussissant à faire collaborer

8. le terme "provisionning" est souvent employé

les différents acteurs et communautés, y compris des concurrents, en mettant en commun leurs travaux, tout en assurant une diffusion ultra-large et ultra-rapide, bien au-delà des canaux de distribution logiciels propriétaires.

Solutions open source

Gestion des sources

Sans gestionnaire de sources, le "Continuous Delivery" n'est presque pas possible. L'historisation des modifications, des versions, ainsi que la gestion des conflits de code ou simplement les concepts de branches sont des notions primordiales.

Il existe pléthore de services pour gérer les sources, allant de l'ancestral CVS jusqu'au très prisé Git, chacun a ses spécificités, ses lacunes, ses complexités et solutions propres.

Subversion a largement investi les projets, que ce soit dans les projets communautaires ou en entreprise, mais force est de constater que Git prend le dessus, certainement parce que ce dernier est développé par l'illustre Linus Torvalds, mais aussi parce que GitHub (plateforme d'hébergement de sources) propose un service très apprécié qui a largement évangélisé Git.

À titre indicatif, en compilant les données du "Eclipse Community Survey [1]" des années 2012, 2013 et 2014, nous arrivons à la conclusion que Git a largement conquis la communauté des développeurs (voir figure 3.1)

1. `https://eclipse.org/org/foundation/reports/annual_report.php`

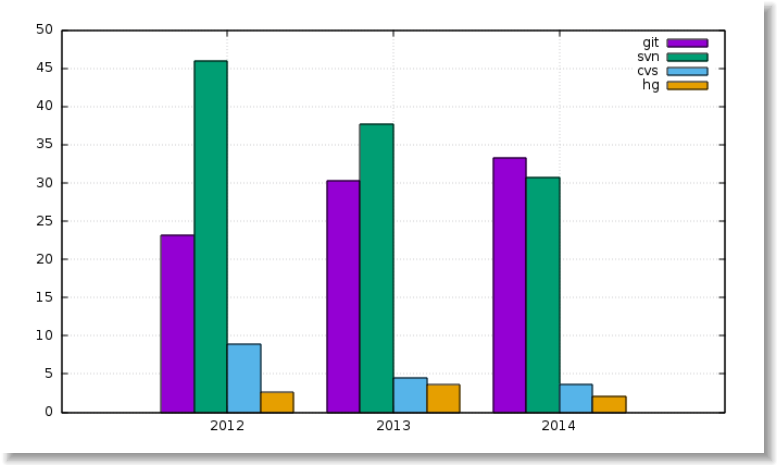


Figure *3.1 – Compilation du sondage Eclipse de l'utilisation Git, SVN, Mercurial et CVS*

En plus de ces logiciels "clients-serveurs", nous parlerons aussi des interfaces web permettant de gérer les projets avec ces outils.

SVN

Subversion (SVN) est un système de gestion de version centralisé. Issu de CVS, son développement est initialisé en 2000 par la société Collabnet. Il est devenu officiellement un projet de la fondation Apache en 2010.

Standard et populaire, il a été choisi par de nombreuses communautés du logiciel libre. De nombreux outils et ressources sont disponibles pour l'exploiter au mieux. Apache Subversion a été initialement écrit pour combler les lacunes de CVS (notamment certains choix d'implémentation historiques). Certaines fonctionnalités ont été repensées : les répertoires et méta-données sont versionnés, les numéros de révision sont globaux à l'ensemble du dépôt, il est possible de renommer ou de déplacer des fichiers sans perte de l'historique, les commits sont atomiques, etc. Le projet poursuit son évolution et intègre régulièrement de nouvelles fonctionnalités qui en font un acteur à l'état de l'art en termes de gestion de version centralisée.

Subversion est distribué sous licence Apache et BSD et est écrit en C.

Subversion ajoute, entre autre, la capacité de pouvoir "déplacer des branches", une meilleure gestion des suppressions de fichier ou encore une capacité à fusionner des branches de manières plus simple et plus efficace que son compère CVS (jusque-là le plus utilisé).

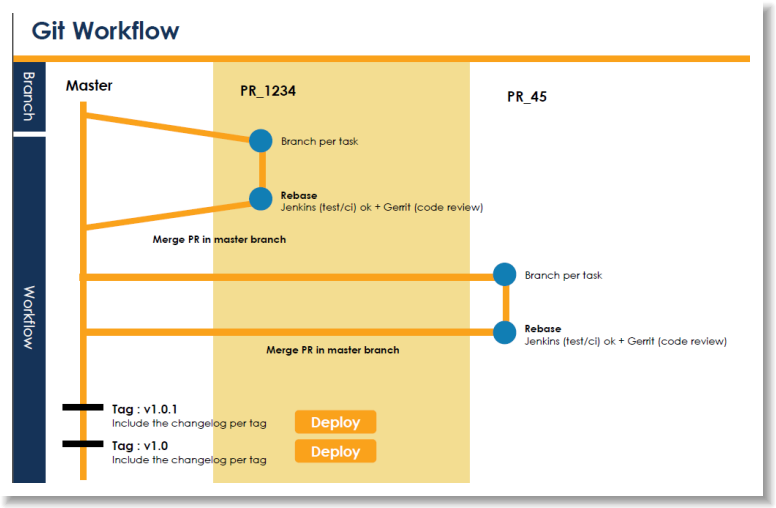
Malgré son apport substantiel d'améliorations vis-à-vis de CVS, Subversion subit la forte progression des gestionnaires de source décentralisés tels que Git, Mercurial ou encore Bazaar. Depuis quelques années, son utilisation est en baisse (figure 3.1).

Git

Git est un système de gestion de versionnement décentralisé (« DVCS »). Il est notamment utilisé pour le noyau Linux ou pour PHP. C'est un logiciel libre créé par Linus Torvalds en 2005.

Git permet notamment de "commiter" localement puis de pousser aux autres développeurs un ensemble de commits locaux. Il permet également d'utiliser un workflow de développement en soumettant par exemple l'envoi de code à l'approbation d'un des développeurs.

La faculté de Git à créer des branches facilement ainsi que de permettre leur administration de façon simple en fait un outil de choix dans le cadre de développement de projets open source.



Git est distribué sous la licence GPL v2, et est écrit en C, Bourne Shell et Perl.

Git n'est que le cœur, le client/serveur. Il suffit pour la plupart des tâches quotidiennes. Cela dit il est souvent très utile de pouvoir naviguer dans les sources, avoir une interface pour les *pull-request* ainsi que pour suivre les releases, branches et tags visuellement. C'est ce que proposent le site GitHub, mais aussi Gitorious ou BitBucket.

Cela-dit, l'entreprise préfère souvent gérer le service lui-même sur ses propres serveurs

afin de se garantir les sauvegardes et l'accès réservé. Effectivement, les dépôts git privés sont généralement payants par exemple chez Github ou BitBucket.

Le plus connu des services d'accès Git web est GitLab [2]. Développé en Ruby (tout comme GitHub), il permet l'authentification simple, OAuth, LDAP, etc. Son interface est simple et complète. Il propose en plus une gestion de ticket, permet de fermer un ticket en mentionnant le numéro dans les "commits", le *forking* de dépôt, les tests unitaires, tests de constructions, tests d'acceptations, respect de standards, etc.

Un autre outil nommé "Gogs", plus léger et plus simple, se veut être un clone de GitHub tant en termes de fonctionnalités que d'interface. Cela dit, sa jeunesse ne propose pas de lancer des tests automatiques à l'heure où nous écrivons ces lignes.

Git est fortement apprécié dans un environnement CI/CD de par sa capacité à gérer de multiples branches, des "pull-request" contrôlables par une interface claire telle que celle de GitLab, d'être facile à lier à des gestionnaires d'intégration comme Jenkins, et surtout de simplifier le flux de commits et de versions.
C'est aujourd'hui un outil ***majeur*** et ***quasi-incontournable***

Mercurial

Mercurial est un système de gestion de concurrence de version décentralisé à l'instar de Git (voir page 22).

Il existe peu de différences entre Git et Mercurial. La ligne de commande [3] offre presque les mêmes opérations à quelques différences près, et tous les deux permettent de taguer, de créer des branches, de les fusionner et créer des *pull-requests*.

Ce qui a finalement décidé de la faible proportion d'utilisateur de Mercurial face à son "concurrent" peut s'expliquer par l'utilisation de Git pour gérer les sources du noyau Linux, puis de GitHub qui a su s'imposer dans la liste des hébergeurs de projets sur le net.

Autre point, Git est développé essentiellement en C. Mercurial est développé en Python et C. C'est d'ailleurs l'une des raisons qui a poussé les développements Python (du langage aux modules) à se tourner vers Mercurial.

2. https ://about.gitlab.com/
3. la commande utilisée par Mercurial est `"hg"`

En bref, nous retrouvons les même concepts que Git, le même principe et à peu près les mêmes objectifs. Le choix entre Git et Mercurial se porte sur l'historique et les préférences de chacun.

Plate-forme d'intégration continue

Les services d'intégration continue sont des points centraux et surtout déterminants pour que le flux de livraison soit fiable. Car si les développeurs intègrent les tests unitaires et "end-to-end", il faut qu'un service s'assure pleinement de leur bon déroulement et permette de faire un rapport des régressions.

Mais il faut aussi que ces tests activent des actions selon les résultats. Mails, alertes, génération de release, déploiement peuvent s'automatiser au travers de ces plateformes d'intégration.

Certaines plateformes sont orientées pour un seul langage, d'autres sont intégrées dans l'outil de gestion de source. Il en existe un certain nombre, en voici quelques-unes qui ont mérité notre attention.

Jenkins

Jenkins est un outil d'intégration continue développé en Java descendant du projet Hudson.

Jenkins propose avant tout une gestion de tâche automatisée sur des projets enregistrés sur la plateforme.

Son principe est d'exécuter des scripts et d'utiliser le résultat de la commande pour définir le succès ou non de la construction du projet. Muni de greffons (plugins, addons) Jenkins permet aussi de rapporter des métriques intéressantes tels que des graphiques d'évolution des tests unitaires, le code coverage ou encore le respect de normes de code.

La liste non exhaustive des tâches exécutées par Jenkins est la suivante :

- l'exécution de tests unitaires

- l'exécution de tests "end-to-end"
- compilation et packaging
- déploiement continu
- etc...

La *version 2*[4] est sortie récemment et propose une interface plus moderne ainsi que la gestion de *pipelines* tel que le propose Go-CD.

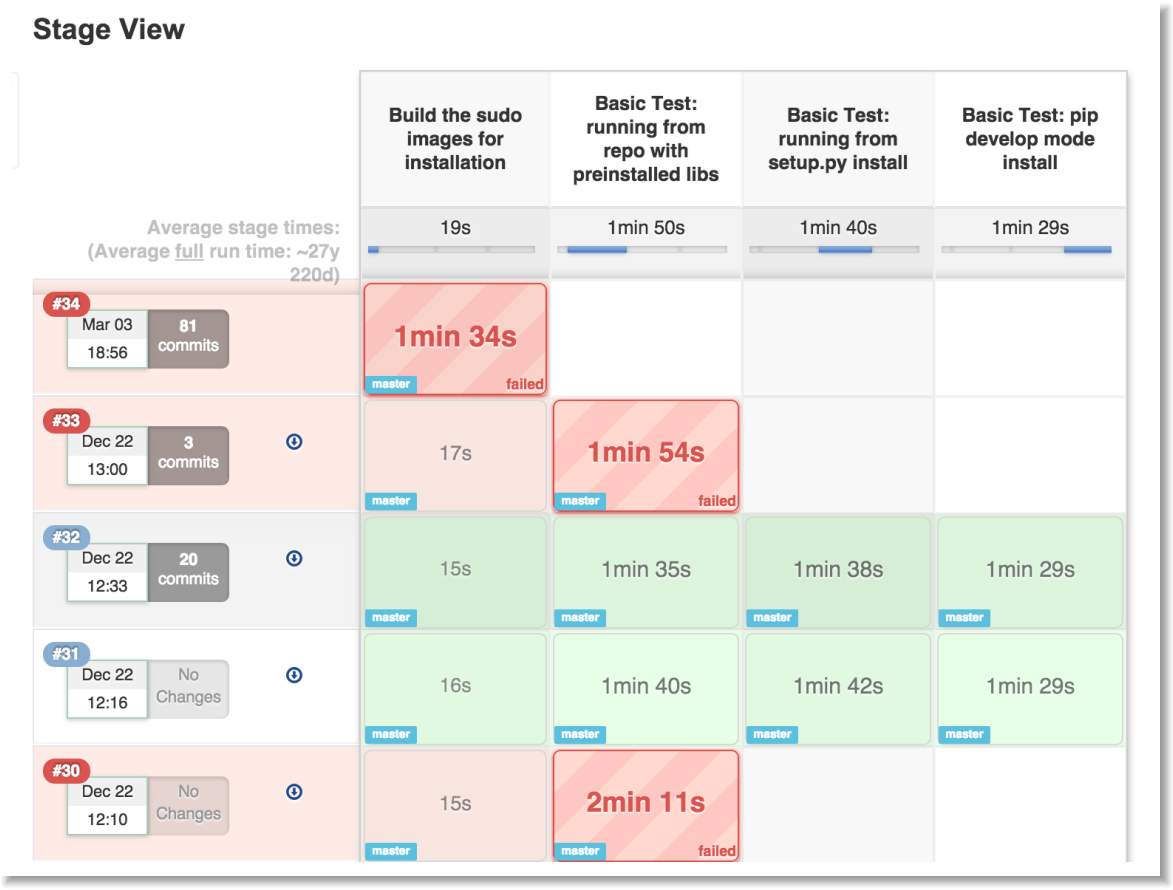


Figure 3.2 – *Jenkins 2 propose désormais une gestion de Pipeline*

Source: https://jenkins.io/2.0/

Cette nouvelle mouture intègre aussi nativement la connexion aux projets sous Git, qui était jusqu'ici activée par plugins, et propose un mode "Getting started" simplifié qui active les plugins nécessaires selon les besoins à l'installation.

4. https ://jenkins.io/2.0/

Jenkins est aujourd'hui très populaire et bénéficie d'une vaste communauté qui délivre des ***addons*** ouvrant un large champ de possibilité. La version 2 promet une interface plus claire et la gestion de ***Pipeline*** va réellement changer son utilisation dans un sens plus proche de ce qu'attendent les équipes qui travaillent en CI/CD

Gitlab - Gitlab-CI

Avec Git, il est très intéressant d'avoir une interface de gestion des projets via une interface puissante. Ne serait-ce que pour visualiser les branches, gérer les pull-requests ou lire plus facilement les logs. ***GitHub*** a largement conquis la communauté en proposant un hébergement des sources (publiques ou privées) de projets depuis près d'une décennie. Aujourd'hui, ***GitLab*** propose une interface qui peut être intégrée sur les serveurs internes et pratiquement toutes les fonctionnalités de GitHub.

GitLab est donc l'interface de gestion des projets Git (utilisateurs, projets, pull-requests, forks...) mais elle propose aussi une gestion de rapport d'anomalie [5] qui réagit aux messages de commit. GitLab a été développé en Ruby, il permet l'authentification d'utilisateurs sur un serveur LDAP, et propose aussi de rendre des projets publiques.

GitLab se décline en deux versions :

- **GitLab CE** Community Edition - version libre et communautaire
- **GitLab EE** Enterprise Edition - version sous licence propriétaire depuis 2014

Il est à noter que GitLab BV, société éditrice du logiciel, a racheté le site Gitorious ("concurrent" de GitHib et BitBucket) en 2015.

GitLab propose désormais la gestion de constructions et/ou de tests via ***Gitlab-CI*** . Ce dernier, anciennement un "addon" fait maintenant partie intégrante de l'outil GitLab.

Développé en Go, il propose un mode 'server-worker' qui permet de lancer des tests et d'effectuer des actions en fonction des résultats obtenus. Les workers sont capables d'exécuter des tests locaux, sur des conteneurs ou sur des machines distantes (tout comme Jenkins).

5. plus connue sous le nom de "issues"

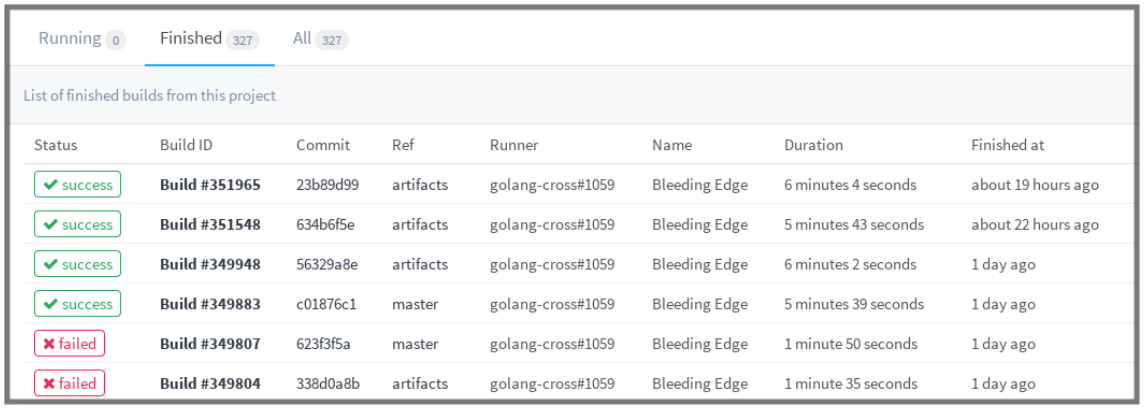


Figure 3.3 – *GitlabCI permet le rapport de construction de projet (test, compilation...)*

Source: http://docs.gitlab.com/ce/ci/quick_start/README.html

Il permet de créer un rapport de résultats et d'agir sur les ***pull-request*** dans GitLab. Les mainteneurs de projet peuvent alors connaître la capacité d'une fonctionnalité à être intégrée, si elle ne casse pas le projet et soulage ces opérations normalement faite manuellement.

GitLab-CI ne peut fonctionner, à l'heure actuelle, que sur la plate-forme GitLab.

Depuis la version 8, GitLab-CI est intégré et activé par défaut dans GitLab.

> GitLab et GitLabCI sont deux acteurs incontournables dans l'univers CI/CD. Qu'ils soient utilisés seuls ou en combinaison avec Jenkins, cette interface est aujourd'hui très développée et propose pléthores d'outils tous très utiles. Outre sa puissance, GitLab bénéficie d'une très large communauté d'utilisateurs qui a permis son évolution depuis sa sortie en 2013

Strider

StriderCD est une plateforme d'intégration qui a pour particularité d'être développée en NodeJS et utilise MongoDB pour la persistance.

Les principaux avantages de Strider sont sa faible consommation mémoire et CPU, ses plugins permettant de lancer les constructions et tests de différentes manières, et une installation simple via NPM [6].

Strider fournit aussi des plugins qui permettent de s'intégrer à différents fournisseurs tels que ***BitBucket*** , ***GitHub*** , ou des dépôts privés. Son interface est légère et fluide.

En contre-partie, il faut admettre que Strider ne rivalise pas avec les mastodontes tels que Jenkins ou GitLab/Gitlab-CI mais il peut malgré tout supporter un nombre important de projets et suffit largement aux tâches récurrentes et à l'intégration dans un flux de Continuous Delivery.

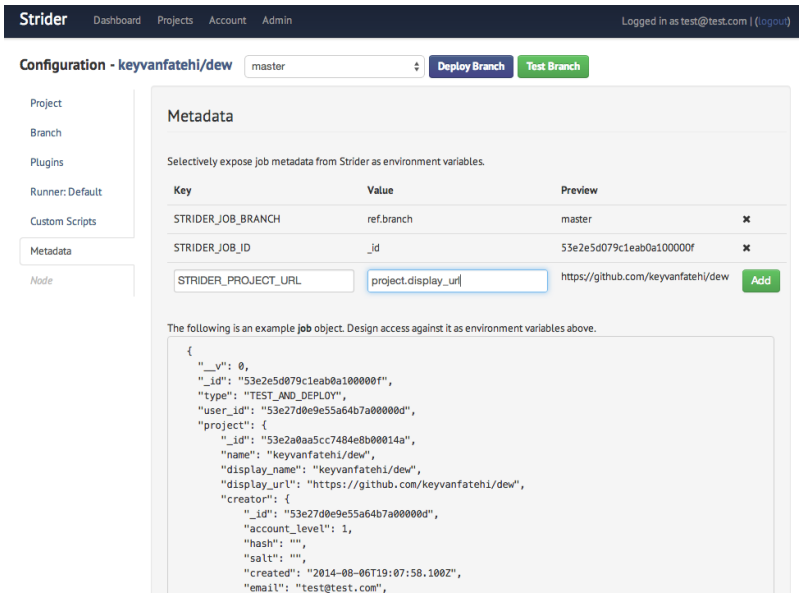


Figure 3.4 – *Exemple d'écran Strider-CI*

Source: `http://github.com/Strider-CD/strider-metadata`

À noter que ***Strider permet l'exécution dans des conteneurs Docker*** et par conséquent de déporter la charge de tests et construction sur d'autres machines [7].

Strider est un outil extrêmement intéressant pour mettre en place une PIC légère avec un minimum de configuration et sur des environnements où les ressources sont limitées. Il est très adapté aux petites équipes développant en NodeJS, tout comme en Python, Go, Java et Ruby. Cependant, cet outil n'est pas encore tout à fait mature, et bien moins connu que ses pairs.

6. Nous avons d'ailleurs conteneurisé Strider dans un conteneur Docker en quelques minutes
7. ce qui s'approche du mode "maitre-esclave" de Jenkins ou encore des "workers" de Gitlab-CI

Strider est certes récent et très épuré, cela ne lui enlève pas sa capacité à être utilisé professionnellement et de s'intégrer dans une optique CD. Sa légèreté peut aussi convaincre et ouvre la possibilité d'avoir plusieurs serveurs Strider pour différents pôles d'une entreprise afin de réduire l'impact sur l'interface Web

Gestion des déploiements

La gestion de déploiement finalise la chaîne de procédures automatiques en permettant d'intégrer une version sortie de la PIC sur un environnement de production.

De nos jours, un simple "copier-coller" des sources sur un serveur FTP n'est plus une option viable de livraison pour différentes raisons. D'abord pour des raisons évidentes de discontinuité de service durant la copie de fichier sans compter les bugs qui peuvent apparaître durant la phase de dépôt.

Mais aussi pour d'autres raisons plus techniques. Il n'est pas rare qu'une application soit mise à l'échelle (scalée) sur un cluster qui n'offre pas, en général, de "point de dépôt central". Il faut alors se pencher sur des outils d'approvisionnements ou sur des solutions de lecture direct du VCS. Voici ceux que nous maîtrisons et utilisons quotidiennement.

Avant de vous pencher sur ce chapitre, notez que deux approches vont apparaître : les services de déploiments centralisé et les services "agentless". Les deux approches ont leurs avantages et inconvénients. Le premier permet de déporter la charge sur un serveur qui centralise et orchestre le déploiement. Cela peut s'avérer extrêmement utile dès lors que des dizaines de machines sont impactées. Les services "agentless" n'ont pas de serveur d'orchestration de déploiement, ce qui peut donc paraître plus simple à installer, mais il faudra prendre en compte le nombre de machines ciblées par un déploiement.

Fabric

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.

Plus bas niveau que Ansible ou Chef, Fabric trouve son intérêt dans le fait qu'un fabfile, est codé en Python, et par conséquent permet l'utilisation de modules externes tels que "requests" ou "re" (module de gestion d'expressions régulières) et ainsi de pouvoir créer des procédures plus complexes.

Car en l'état, un "fabfile" est bel et bien un script et non un fichier de configuration.

Fabric trouve sa place dans beaucoup de projets non "Python" bien qu'il soit surtout utilisé dans des projets du même langage (Django, Zato, etc.)

Il est surtout utilisé pour *bootstraper* un projet - il permet par exemple de préparer l'arborescence locale, de récupérer les paquets et de configurer un environnement.

Des règles pour déployer le projet ou lancer les tests sont aussi une pratique courante.

Son implémentation bas niveau peut permettre de réduire certaines contraintes ou régler des cas particuliers.

Fabric peut aussi être intégré dans un écosystème plus large de déploiement, par exemple en l'utilisant dans des playbook Ansible ou pour initialiser les tests unitaires dans un job Jenkins.

Ansible

Ansible est un moteur d'automatisation de déploiement et d' approvisionnement de Cloud, gestion de configuration, déploiement d'applications, orchestration inter-services, etc.

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 service spécifique pour être provisionnées par Ansible. Il utilise SSH par défaut et se sert de "sudo" pour avoir certains droits si besoin (selon la configuration de la tâche ou du rôle).

La configuration de tâches est effectuée via des fichiers au format YAML à l'instar de Docker-Compose.

Ansible définit une série de concepts identifiés afin de modulariser les déploiements.

- **Task** tâche à effectuer. Une tâche peut effectuer plusieurs opérations telles qu'exécuter une commande, installer un paquet, préparer une arborescence ou encore démarrer un conteneur, redémarrer un service...

- **Handler** Idem qu'une tâche mais ne s'exécute qu'à la fin d'une série de tâches et si ces tâches ont effectuer un envoi d'évènement correspondant. Par exemple, une tâche d'installation de serveur Web va envoyer un évènement particulier qui sera intercepté par un handler qui redémarrera des services
- **Variables** simplement des variables utilisées par les tâches et handlers défini pour un projet
- **Templates** au format *Jinja2* - permet de créer des fichiers sur les serveurs en utilisant une syntaxe permettant l'utilisation de variables, d'effectuer des boucles ou encore des tests conditionnels.
- **Roles** permet de définir les tâches, variables, handlers, etc. pour un rôle de machine donné (par exemple "webserver", "database", "common",...)

Un rôle est clairement un ensemble de tâches, de handlers, de variables, de fichier, de templates... défini pour une installation type (eg. un rôle "fileserver" contient toutes les tâches, variables, et fichiers nécessaire à créer un serveur "NFS").

En gérant un "inventaire"[8], il est possible de définir une série de serveurs classés par rôles.

```
# exemple d'inventaire
[webserver]
192.168.1.20
192.168.1.22
192.168.1.22

[databases]
192.168.1.20
192.168.1.30
192.168.1.31
```

La création de rôles yaml, ainsi que les taches est relativement aisée, mais peut devenir plus subtile pour des environnements multiples et complexes. La structure d'une installation de serveurs maître et esclaves *Kubernetes* (page 52) peut aisément dépasser une trentaine de fichiers, le tout classé dans une arborescence conséquente. Cela-dit, la maintenance du "playbook" Ansible est appréciée.

Ansible peut être utilisé via son client en ligne de commande mais aussi en tant qu'outil intégré de déploiement. Par exemple, *Jenkins* (page 24) peut parfaitement utiliser un playbook Ansible pour provisionner des serveurs après validation d'un jeu de tests de validation.

8. ou "inventory"

En plus du "moteur" principal Ansible, la solution "Tower" est une interface de contrôle ajoutée à une API REST qui permet de gérer et monitorer les déploiements via Ansible, le tout avec une interface graphique, plus rapidement accessible.

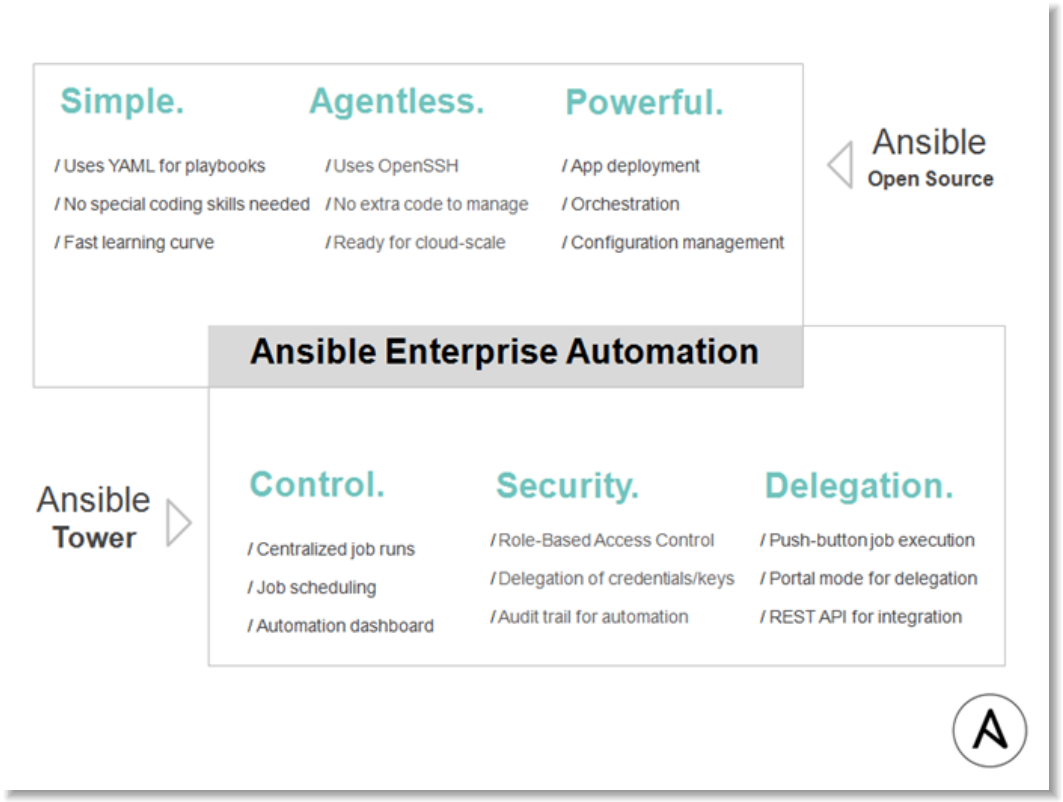


Figure *3.5 – Ansible Tower - le principe*

C'est donc un espace centralisé et visuel de contrôle de l'infrastructure permettant aussi le contrôle de rôles, d'orchestration et d'inventaire. L'API REST et le client fourni permet de s'intégrer rapidement dans une infrastructure existante.

Dans un contexte où la sécurité est importante, avec par exemple une restriction des accès aux déploiements selon les utilisateurs, ou encore des accès à la production interdits aux développements, Tower va rapidement devenir incontournable.

Entre autres, Ansible Tower permet aussi de créer des assistants de déploiement [9], de planifier des tâches (coupure de machine à une heure précise, redéploiement de conteneurs, rapports d'état d'une instance, etc) proposant alors un véritable outil d'orchestration système malléable.

9. afin de personnaliser un déploiement en répondant à des questions

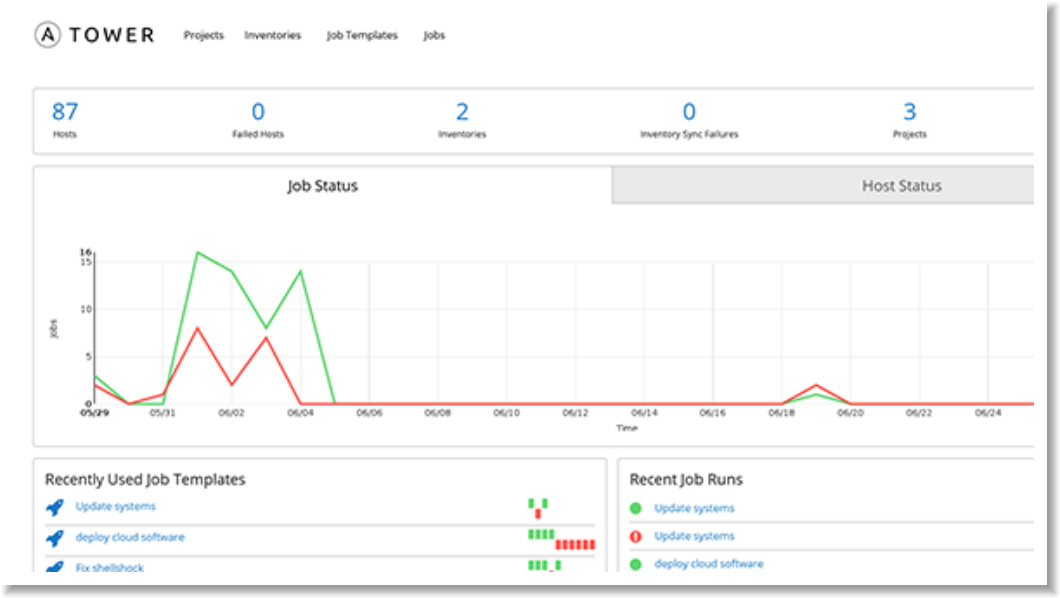


Figure 3.6 – *Ansible Tower - une interface graphique claire*

Source: https ://www.ansible.com/tower

Tower n'est pas une solution libre mais une offre gratuite qui permet l'utilisation de cette solution pour un nombre de serveurs limités.

Ansible a été racheté en 2015 par Redhat et vient compléter son offre devops. À noter qu'auparavant Redhat avait déjà fait le choix d'Ansible et l'avait intégré dans sa solution PaaS OpenShift.

Ce rachat va permettre un déploiement encore plus fort de cette solution dans le monde, et aussi pousser encore plus sa dynamique open source. La conférence de février 2016 à Londres a permis d'ailleurs d'annoncer que Tower, auparavant soumis à licence, va basculer en open source.

> Ansible est aujourd'hui très prisé pour sa souplesse et ses nombreux modules permettant de contrôler ***Docker*** , ***Kubernetes*** ou encore de déployer des machines sur ***OpenStack*** et hlAWS pour ne citer qu'eux. ***Ansible Tower*** propose quant à lui une interface de gestion des "playbook" très fins, la création d'assistante de déploiement, la planification d'exécution, la construction de formulaires, etc.

Exemple d'implémentation : Déploiment Kubernetes via Ansible

Dans un environnement OpenStack, nous avons eu l'occasion d'implémenter un playbook Ansible pour déployer Kubernetes sur une série de machines virtuelles.

Les VM RHEL7 déployées sont minimales et n'ont pas d'accès aux dépôts (restriction de sécurité). La solution a été de récupérer les binaires des services etcd, flannel, docker, kubernetes sur le poste de déploiement et de classer chaque machine dans l'inventaire afin d'identifier les rôles à assigner à ces VM.

```
[masters]
k8s-master.domain.tld

[minions]
k8s-minion-1.domain.tld
k8s-minion-2.domain.tld
k8s-minion-3.domain.tld
k8s-minion-4.domain.tld
```

Les rôles sont ensuite définis de cette manière :

Un fichier `master.yml` :

```
- hosts: masters
  roles:
    # common est un rôle en commun pour
    # installer des binaires en commun, par
    # exemple "etcdctl" ou les certificats SSL
    - common
    # les rôles pour créer un "maître"
    - etcd
    - kube-apiserver
    - kube-controller-manager
    - kube-scheduler
    # si on veut que le maître soit aussi un "minion":
    # - kubelet
    # - kube-proxy

- hosts: minions
  roles:
    - common
    # les rôles pour créer un "minion"
```

```
    - kubelet
    - kube-proxy
```

Chaque rôle, défini dans les sous-répertoires adéquats, définit alors comment déployer les binaires, assigner la configuration, déployer les certificats SSL, etc.

Une des tâches permet aussi l'installation du dashboard kubernetes [10], après déploiement et démarrage des services via Ansible.

Ce playbook Ansible spécifique à notre client lui permet de déployer des clusters kuberenetes en quelques secondes après avoir créé les machines virtuelles dans leur infrastructure OpenStack.

Et pour couronner le tout, il est tout à fait possible de définir des rôles qui utiliseront l'API OpenStack pour générer les machines en amont.

En d'autres termes, un "playbook" pourra créer les machines virtuelles, installer les outils et services nécessaires, puis provisionner les applications développées par les équipes.

Go Continuous Delivery

La solution Go-CD [11] est éditée par ThoughtWorks, une organisation précurseur sur le déploiement continu et l'architecture logicielle java.

Go-CD est construit autour d'une modélisation très puissante :

- des tâches d'intégration
- des pipelines de déploiement
- et des environnements

Go-CD est spécifiquement développé pour configurer et administrer des tâches de déploiement continu, contrairement à des plate-formes d'intégration continue comme Jenkins qui nécessitent des plugins pour le gérer partiellement. Grâce à ses agents, il peut nativement déployer les tâches sur plusieurs machines.

Les notions explicites à Go-CD :

- **Server** permet de contrôler les agents et propose une interface graphique pour paramétrer les Pipelines

10. https ://github.com/kubernetes/dashboard
11. Il est préférable de parler de Go-CD et non de "Go" pour éviter l'ambiguïté entre le langage Go et la plate-forme Go Continuous Delivery

- **Agents** sont contrôlés par le serveur et vont exécuter les Pipelines.
- **Pipelines** sont les étapes de livraison. Déclenchés par des ***Materials*** , ils vont permettre de préparer et déployer les environnements
- **Materials** sont les environnements extérieurs qui vont déclencher l'exécution d'un ***Pipeline*** .

Go-CD permet de visualiser les pipelines de déploiement et d'automatiser puis de visualiser tous les déploiements :

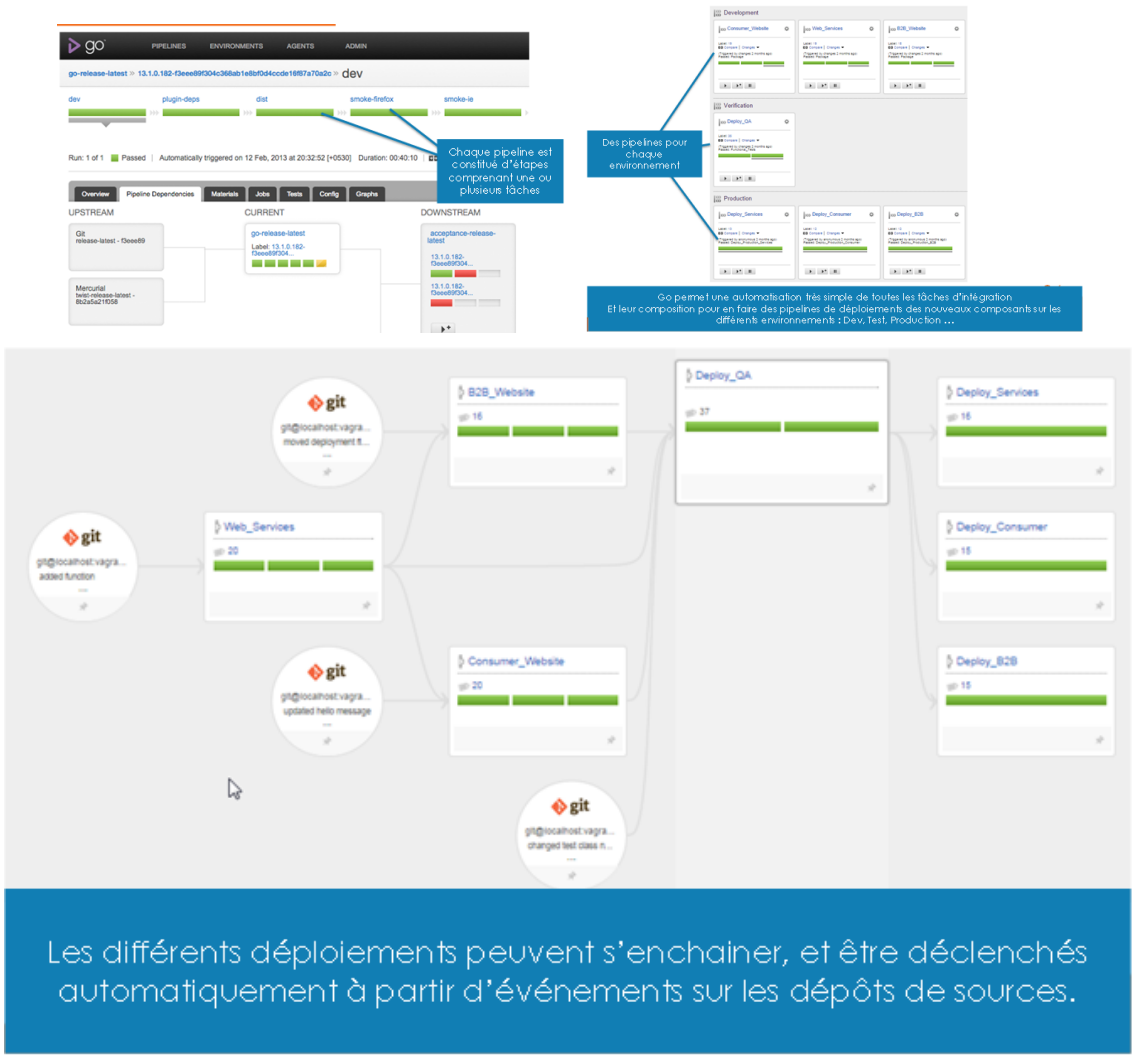


Figure 3.7 – *GoCD - pipelines*

Chef

Chef est un outil d'automatisation d'infrastructure écrit en Ruby, dont le fonctionnement est analogue à Puppet. Il est multi-plateforme et permet entre autres choses :

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

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"

La documentation présente très bien le principe de pipelines en décrivant clairement la séparation "code" et "artefact". La représentation graphique ci-dessous, tirée de la documentation de Chef, illustre les phases généralement utilisées dans Chef-Automate :

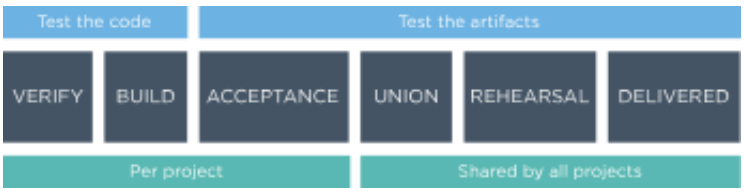


Figure 3.8 – *Pipeline du test à la livraison avec Chef*

D'autres greffons existent comme une interface Web développée avec Chef-DK [12] qui permet de maîtriser les actions, déclarer des rôles et des noeuds, etc.

Les "recettes" et "livres de recettes"

L'administrateur écrit des "recettes" qui décrivent l'état dans lequel doivent se trouver les nœuds administrés par Chef (déploiement de fichiers de configuration, installation de paquets, gestion de mots de passe, ...). Chef se charge alors d'appliquer les recettes sur les

12. kit de développement Ruby pour Chef

différents noeuds, permettant d'administrer de manière centralisée un parc hétérogène composé d'un grand nombre de machines. Tout comme Puppet, l'état des fichiers, des paquets ou des services est pris en compte pour ne pas effectuer des tâches en double ou inutilement.

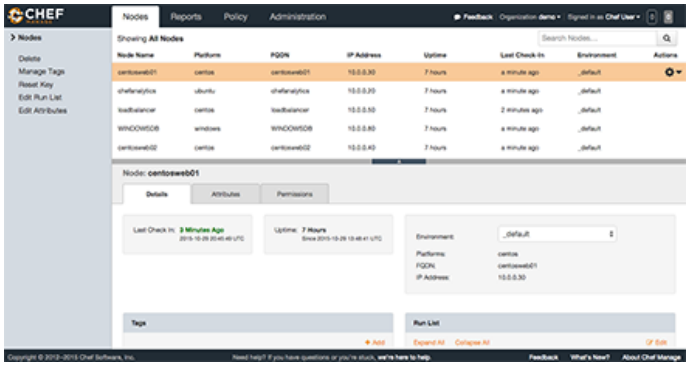


Figure 3.9 – *Gestion de noeuds à provisionner dans l'interface de Chef*

A l'instar de Ansible, Chef tend à proposer un format unique de description d'approvisionnement de machine et ce quel que soit la distribution visée.

Mais à la différence de Ansible, Chef nécessite un ou plusieurs serveurs d'approvisionnement sur lequel des nœuds sont enregistrés. Par la suite, des recettes sont appliquées sur ces noeuds via le serveur.

Les recettes sont de simples fichiers de description, par exemple :

```
package 'httpd'

service 'httpd' do
  action [:enable, :start]
end

directory '/var/www'

file '/var/www/html' do
    content '<b>Hello</b>'
end
```

Au final, une recette n'est souvent pas suffisante et la création de "CookBook" (ou livre de recettes) sera nécessaire. Un Cookbook est une structure fichiers qui regroupe dans son arborescence des méta-données, différentes recettes, des templates, des spécifications...

La ressemblance avec un "playbook" Ansible n'est pas un hasard, ce principe de rassemblement de tâches et états à gérer est une notion commune à presque tous les outils de "provisionning".

Chef-Automate

Finalement, la partie la plus importante en ce qui concerne le CI/CD est bel et bien Chef Automate qui va permettre de gérer des "pipelines" et des "cookbooks".

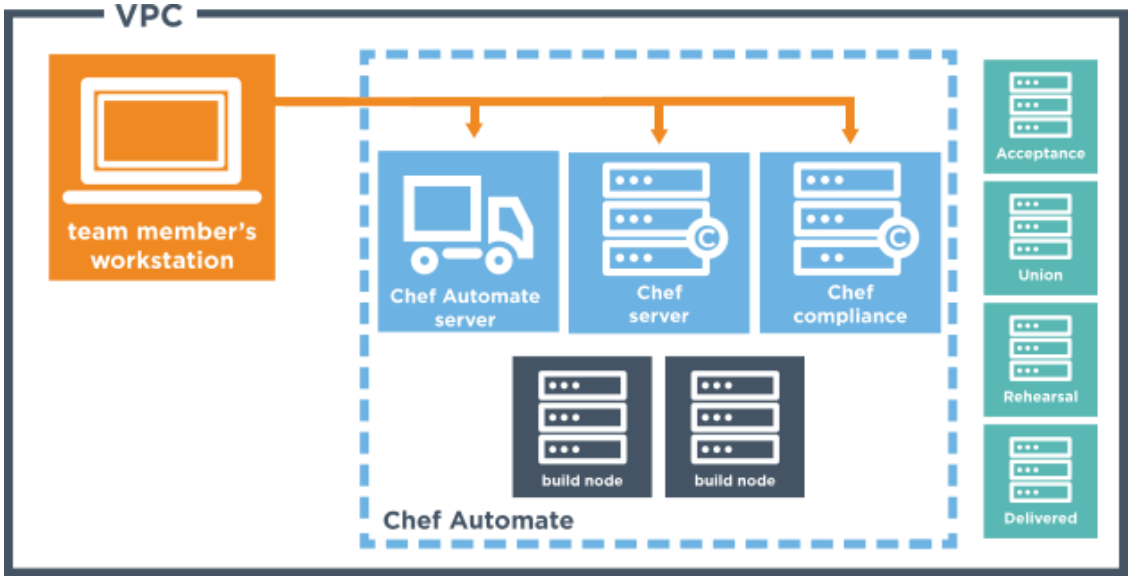


Figure *3.10 – Scénario de Chef-Automate*

Source: https ://learn.chef.io/automate/install/install-chef-automate/

Ici, l'équipe provisionne les services pour qu'un nœud de construction (build node) puisse construire un *artefact d'installation*. Chef-Compliance va permettre de faire différents tests (analyses, tests d'état, certification manuelle si besoin, ...) de cet artefact sur le nœud et, finalement, le déploiera dans un "pipeline" pour que des tests d'acceptation. Après cette ultime phase, l'artefact de livraison peut être déployé.

Le nombre de phases peut donc être relativement important au sein de la structure "Automate" et avoir un nombre de nœuds en conséquence aidera à paralléliser ces automatisations.

Chef est un logiciel libre distribué sous licence Apache.

L'intérêt de Chef réside essentiellement dans sa capacité à gérer des dizaines de milliers de nœuds répartis sur différentes grappes et via plusieurs serveurs centraux. Le client

fourni des analyses des déploiements de recette très lisibles et l'interface Web permet de maîtriser l'état du parc de manière claire.

> Chef est un outil poussé, proche de la philosophie de Puppet. De nombreuses idées introduites par Chef ont d'ailleurs été reprises par Puppet. C'est un ensemble très intéressant dans des processus de validation et d'approvisionnement sur de larges parcs de production et il s'intègre très bien dans la logique DevOps

Puppet

Puppet [13] est un outil d'automatisation d'infrastructure très largement répandu.

Puppet est réalisé en Ruby et la version communautaire est disponible sous licence GPL.

Puppet utilise un langage déclaratif (au lieu de décrire une suite d'actions à réaliser, comme avec les outils d'administration classiques), l'administrateur saisit l'état qu'il souhaite obtenir (permissions souhaitées, fichiers et logiciels à installer, configurations à appliquer), et Puppet se charge automatiquement d'amener le système dans l'état spécifié quel que soit son état de départ. ***Puppet permet ainsi d'administrer un grand parc hétérogène de façon centralisée*** .

Figure 3.11 – *Le prinicipe d'état de machine utilisé par Puppet*

Le langage déclaratif Puppet est un graphe orienté, avec une gestion de dépendance fine. Il est même possible de visualiser ce graphe via GraphViz. Ce n'est pas un langage procédural, dans lequel chaque tâche serait exécutée de manière séquentielle.

13. `https://puppet.com/`

Fonctionnement

Puppet repose sur l'utilisation d'un agent qui doit être installé sur les machines "puppetisées". Cet agent existe sur de nombreuses plateformes : Linux (toutes distributions), BSD, AIX, HP-UX, MacOS-X, Windows...

La communication entre le client et le serveur Puppet se fait par un canal HTTPS (port 8140 par défaut) avec une authentification par certificats SSL. Puppet inclut pour cela une mini PKI (Infrastructure à clé publique) qui simplifie la gestion des autorisations.

À partir de la version 4, l'agent est contenu dans un seul paquet RPM qui embarque directement toutes les dépendances (Ruby). Cela simplifie grandement l'installation de l'agent et sa portabilité. L'agent fonctionne en mode « Pull », c'est-à-dire que c'est lui qui interroge le serveur de manière régulière, afin de récupérer le catalogue et en déduire les actions adéquates.

De plus, Puppet peut être connecté à un composant de persistance de données nommés PuppetDB. Celui-ci stocke l'état de tous les serveurs, ainsi que les rapports d'exécution de Puppet. Il dispose d'une API REST documentée, et permet de construire facilement un dashboard javascript (le client interroge directement PuppetDB) .

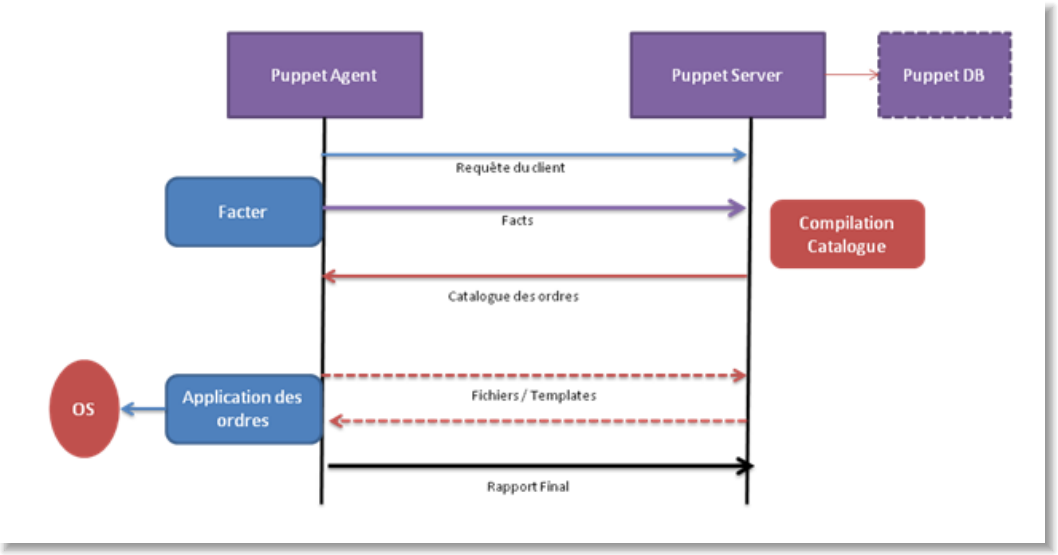


Figure 3.12 – *Interface REST entre agent et serveur*

Exemple

Le code Puppet est regroupé en modules, qui contiennent des classes contenant elles-mêmes des ressources. Les ressources Puppet standard sont cross-platform, elles

s'adaptent automatiquement à votre plateforme cible.

Exemple pour un serveur web :

```
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"]
  }
}
```

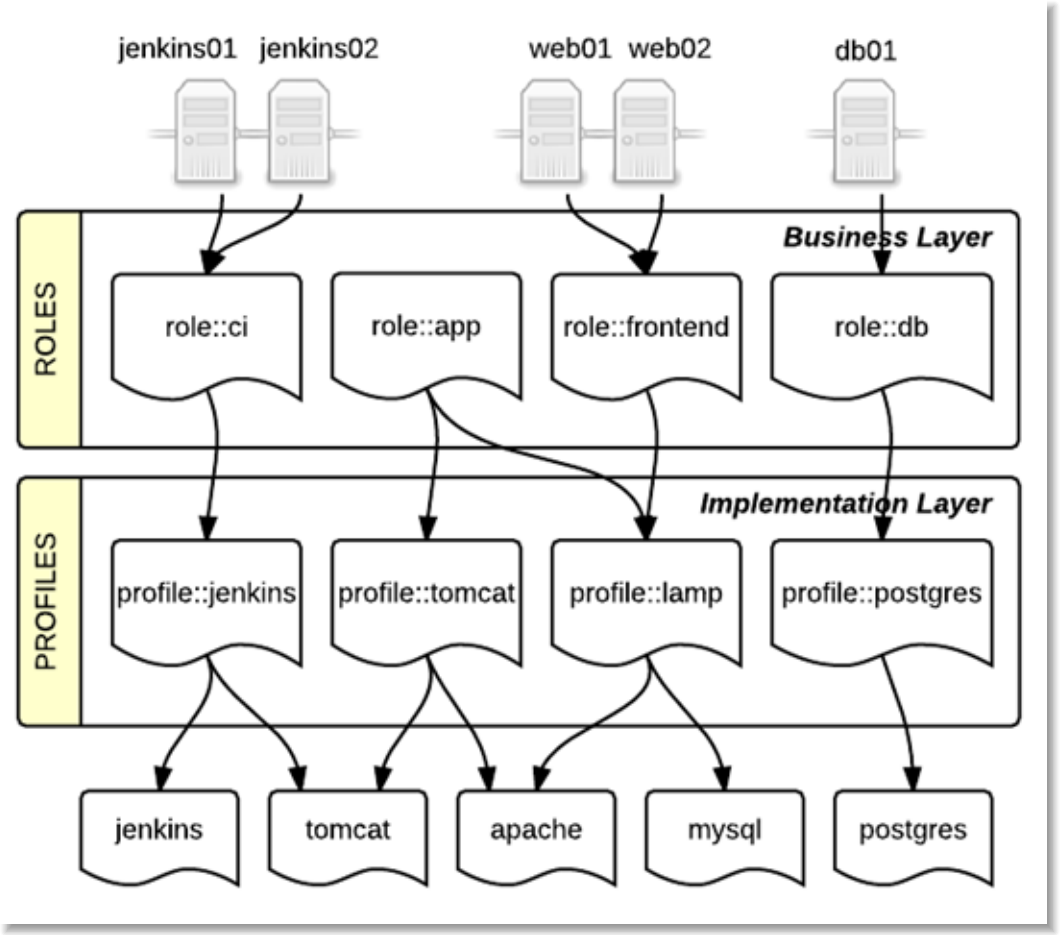
La séparation code/configuration : Hiera

Puppet permet la séparation propre de votre code avec vos variables de configuration qui dépendent de vos environnements (Ex : Ports d'écoute, répertoires, noms de serveurs). Pour cela Puppet met à disposition Hiera. Cet outil permet d'externaliser toute la configuration hors de vos modules. La configuration est alors stockée dans un puit de configuration qui prend la forme de fichiers yaml (il existe plusieurs providers hiera). Les fichiers yaml de hiera sont surchargeables, par environnement, par serveur, ou par facts (Ex : OS, Adresse IP)

La nécessité d'un bon design pattern

Lorsque votre base de code Puppet grandit, il est important d'utiliser dès le début des projets des bonnes pratiques, pour éviter la duplication de code. L'état de l'art de l'écosystème Puppet recommande pour cela le design : Rôles / Profils. Ce modèle permet d'ajouter une couche d'abstraction entre le serveur provisionné (le node en langage Puppet) et les modules qu'on lui affecte.

Chaque node inclut un seul rôle et chaque rôle appelle la notion de profils.



Communauté

Puppet bénéficie d'une communauté d'utilisateurs enthousiastes et dynamiques et de nombreux modules prêts à l'emploi sont fournis via le site communautaire PuppetForge (>4000 modules).

Puppet dispose en standard d'une commande permettant d'installer et de mettre à jour un module et de gérer correctement ses dépendances. Attention tout de même à la qualité de ses derniers qui n'est pas uniforme…

Ces modules prêts à l'emploi permettent entre autre :

- De gérer et configurer les bases MySQL
- De gérer votre serveur tomcat
- De configurer l'authentification Active Directory sur vos machines (en deux lignes)
- D'installer et de gérer votre Solr, ElasticSearch…

Une version entreprise, assortie d'un support professionnel, est fournie par son éditeur PuppetLabs.

Gestion des conteneurs et Clusters

Le déploiement continu s'effectue sur des serveurs de production, et bien que ces livraisons puissent s'effectuer sur des serveurs de manière traditionnelle (sans conteneurs), la tendance vise à approvisionner des "clusters" capables de gérer des conteneurs (applicatifs ou non, via Docker, LCX, OpenVZ, etc.)

Tout d'abord parce que les ressources matérielles ne sont plus nécessairement importantes dans le cas d'utilisation de conteneurs applicatifs. Comme nous l'avons expliqué précédemment, un conteneur ne gérant pas d'émulation matériel ni d'OS à proprement parler, le nombre de micro-services qu'un serveur est capable de gérer est relativement élevé.

Et d'autre part, parce qu'à l'autre bout de la chaîne, du coté développeur, l'utilisation d'un conteneur applicatif est certainement déjà en vigueur. De ce fait, l'image de conteneur utilisée par les développeurs sera la même que celle utilisée sur le serveur. Et comme vu dans les pages précédentes, la PIC aura aussi utilisé cette même image ce qui va considérablement réduire le risque de dysfonctionnements dû à une différence de configuration logicielle.

Aujourd'hui, nombreuses sont les solutions qui vont permettre de s'abstraire du déploiement brut sur un serveur et passer de préférence par une solution d'orchestration de conteneurs. Un simple fichier "manifest" définit comment démarrer un conteneur et la configuration nécessaire. Le service va alors trouver une machine disponible, démarrer l'application et faire le nécessaire pour que développeurs, équipe de validation et clients aient accès à celui-ci.

Un dernier point essentiel, la gestion de conteneurs via un orchestrateur va permettre, par la suite, de gérer la "mise à l'échelle" (ou scaling).

Même s'ils ne sont pas une obligation, les conteneurs sont aujourd'hui une des clefs de succès du Continuous Delivery.

Docker, Docker Compose, Docker Swarm

Développé par le français Solomon Hykes, logiciel libre développé en Go, Docker a rapidement conquis les administrateurs systèmes ainsi que les développeurs.

Docker est une solution de gestion d'images et de conteneurs applicatifs.

Les images et conteneurs

Une image propose un service installé dans un environnement spécifique, le conteneur crée alors une instance de l'image pour utiliser le service. De plus, une couche réseau et une gestion de volumes permet d'isoler le service.

Basé sur la librairie "libcontainer", Docker utilise aussi une méthode dite "union FS" qui permet de proposer des couches de système de fichier plutôt que de la duplication. C'est d'ailleurs la représentation schématique de ces couches qui a valu son nom à Docker.

L'intérêt grandissant de Docker se porte sur sa souplesse et sa communauté qui a rapidement adopté la solution. Le "hub", serveur de registre d'images officiel de Docker, propose une série d'images pré-configurées, mais aussi des images spécifiques proposées par la communauté. Le système de registre est librement utilisable et permet d'héberger son propre serveur interne.

Docker-Compose

Afin de simplifier la création de conteneur et les liens entre eux, un outil nommé "docker-compose", initialement nommé "fig", est aujourd'hui utilisé. Il s'agit de créer un fichier au format "yaml" pour définir quels services lancer, comment les lier, quels ports utiliser, etc.

Un exemple de composition :

```
version: '2'
services:
    # on définit un service "drupal"
    drupal:
        image: php/fpm # image du "hub" à récuperer
        # on monte les sources dans le répertoire
        # de données web du conteneur
        volumes:
        - ./src:/var/www/html
        ports:
        - 8080:80
        - 8443:443

        links:
        # le nom "database" sera résolu
        # dans le conteneur
        - database

    # définir une base de donnée
    database:
        image: mariadb
```

Ci-dessus sont définis deux services dont les noms sont choisis de manière arbitraire : "drupal" et "database". Une liaison entre les deux conteneurs est aussi déclarée par la directive "link". De cette manière, le conteneur "drupal" peut utiliser le nom "database" comme adresse de base de données. La résolution de nom est effectuée par diverses méthodes en fonction du type de réseau utilisé par les conteneurs.

Les développeurs qui utilisent ce fichier vont alors simplement utiliser la commande "docker-compose up" qui va automatiquement télécharger (pull) les images nécessaires et démarrer les services tels qu'ils sont décrit pas les images. Docker repose sur le même principe que "git". Les images ont des versions (tags) qui permettent de proposer plusieurs alternatives. Par exemple, les images PHP proposent un mode "fpm", un mode "apache", et avec diverses distributions de base (ubuntu, alpine, ...) Mais il propose aussi la possibilité d'adapter les images par le biais d'un fichier nommé "Dockerfile". Dans ce fichier, il est possible de préparer une configuration spécifique, compiler ou installer d'autres paquets afin de coller au plus près à la configuration de la production ou pour combler les besoins. Par exemple, un Dockerfile basé sur l'image "node" qui propose en plus les outils "gulp" et "bower" :

```
FROM node:latest
RUN npm install gulp-cli bower && npm cache clear

CMD ["gulp", "serve"]
```

Ainsi, il est possible de construire l'image (docker build), et de l'utiliser dans un projet. L'image peut être déposée dans un registre, mais il est aussi possible de fournir le fichier Dockerfile au projet et de laisser "docker-compose" construire l'image. En admettant que ce fichier se trouve dans le sous-répertoire "dockerfiles", le fichier "docker-compose.yml" pourra définir que le service "nodejs" utilise l'image résultante :

```
version: "2"
services:
    nodejs:
        build: ./dockerfiles
        volumes:
        - ./src:/workspace
```

Pour résumer, Docker permet de proposer un environnement de service homogène quelle que soit la distribution utilisée. Les administrateurs systèmes peuvent alors fournir des images ou des fichiers Dockerfile qui se rapprochent au mieux de la configuration de production et assurer alors un déploiement moins risqué.

De plus, les solutions de cluster basées sur Docker tels que CoreOS/Fleet, Kubernetes, Mesos, NomadProject, permettent de mettre en production une réplique exacte de l'environnement utilisé par les développeurs. La jonction administrateur-développeur se réduit fortement.

MesOS

Mesos est un gestionnaire de cluster développé par la fondation Apache. Il "fournit une isolation efficace des ressources et le partage entre les applications distribuées, ou des frameworks" [14].

Mesos seul n'est qu'une infrastructure qui distribue les processus sur les machines du cluster. Il isole et partage les ressources à la manière d'un "noyau distribué" en permettant de faire tourner plusieurs systèmes sur un seul et même cluster. Mesos abstrait le principe de processeur, de mémoire vive, et de ressources en général ce qui lui vaut d'être comparé à de la ***virtualisation élastique*** .

Il est très apprécié par les utilisateurs de Hadoop et le Big Data en général qui profitent du mécanisme efficace de partage de ressources.

C'est aussi un socle qui permet d'exécuter des frameworks ayant différentes attributions. Par exemple, Chronos qui est un framework permettant d'exécuter des tâches planifiées (en quelque sorte, un genre de "crontab" haute disponibilité), ou Aurora qui permet l'exécution de services à long terme en profitant du partage de ressources de Mesos et la ***tolérence de panne*** .

Les nœuds sont paramétrés dans des fichiers de configuration et c'est le serveur 'ZooKeeper' qui fait office de serveur clé/valeurs pour partager la configuration. L'interface "basique" de Mesos permet de contrôler les ressources utilisées sur le cluster.

14. https://en.wikipedia.org/wiki/Apache_Mesos

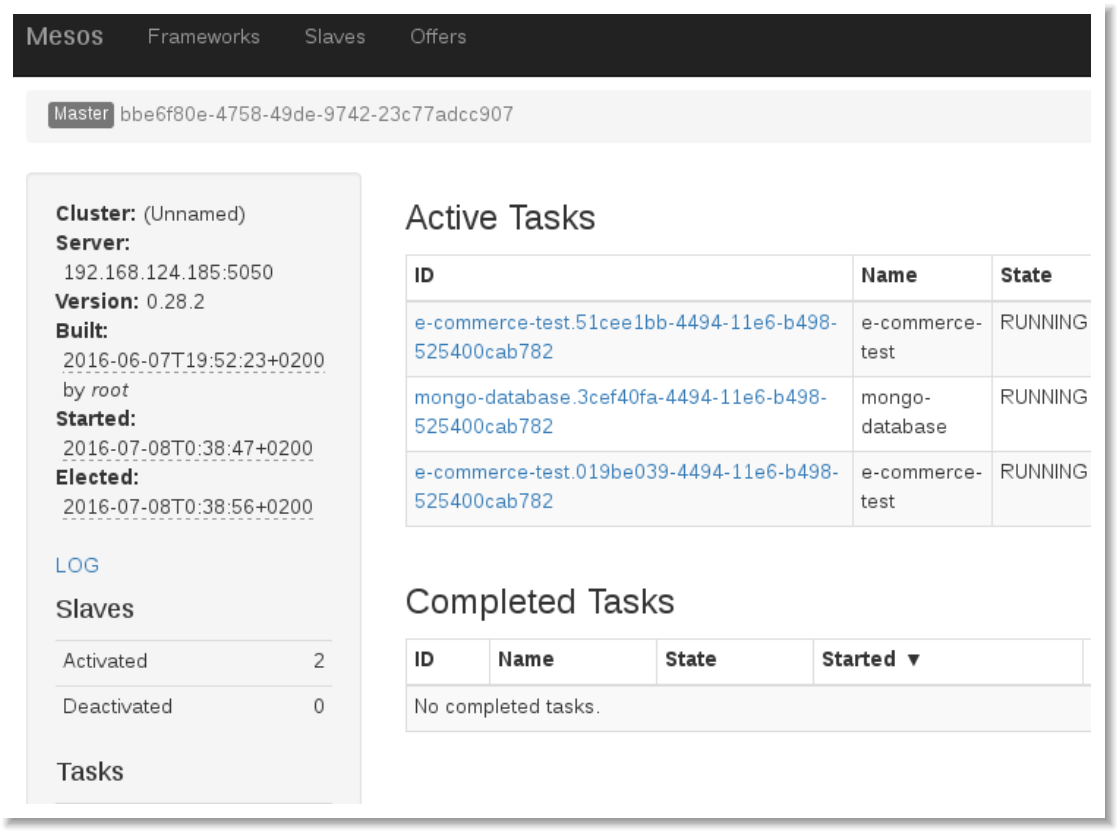


Figure *3.13 – L'interface "basique" de Mesos*

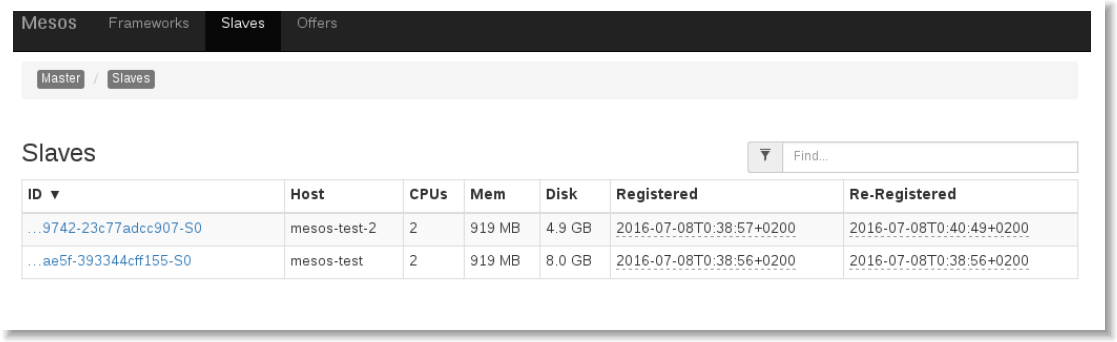


Figure *3.14 – Vue de 2 "esclaves" Mesos dans un cluster*

Mais le framework en vogue reste le très prisé ***Marathon*** - un PaaS efficace permettant de démarrer des applications conteneurisées via Docker. À l'instar de Kubernetes, Marathon donne la possibilité de démarrer des conteneurs sur différents nœuds (esclaves) et donc de "scaler" le service.

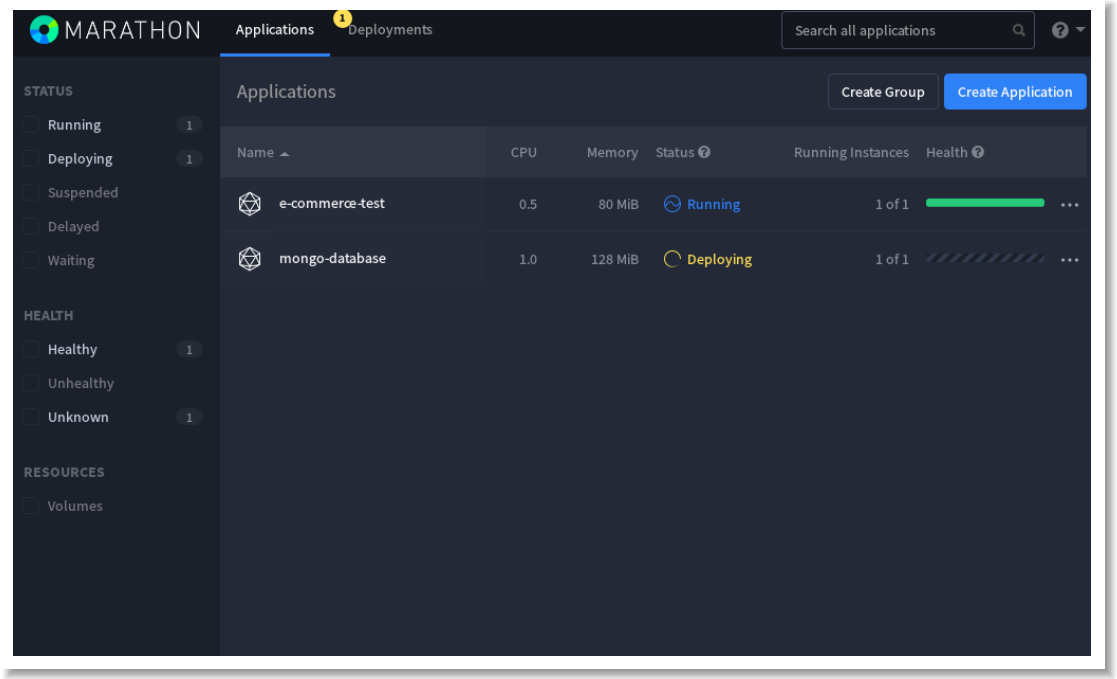


Figure *3.15 – L'interface de Marathon*

L'interface graphique est claire et relativement simple. Mais toutes les opérations peuvent être contrôlées et exécutées via l'API REST.

```
~ curl -s http://mesos-test:5050/system/stats.json | json_pp
{
   "avg_load_1min" : 0.02,
   "mem_total_bytes" : 1929216000,
   "cpus_total" : 2,
   "avg_load_15min" : 0.05,
   "mem_free_bytes" : 667828224,
   "avg_load_5min" : 0.04
}
~
```

Figure *3.16 – L'API permet de contrôler l'état du cluster ou d'effectuer des opérations*

L'interface Web utilise finalement cette API pour faciliter le paramétrage, le monitoring et les états de conteneurs.

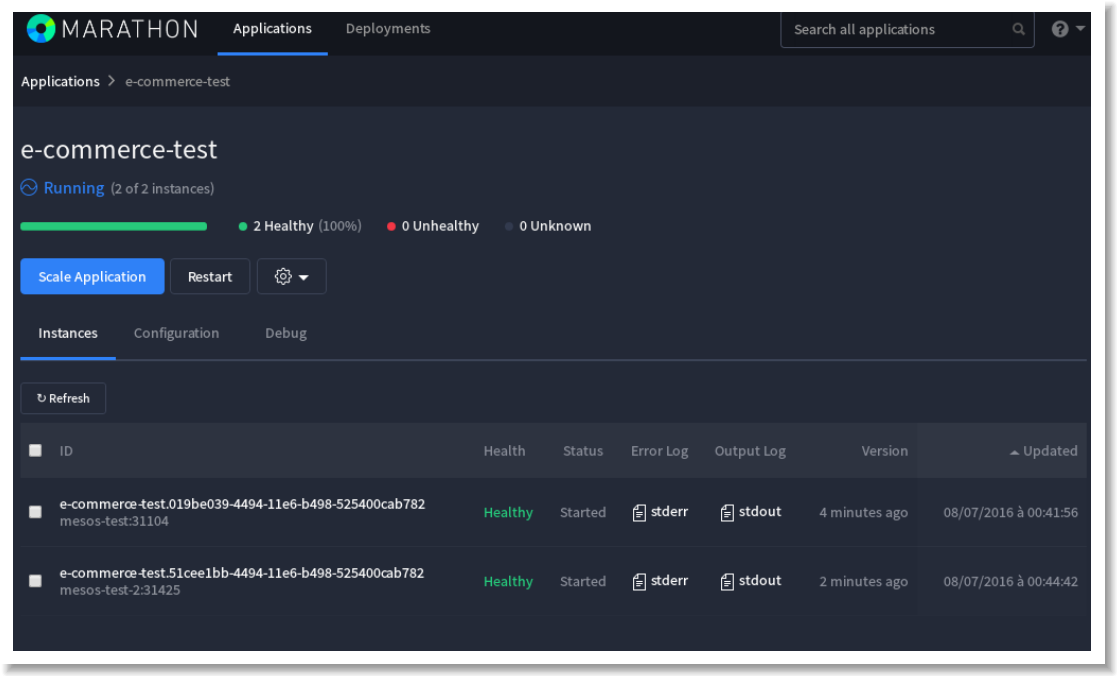


Figure 3.17 – *État des conteneurs sur les nœuds*

Mesos est *cross-platform*, il fonctionne sur les systèmes Linux, OSX ou Windows.

L'engouement de Marathon masque malheureusement toute la capacité que propose Mesos. En effet, Docker étant en vogue, et Mesos permettant le démarrage de conteneurs via Marathon, un amalgame est souvent fait entre ces deux derniers.

Il faut retenir que Mesos est une solution bas niveau sur lequel il est possible, entre autre, de gérer des conteneurs, mais pas seulement !

Nous avons aussi la possibilité de faire tourner des services "directement" via des frameworks plus ou moins prêt à l'emploi. Par exemple, le framework ElasticSearch [15] permet d'exécuter ce service sans passer par la conteneurisation, ou encore le framework Jenkins [16] qui permet d'exécuter les processus de construction sur des esclaves Mesos.

Et nous n'avons pas non plus parlé de l'intérêt de Mesos pour faire tourner des solutions telles que Kafka, Hadoop, Spark et de la Mesosphere [17] qui propose des outils tels que DC/OS, Velocity et Infinity afin de démarrer et gérer votre plateforme en utilisant différentes technologies sur Mesos.

15. https ://github.com/mesosphere/elasticsearch-mesos
16. https ://github.com/jenkinsci/mesos-plugin
17. `https://mesosphere.com/`

Les possibilités sont nombreuses et Mesos est un outil extrêmement puissant, capable de permettre la clusterisation et la scalabilité confortable.

> La popularité de Mesos ne cesse de croître et la fondation Apache, déjà forte de ses succès tels que Tomcat et Httpd, apporte une fois de plus une solution très puissante et adaptée à l'entreprise. Mesos est une solution complète mais aussi complexe à mettre en oeuvre. Cela-dit, Mesos apporte énormément de confort et d'efficacité dès lors que le besoin en calcul et ressources partagées devient primordial.

Kubernetes

Kubernetes est un gestionnaire complet de provisionnement de conteneurs sur un cluster. En d'autres termes, Kubernetes associé à Docker permet d'avoir un PaaS en simplifiant sa mise en oœuvre.

Kubernetes est en fait une association de plusieurs services développés en Go. On retrouve donc généralement le service d'API, le contrôleur de gestion (controler manager) et le planificateur (scheduler) qui vont généralement tourner sur le serveur maitre. Les "esclaves" (appelés "minions") vont simplement faire tourner deux services : kubelete qui va démarrer le conteneur en respectant la demande du contrôleur, et un service "proxy" pour permettre la réponse des services (y compris des conteneurs qui tournent sur d'autres minions)

Ces services utilisent Etcd, un serveur clés/valeur développé et utilisé par CoreOS. Afin de communiquer de conteneur à conteneur sur des machines distantes, un service de réseau superposé (ou "overlay network") tel que Flannel sera opportun.

Tous les noeuds utilisent Flannel et Etcd pour communiquer et échanger des informations.

À cela s'ajoute une API REST qui donne accès aux informations du cluster, un interface Web [18] qui permet de créer aisément des Pods (voir ci-dessous) sans accéder au terminal ou encore un client en ligne de commande nommé "kubectl" permettant, même à distance, de paramétrer le cluster, les services, etc.

18. sous forme d'addon

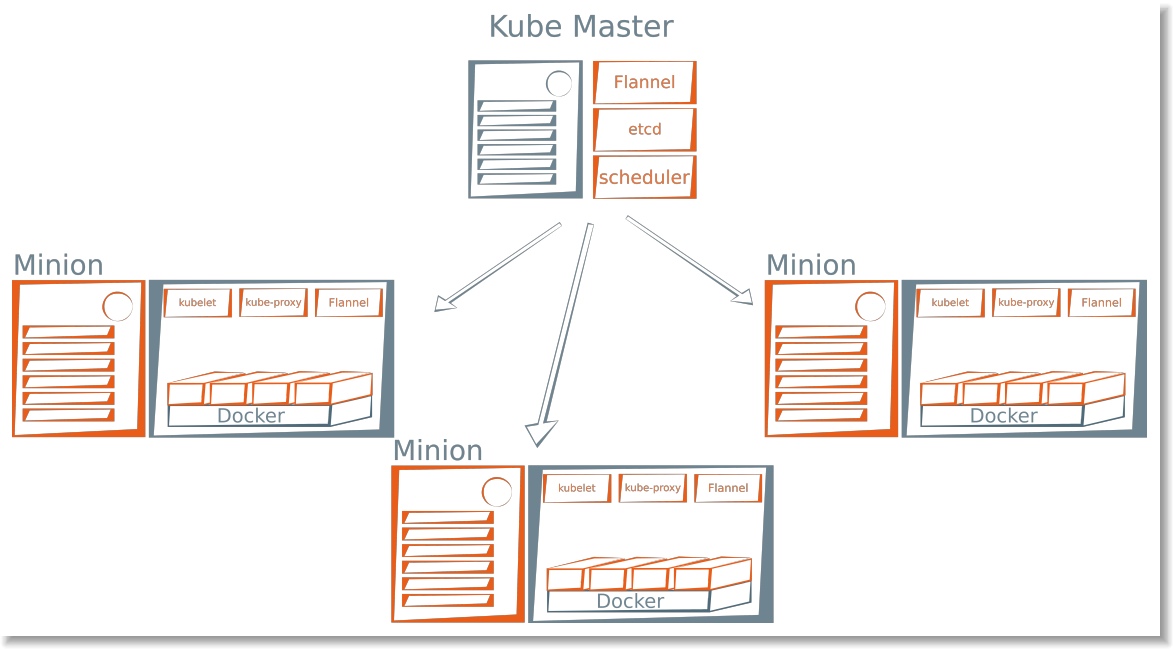


Figure *3.18 – Cluster Kubernetes*

Kubernetes apporte son propre vocabulaire à assimiler.

- **Pod** Entité englobant plusieurs conteneurs liés entre eux
- **Service** Point d'accès à une ressource encapsulée dans un Pod
- **Replication Controller** qui permet de déclarer des Pod dont le nombre de répliquas est malléable
- **Deployment** est comme un "Replication Controller" mais permet aussi de mettre à jour les versions de service sans interruption de service
- etc.

Afin de décrire une ressource à utiliser, par exemple un service web, une base de données, une réclamation d'espace disque... il suffit de charger un fichier YAML qui décrit le type de ressource et ces spécifications.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: website-rc
  labels:
    app: drupal
```

```
spec:
  selector:
    name: website
    version: 1.0
  template:
    metadata:
      name: website-app
      labels:
        name: website
        version: 1.0
    spec:
      containers:
        - image: drupal:8
          name: frontend
          ports:
            - containerPort: 80
            - containerPort: 443
        - image: mariadb
          name: database
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: XXXXXX
```

Une autre manière de faire est de renseigner ces informations dans le "dashboard" proposé par Kubernetes. Cela dit, tout l'intérêt à utiliser un format structuré est de pouvoir utiliser ***Ansible*** , ***Puppet*** ou simplement un job ***Jenkins*** pour démarrer, ou mettre à jour une entité de service.

Kubernetes est le coeur de ***OpenShift*** (que nous verrons à la section suivante) et l'on peut d'ailleurs repérer un certain nombre de concepts communs entre les deux outils. La gestion d'espace de nom, les volumes persistants sur différents pilotes (glusterfs, nfs, RDB, Ceph...), la notion de "Pod", "Replication", etc.

Il peut s'installer sur à peu près tout type de serveur Linux et utilise Docker comme gestionnaire de conteneur par défaut. Mais il permet aussi d'utiliser d'autres pilotes de conteneurs (à ce jour, seuls Docker et Rkt sont supportés officiellement).

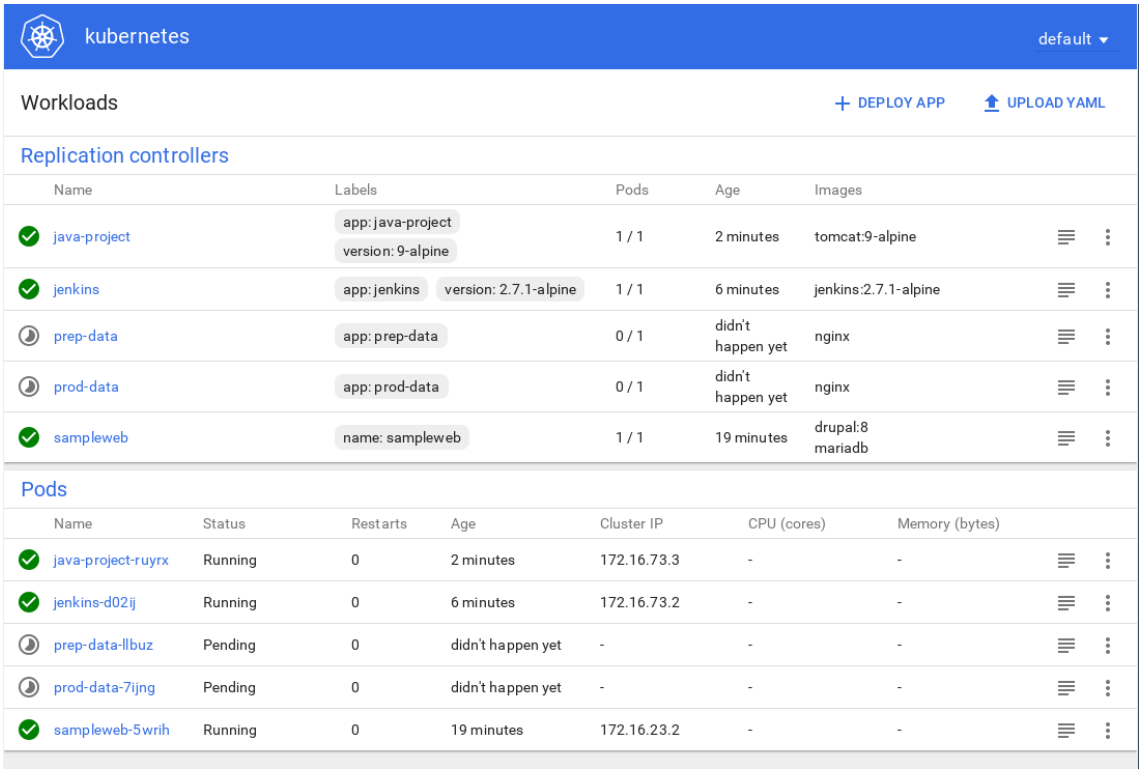


Figure *3.19 – Kubernetes Dashboard permet de contrôler le cluster*

Sa communauté grandissante a démontré son utilisation dans des environnements de cluster tels que CoreOS, OpenStack, AWS, DigitalOcean…

Kubernetes intègre d'ailleurs des mécanismes pour manipuler différents répartiteurs de charges (Load Balancer) via un "objet" spécifique nommé Ingress, et dont Traefik, Vulcan ou d'autres gestionnaires de configuration pour HAProxy, Apache HTTPd ou Nginx peuvent se servir. On retrouve donc l'essentiel des besoins pour intégrer une plate-forme de test et même de production.

Un mode tout aussi intéressant est de faire en sorte qu'un service ouvre un port identique sur tous les Minions qui permet de se connecter au service choisi. Même si cet esclave ne sert pas le Pod en question, Kubernetes trouve un noeud qui pourra répondre à ce service.

Kubernetes est une solution provenant du prolifique Google ce qui le gage d'une pérennité quasi-sûre. Il est simple à mettre en oeuvre et ne demande pas énormément de ressources pour tourner. Cela-dit, bien que le cluster soit rapidement mis en œuvre, la création de contrôleurs de réplication, ainsi que la gestion des volumes demande de la pratique et de l'analyse sitôt que les services liés deviennent conséquents.

Exemple d'implémentation : Bac à sable Kubernetes

Un de nos clients *Smile* avait besoin d'un *bac à sable*[19] pour démarrer des services à la volée mis en conteneur. Travailler avec Docker était pratique et l'utilisation de docker-compose était déjà dans les habitudes des développeurs.

Cela dit, il était difficile de tester et de vérifier efficacement la scalabilité des services. L'utilisation de VM dans un environnement OpenStack était à l'ordre du jour, mais le paramétrage des images et conteneurs pour les faire communiquer de machine à machine s'avérait difficile.

Kubernetes abstrait cette complexité en permettant la création de 'pods' dans lesquels les conteneurs communiquent via l'adresse localhost. La création de fichier de description est aisée (en YAML) et Ansible leur permet de déployer rapidement un pod, ou de le supprimer. Les Pods étant tous identifiés et isolés par Kubernetes, les déploiements n'entraînent pas de conflits entre services. Chaque équipe peut isoler, de plus, les pods dans des espaces de noms spécifiques.

A tout moment, il est possible de *scaler les applications* , de tester la réaction des services, de les couper et/ou les supprimer, puis de les redéployer.

La vitesse de démarrage d'un conteneur soulage aussi les temps de déploiements et de tests.

Ici, c'est essentiellement un travail porté sur Ansible qui a été le coeur du projet. Que ce soit l'installation du cluster ou le déploiement de services, ainsi que la gestion des volumes persistants ou des liens inter-services, tout est géré via des playbook Ansible.

19. ou sandbox

Les déploiements de clusters et de services sont exécutés par des jobs Jenkins développés en Job DSL [20].

OpenShift

OpenShift est une plateforme PaaS développée par RedHat, qui propose aussi la solution en mode SaaS. Le projet est open source et se nomme ***Openshift Origin*** .

Openshift Origin est fortement attaché à trois briques :

- **Docker** en tant que gestionnaire de conteneur applicatif
- **Kubernetes** pour l'orchestration des applications et la gestion de cluster
- **Atomic** qui permet le démarrage rapide de projets, de gérer des hôtes légers dont les services sont démarrés sous Docker, un gestionnaire de paquets, en bref c'est un projet à part entière [21].

Dans le principe, les applications tournent toutes dans des conteneurs Docker rassemblés en "Pods" délégués à ***Kubernetes*** . Ce dernier va orchestrer les déploiements de réplicas dans le cluster.

20. `https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin`
21. Pour plus d'information sur Atomic : http ://www.projectatomic.io/docs/

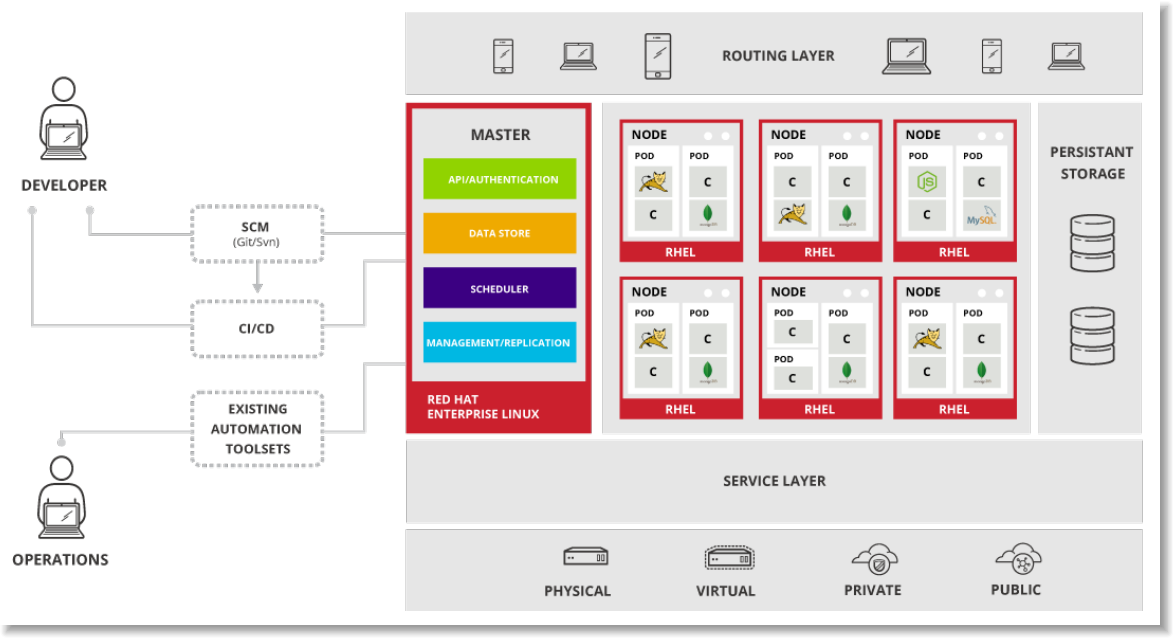


Figure *3.20 – Infrastructure Openshift*

OpenShift permet de raccorder des services à la plateforme afin de déployer des conteneurs applicatifs via Kubernetes dans l'infrastructure cluster. Par exemple, un "push" sur une branche Git et/ou un nouveau tag va engendrer la mise à jour de l'application dans l'infrastructure de serveurs en utilisant par ailleurs la capacité de Kubernetes de déployer des nouvelles versions en cascade [22].

OpenShift Origin propose une API REST, une interface graphique Web et des plugins.

22. Kubernetes peut en effet utiliser un type de réplication nommé "Deployment" à cet effet

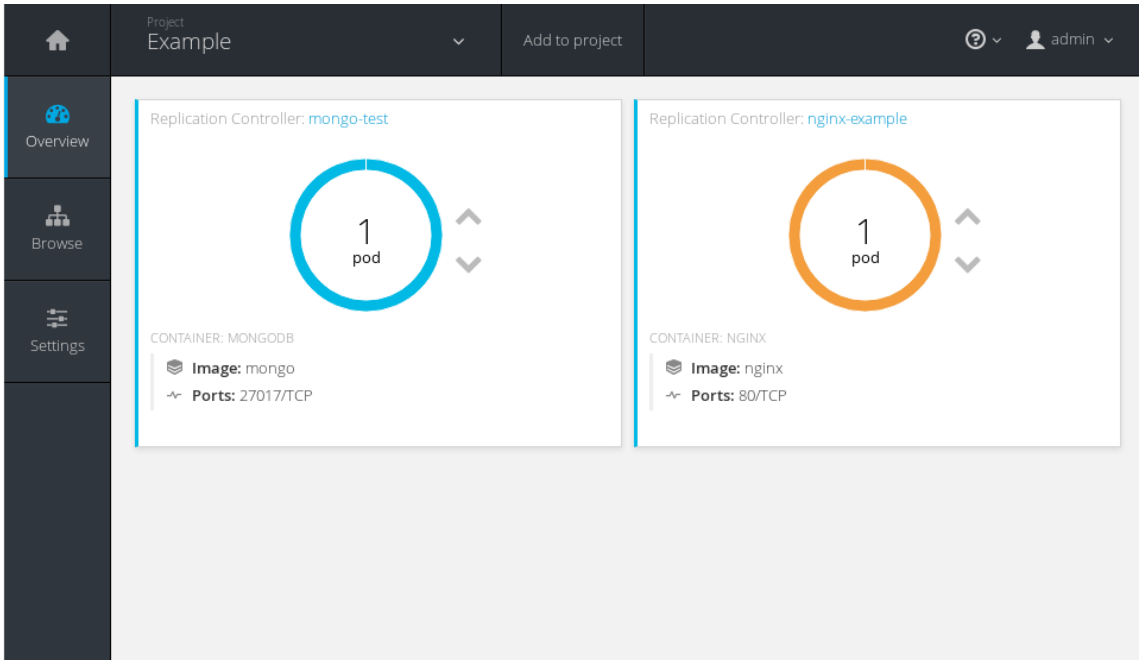


Figure *3.21 – L'état des controlleurs et "Pods" vu depuis l'espace projet dans l'interface web*

Cette interface graphique est certainement l'un des points forts de OpenShift. Elle offre la possibilité de créer et de gérer des projets complets qui ont leurs propres quotas, leurs propres droits, et permet aussi de travailler sur les conteneurs via une console directement sur la page web.

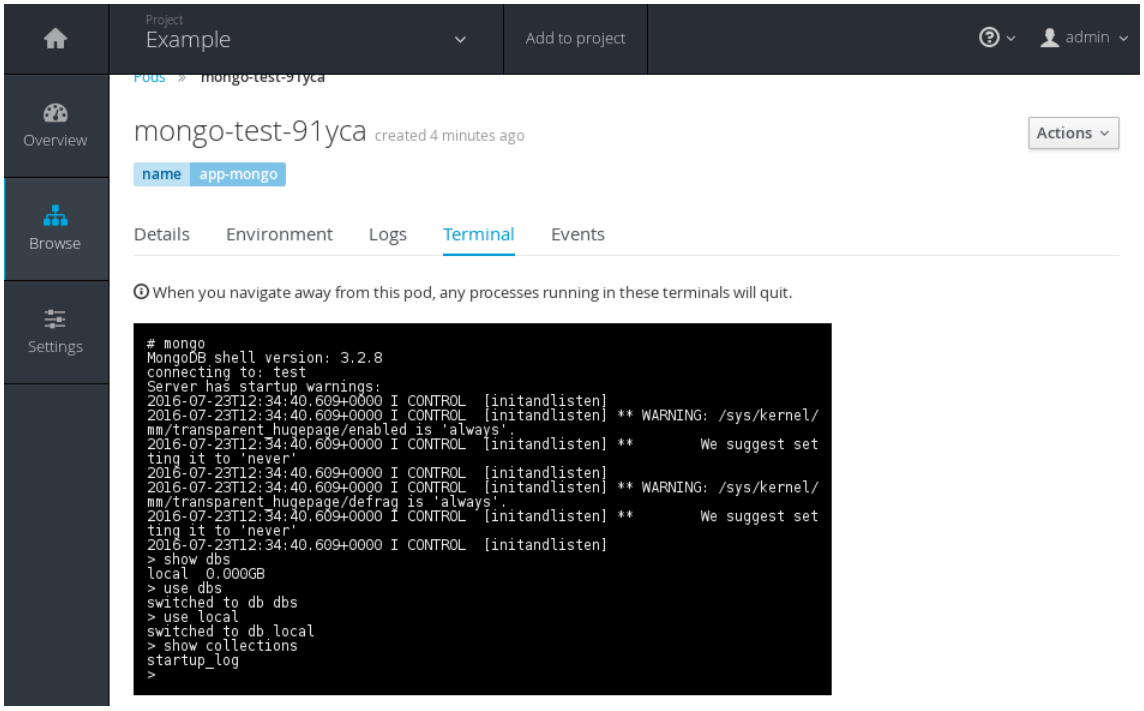


Figure *3.22 – OpenShift donne accès à une console sur l'interface Web*

En outre, une console de "debugage" est accessible lorsqu'un conteneur ne démarre pas correctement.

Autre intérêt solide de OpenShift : il prend en charge la construction des images Docker du projet en gérant son registre interne . De ce fait, Kubernetes va utiliser les versions d'images déployées dans le registre et abstrait cette gestion qui est souvent faite en amont.

OpenShift est donc bien plus orienté "développeurs" car il soulage les concepts de construction d'images, de gestion de registre, et facilite l'accès aux conteneurs via l'interface web. Mais en contre-partie, il faudra un paramétrage plus complexe, moins malléable que s'il on souhaite gérer ces parties via Jenkins, GoCD, ou tout autre outil externe.

Le choix de OpenShift est donc souvent déterminant. Sa puissance et sa capacité à simplifier le travail de gestion de projet peut aussi bien être un avantage qu'un inconvénient selon la chaîne de procédures que vous souhaitez mettre en place.

C'est donc l'espace "clé en main" de Openshift Origin qui peut charmer vos équipes DevOps.

OpenShift Origin est outil facile à intégrer dans un environnement de livraison continue de part ses aptitudes à se greffer dans la chaîne de validation. Il repose sur Kubernetes qui bénéficie d'une excellente réputation et d'un support très important. Son interface est accessible à des non initiés et permet de contrôler les applications dans le cluster sans forcément paramétrer des outils clients ce qui lui confère une ergonomie agréable pour le développeur.

CoreOS

CoreOS est une distribution Linux légère incluant les services fleet, etcd, flanneld et se basant sur le principe de systemd. Une de ses particularités est de ne pas proposer de gestionnaire de paquets.

Docker est installé par défaut sur la plateforme mais CoreOS propose aussi son propre système de conteneur nommé "Rockit" ou "rkt".

Le principe est de créer un service via un fichier "unit" qui sera injecté dans le cluster avec l'outil "fleetctl", le service "fleetd" va alors déployer ces fichiers de déclaration de service dans les noeuds souhaités.

Fleet est un service communiquant avec ***etcd*** qui contrôle ***SystemD*** pour installer et démarrer des services. Ces services démarrent des conteneurs et peuvent correspondre à leur tour avec Etcd.

Les fichiers "unit" sont alors installés par SystemD, ces derniers vont simplement démarrer un conteneur Docker ou Rockit et servir le logiciel souhaité.

Par exemple :

```
[Unit]
Description=Starting MongoDB Service

[Service]
ExecStartPre=-/usr/bin/docker stop DB
ExecStartPre=-/usr/bin/docker rm DB
ExecStart=/usr/bin/docker run --name DB -p 27017:27017 mongo
ExecStartPost=/usr/bin/etcd set /databases/DB "is up"
```

```
[X-Fleet]
Global=true
```

Ce service démarre un conteneur sur tous les noeuds du cluster CoreOS, simplement en passant ce fichier en paramètre à "fleetctl".

La logique d'orchestration est directement déclarée dans les fichiers units. L'utilisation de "etcd", un serveur clé/valeur haute disponibilité et très rapide va servir de bus de message pour proposer un partage de configuration, ou pour synchroniser des tâches entre les services.

> CoreOS est une solution de cluster légère et adaptée pour les administrateurs désireux de contrôler finement la réplication et l'orchestration. Mais il est clairement une "base" sur lequel il est recommandé d'installer Kubernetes ou un autre gestionnaire de cluster du moment où l'orchestration s'avère plus large. CoreOS recommande d'ailleurs de n'utiliser Fleet qu'à des fins d'installation d'un orchestrateur de plus haut niveau [a]
>
> a. `https://github.com/coreos/fleet/blob/master/README.md` : This project is quite low-level, and is designed as a foundation for higher order orchestration

Exemples d'implémentation : Cluster Nuxeo via CoreOS

Un de nos clients dessert plusieurs sites via la plateforme Nuxeo. Chaque site est une instance isolée. Les équipes de développement et intégration utilisent ***docker-compose*** afin de servir localement l'application, tester, intégrer et délivrent finalement une ***image docker*** dans un registre privé.

Une des problématiques était de permettre à notre client de pouvoir créer des nouveaux sites Nuxeo à volonté et à part entière sans avoir à configurer manuellement le déploiement.

Nous avons donc créé un cluster CoreOS de 5 machines permettant de supporter les différents sites. Tous les nœuds ont été raccordés à un service de stockage ***GlusterFS*** où se trouve des templates de services ainsi que les définitions docker-compose au format YAML.

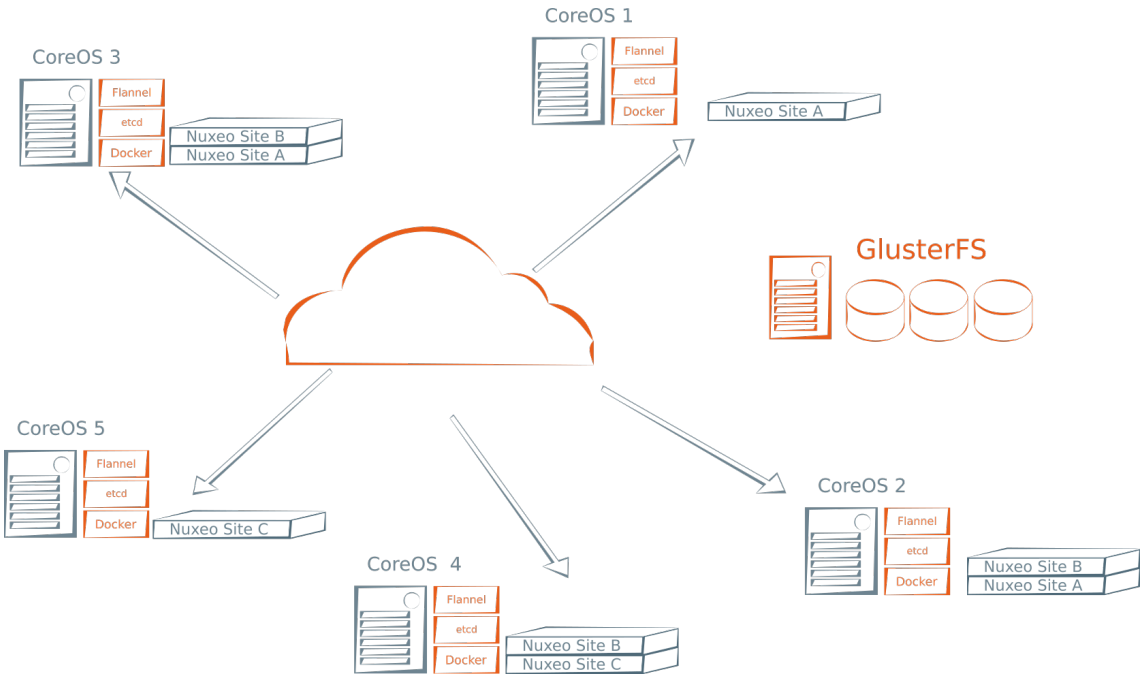


Figure *3.23 – Implémentation CoreOS Nuxeo et GlusterFS*

Quand un nouveau site doit être ajouté, l'utilisation d'un "unit file" SystemD template était alors capable d'intégrer un fichier d'environnement qui était interpolé ensuite dans le fichier "docker-compose.yaml" spécifique à tous les sites Nuxeo. Une instance est alors démarrée en tant que pile de conteneurs Docker sur un ou N nœud CoreOS en fonction de la charge estimée.

> Les affinités de session et les contraintes d'installation sont réduits grâce au paramétrage fin des services ***SystemD*** et ***docker-compose*** . Quant à ***Fleet*** , il permet de régler très finement le déploiement sur les noeuds du cluster

Conclusion

Force est de constater que les outils ne manquent pas pour la mise en place de méthodologie DevOps en Continuous Delivery. Vous l'avez compris, il est possible d'adapter les solutions pour répondre aux contraintes métiers, aux exigences des équipes, à leurs compétences et aux types de produits à gérer.

Mais il est aussi important de prendre en considération l'infrastructure à mettre en place ainsi que la formation des différents acteurs de la chaîne "CD". Se jeter à corps perdu dans la livraison continue peut tout aussi bien être compliqué et réduire vos chances de réussite si vous n'êtes pas accompagnés et informés sur les tenants techniques et méthodologiques qui vont s'imposer.

Cela étant dit, la livraison en continu, si elle est maîtrisée, fluidifie les procédures et soulage fortement toutes les équipes ; la correction et maintenance laissant place à la qualité et l'évolutivité.

En effet, les retours que nous avons sont parlants : le nombre de demandes faites aux administrateurs système a largement réduit, et les développeurs ont bien plus de maîtrise sur la continuité *post-dev*. La communication s'est clarifiée avec les chefs de produits, les actions sont plus courtes et les résultats sont visibles quasiment instantanément.

Nous avons aussi remarqué qu'une partie sensible se trouve en bout de chaîne ; c'est effectivement la production de services (la partie serveurs) qui demande une attention particulière car elle peut induire une modification ou adaptation de l'infrastructure, bien souvent causée par l'utilisation de conteneurs.

Toutefois, il est bien souvent possible de réduire l'impact de cette intégration et d'adapter l'existant ou de s'adapter à l'existant.

Les réussites obtenues grâce à cette démarche, ces outils, sont nombreuses, et constituent l'évolution technique aujourd'hui nécessaire dans la digitalisation des entreprises.

Pour cela, vous pouvez sélectionner un ou plusieurs outils présentés ici, et assembler votre propre chaîne de livraison continue, sachant qu'il n'y a pas une solution meilleure que les autres, cela dépend de votre contexte, de votre besoin.

Smile intervient sur tout ou partie de la chaîne CD, et nous pouvons vous accompagner sur cette voie.