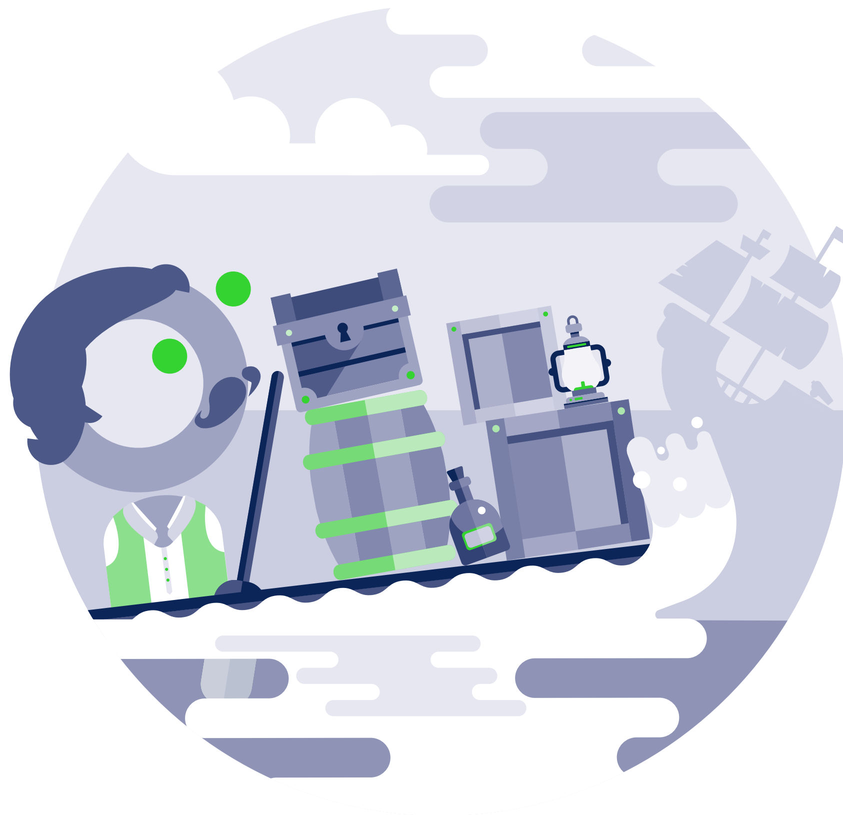
OCTO Technology

There is a better way

CULTURE DEVOPS

VOL. 02



GUIDE DE SURVIE DANS LA JUNGLE TECHNOLOGIQUE

Les innovations technologiques en rupture pour nourrir notre technophilie

Nous assumons pleinement notre boulimie technologique et tentons de la nourrir chaque jour pour rendre notre travail toujours plus passionnant. Nous sommes convaincus qu'il existe un réel amour de la technologie, pour la technologie et le potentiel d'innovation qu'elle offre. Sa mise en œuvre dans des situations ambitieuses constitue des défis stimulants que nous aimons relever.

INTRODUCTION À LA TRILOGIE

Les trois histoires de DevOps : Organisation, Qualité et Technologie

Si vous êtes resté caché dans une caverne ces huit dernières années, vous êtes peut-être passé à côté de la mouvance DevOps. Aujourd'hui, c'est un fait : le mot est sur toutes les lèvres de notre milieu IT. Mais le terme est souvent galvaudé, chacun y allant de sa définition, et il devient urgent de converger sur un concept partagé pour éviter que DevOps ne demeure qu'un *buzzword* de plus. Ainsi, il est de notre devoir d'apporter une pierre à l'édifice de cette convergence en vous livrant, en toute humilité, notre vision[1] de DevOps : celle qui nous passionne et qui berce notre quotidien.

Nous définissons DevOps comme un ensemble de pratiques qui visent à réduire le *Time to Market*[2] et à améliorer la qualité des produits logiciels, en réinventant la coopération entre DEV et OPS. DevOps, c'est un modèle d'organisation, une culture, un assemblage de processus, d'outils et de *patterns*[3] d'architecture. Pour nous, c'est aussi un métier, une passion. Et c'est la nature même de cette passion que nous désirons vous partager ici.

Nous avons pour habitude de regrouper ces pratiques DevOps en quatre piliers :

1. **Culture, méthodes et organisation :** on y retrouve les ingrédients secrets d'une organisation IT équitable et durable.
2. ***Infrastructure as Code* et les pratiques de qualité qui l'accompagnent :** ce pilier détaille les techniques d'automatisation du déploiement des applications et des infrastructures (réseaux, stockages, serveurs). Il vise à appliquer au monde des opérations informatiques, des outils et des pratiques issus du monde de l'ingénierie logicielle (gestion de versions du code source, tests automatisés, répétabilité des opérations, etc.)
3. ***Patterns* d'architecture *Cloud Ready Apps*[4] :** on regroupe ici les modèles d'architectures techniques qui permettent de tirer parti du *Cloud Computing* pour offrir davantage de résilience, de sécurité, d'exploitabilité et de scalabilité[5] aux applications.

4. **Processus d'intégration et de déploiement continus :** dans ce pilier, on industrialise la production logicielle et son processus de validation de la qualité. En se reposant sur l'usine d'intégration et de déploiement continus (CI/CD[6]), on automatise toutes les étapes de la chaîne de production : tests fonctionnels, tests de performance, tests de sécurité, tests de recette, déploiement de l'application sur un environnement, etc.

Quatre piliers donc, avec le choix éditorial de les traiter en trois volumes. C'est bien à vous que cette trilogie s'adresse, vous qui êtes informaticien, développeur, administrateur système, architecte, responsable des études ou de la production, CTO, CIO, CDO, CEO, CxO, DevOps amateur ou confirmé, ou juste curieux. Vous y trouverez notre vision, certes optimiste, de cette discipline sous trois angles complémentaires et indissociables :

- La culture et les modèles d'organisation équitables,
- La passion pour les technologies ouvertes,
- L'amour du travail de qualité et le *software craftsmanship*[7] appliqué à DevOps.

Voici donc le second volet de la trilogie. Il présentera des traits d'ADN ou des technologies qui nous sont chers et améliorent notre quotidien lorsque l'on pratique DevOps. Nous terminerons en identifiant quelques tendances. Qu'elles soient latentes sous forme de signaux faibles ou vraisemblablement présentes dans notre futur proche, elles s'annoncent comme des puissants vecteurs de transformation de nos métiers.

Asseyez-vous confortablement et commencez la dégustation par le chapitre qui vous plaira.

---

① Notre vision est celle d'OCTO Technology et de sa Tribu OPS qui œuvre depuis plus de cinq ans à la transformation DevOps de belles et grandes entreprises de France et de Navarre.

② *Time to Market* est le temps moyen écoulé entre l'émergence d'une idée et sa commercialisation. Dans cet ouvrage, on utilisera cette notion dans un sens plus large de *Time to Value* : le temps écoulé entre l'idée et sa mise à disposition pour l'utilisateur (dans notre contexte, il s'agira du temps de mise à disposition d'une fonctionnalité logicielle).

③ Un *pattern* est une solution répétable et réutilisable pour un problème identifié.

④ Voir notre publication sur ces architectures *Cloud Ready Apps* disponible sur notre site octo.com.

⑤ "La scalabilité (*scalability*, en anglais) désigne la capacité d'un logiciel à s'adapter à la montée en charge, en particulier sa capacité à maintenir ses fonctionnalités et ses performances en cas de forte demande" (source Wikipedia).

⑥ CI/CD : *Continuous Integration / Continuous Delivery*.

⑦ Voir le manifeste *Software Craftsmanship* : http://manifesto.softwarecraftsmanship.org/

Guide de survie

01 **Introduction** 09

02 **Une API REST sinon rien** 10-17
- Un peu d'histoire 12
- La promesse des API pour les infrastructures 13
- Une API est un produit avant tout 14
- L'APIsation des infrastructures est encore en cours 15
- cURL + SDK + CLI : le trio gagnant pour utiliser les API ? 16
- Conclusion 17

03 **L'open source** 18-31
- Des outils déjà présents en entreprise 20
- Des traits d'ADN séduisants 21
- Un modèle de collaboration hors du commun 23
- Tout n'est pas rose pour autant 26
- Et côté DevOps ? 28
- Conclusion 29

04 **Les architectures distribuées : la magie des clusters applicatifs** 32-43
- Problématique 34
- Promesses et principes 37
- Impacts sur les infrastructures 39
- Principes de fonctionnement 40
- Risques 42
- Conclusion 43

05 **L'orientation SDx** 44-63
- Contexte 46
- Nouveaux termes pour nouvelles offres 47
- Conclusion 49

06 L’Infrastructure as Code 64-81
Introduction 66
Historique 67
Approche déclarative vs. impérative 68
Différentes familles d’Infrastructure as Code (IaC) 71
Abstraction 74
Composition 75
Nos chouchous du moment 76
Conclusion 81

07 La conteneurisation 82-99
Introduction 84
La conteneurisation en quelques lignes 85
En pratique : comment emprisonne-t-on un processus ? 87
Docker : ces petits plus qui font toute la différence 89
Pourquoi cette technologie est-elle disruptive ? 93
La situation à date 97
Conclusion 99

08 Le cloud 100-109
Introduction 102
"Élève en progrès, mais peut mieux faire" 104

09 Et à part ça ? 110-123
Crypto everywhere 112
L’IA partout ? 114
La conception responsable 118

10 Conclusion 124

Introduction

Construire des systèmes d'information puissants, évolués, robustes n'a jamais été aussi passionnant. La technologie évolue sans cesse et rend ce qui était hier impossible tout à fait accessible et réalisable aujourd'hui. De nouvelles idées apparaissent tous les jours et nous sommes en permanence face à de nouvelles technologies qui nous "décollent la pulpe du fond".

Voyons de quelle manière certaines innovations ou caractéristiques technologiques sont suffisamment impactantes pour **changer en profondeur la façon dont les entreprises pensent, construisent ou consomment des capacités d'infrastructure.** La promesse de ces nouveaux paradigmes ? Rendre les infrastructures moins chères, plus rapides à approvisionner, plus performantes, plus résilientes, mieux maîtrisées – même à très grande échelle –, plus simples, plus standard et interconnectables, plus propices aux changements (d'architecture, de taille).

Si des innovations interviennent encore du côté du matériel, c'est bien l'adage "*software is eating the world*" qui révolutionne l'usage qui est fait désormais des infrastructures. Il est par conséquent logique de retrouver, d'année en année, toujours plus de concepts issus du monde du logiciel prendre leur place dans les *datacenters*.

Software is eating the hardware

Réservée initialement aux grandes entreprises, l'infrastructure est aujourd'hui accessible à tous. N'importe quelle start-up peut, en moins de temps qu'il ne lui faut pour se faire un café, démarrer un serveur. Une journée de "location" de ce serveur lui coûtera d'ailleurs à peu près le même prix qu'un expresso à emporter.

Les limitations inhérentes à la possession d'une infrastructure physique sont toujours les mêmes : chères, encombrantes, immuables. Mais, désormais, ce n'est plus votre problème. Il existe des solutions. Ce "*new deal*" permet à de nouveaux acteurs d'atteindre une agilité (et une élasticité) jusqu'ici impossible. À chaque technologie son porte-étendard : Netflix pour le *cloud*, Spotify pour les micro-services, Amazon Web Services pour le *Software Defined Hardware*…

S'il ne faut pas perdre de vue que ces innovations ont toutes été popularisées par un usage réel et économiquement viable, nous vous proposons ici un **voyage au cœur de ces révolutions**, à travers nos yeux de passionnés. Au fil de cet ouvrage, nous vous présenterons quelques-unes de ces technologies qui sont, pour nous, au cœur de la transformation de nos sociétés.

Une API REST sinon rien

Si les API ne sont pas un phénomène récent, c'est leur généralisation qui a profondément transformé notre façon d'exposer et de consommer des services. C'est notamment le cas dans le monde des infrastructures.

Tomato
SOUP
TUNA

Un peu d'histoire

On a quelque temps pensé que les portails Web étaient le Saint Graal pour créer des ressources d'infrastructure (machines virtuelles en premier lieu). La multiplication de projets de type "Portail de création de Machines Virtuelles en Libre Service" en est un des vestiges les plus poignants. Même s'ils sont toujours présents, ces projets ne représentent aujourd'hui qu'un usage tout à fait marginal.

Le changement de paradigme vient d'où on ne l'attend pas, lorsqu'en 2002, Jeff Bezos a proposé l'approche suivante de l'architecture interne d'Amazon :

1. *All teams will henceforth expose their data and functionality through service interfaces.*

2. *Teams must communicate with each other through these interfaces.*

3. *There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.*

4. *It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols— doesn't matter.*

5. *All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.*

6. *Anyone who doesn't do this will be fired.*

7. *Thank you; have a nice day!*

Lorsque 4 ans plus tard AWS (Amazon Web Services) voit le jour, le modèle reste applicable pour des services offerts aux utilisateurs. La bombe est lancée, **l'infrastructure de demain sera pilotée par une API ou ne sera pas.**

La promesse des API pour les infrastructures

Des API implémentées devant les services d'infrastructure offrent des possibilités infinies en termes d'**automatisation** : approvisionnement, destruction, configuration, monitoring, administration. Il devient alors possible de piloter à distance, via des systèmes tiers, le cycle de vie complet de pans entiers du système d'information. De nombreux outils de management, de déploiement, ou d'*Infrastructure as Code*[8] savent s'interfacer avec des API de services d'infrastructure (les *clouds* publics en premier lieu) pour en automatiser la gestion de façon **programmatique.**

Finies les installations de machines ou de logiciels à la main, *exit* les wizards d'installation (next => next => next => ok) à cliquer manuellement sur toute une série d'écrans pour paramétrer un système.

[8] Nous aurons l'occasion de reparler très largement de cette thématique dans le chapitre éponyme *"Infrastructure as Code"* plus loin dans cet ouvrage.

Une API est un produit avant tout

Le rôle des API aujourd'hui dépasse très largement une simple problématique technique. Mettre en œuvre ses propres API, pour des services d'infrastructure ou non, conduit à adopter **l'approche *API as a product.*** Son cycle de vie est alors régi comme un réel produit en termes de facilité d'usage, de technologie, de stratégie de versionning, de priorisation du besoin, de pilotage, de réalisation…

API ne rime donc pas pour autant avec *open bar.* Une API REST se conçoit comme tout service : avec des garde-fous. Nous pensons par exemple à l'authentification (*authentication* ou *authn*), l'autorisation (ou *authz*), la limitation de taux d'appels (*rate limit*), la comptabilité des appels (*accounting*)...

L'APIsation des infrastructures est encore en cours

L'APIsation – si l'on peut se permettre le néologisme – se généralise progressivement à tout le SI. Cela devient un véritable confort de savoir qu'un service est simplement accessible en REST. Il en va donc de même avec les services d'infrastructures. Le *cloud* a montré la voie avec une politique du "tout API".

L'importance des API est telle qu'elles sont désormais, pour nous, un critère incontournable dans la sélection des solutions et des partenaires. En effet, lorsqu'il faut s'intégrer avec un fournisseur de service en SaaS, la première chose que nous devons évaluer, c'est précisément la qualité de son API : cohérence des ressources exposées, respect des standards, documentation, performance, disponibilité. **Un bon produit avec une mauvaise API devient automatiquement un mauvais produit.**

"Envoie-moi un exemple de cURL, je comprendrai comment utiliser cette ressource "

Nous vous invitons à creuser le sujet en consultant les nombreuses publications OCTO sur l'art d'une **bonne stratégie d'API**, notamment à travers nos *Refcards*[9].

Ce n'est pas un hasard si aujourd'hui, la plupart des produits d'infrastructure modernes (*open source* ou non) mettent en avant la qualité de leur API, mesurable via un "*Time To First API Call*[10]" bas. Les API très mal conçues existent malheureusement dans bon nombre de produits et peuvent rendre leur intégration très laborieuse. Généralement, elles ne respectent pas l'orientation ressource, l'utilisation des verbes HTTP ou les codes de retour standard. Certaines, derrière une apparente innocence peuvent cacher un comportement erratique, non répétable, lent, instable et peu loquace en cas d'erreur. Un calvaire.

⑨ Retrouvez toutes nos *Refcards* sur notre site octo.com à la rubrique Publications ⑩ Le TTFAC (*Time To First API Call*) désigne le temps nécessaire à un développeur pour réaliser un 1er appel d'API, après avoir consulté la documentation.

cURL + SDK + CLI : le trio gagnant pour utiliser les API ?

Avoir une bonne API s'accompagne donc très avantageusement de 3 outils :

- **cURL** : bas niveau, mais primordial pour comprendre unitairement les appels d'API : positionnement des en-têtes (*Headers*), de la charge utile (*payload*). C'est l'outil de base à utiliser quand plus rien ne fonctionne. cURL peut devenir assez laborieux à manier quand il faut mêler l'authentification (basique ou via un jeton *bearer*) à une structure de données complexe.

- **SDK** pour *Software Development Kit* : disponible pour votre langage de programmation favori, il permet de coder des applications utilisant nativement vos API, et ce, avec peu d'effort.

- **CLI** pour *Command Line Interface*. Le client, généralement basé sur le SDK, permet d'utiliser simplement l'API dans un terminal pour effectuer des opérations manuelles. D'après quelques légendes urbaines, certains se risqueraient même à utiliser cette solution pour des tâches d'administration récurrentes dans des *scripts shells*…

Si c'est à votre tour d'exposer une API pour fournir un nouveau service à valeur ajoutée, posez-vous les questions suivantes :

- Qui sont les consommateurs de mon API (internes, externes, authentifiés ou non, combien sont-ils, etc.) ?

- Comment vont-ils s'en servir et comment leur simplifier la tâche ?

- Quel est le SLA de mon API (performances, disponibilités) ?

- Quelles sont mes ressources, quels sont leurs cycles de vie ?

- À quelle stratégie d'entreprise ma nouvelle API s'intègre-t-elle (*gateway, accounting, billing*) ?

Conclusion

LES RAISONS QUI NOUS FONT AIMER LES API	LES LIMITES DES API REST
• Exposition programmatique systématique • Standardisation indiscutable des interfaces techniques (HTTP REST)	• Performances par rapport à certains protocoles (gRPC par exemple est une alternative très crédible pour du *machine-to-machine*)

CE QUI FAIT QUE CETTE TECHNOLOGIE TRANSFORME NOTRE MÉTIER

- Penser *API first* change les priorités dans la mise en place de services d'infrastructure
- L'API devient un nouveau critère de choix de solutions d'infrastructure
- Exposer des API devient un nouveau métier côté infrastructure, plus important que l'exposition d'interfaces Web ou le traitement de tickets
- L'API va permettre aux utilisateurs de créer leurs propres fonctionnalités par dessus l'infrastructure, démultipliant sa richesse

L'Open Source

L'utilisation de l'open source en entreprise est un sujet qui déchaîne toujours les passions tant du côté de ses soutiens que du côté de ses détracteurs. Nous ne prétendons pas traiter ici l'intégralité de la question ou la pertinence du logiciel libre en entreprise. Elle a été abordée et continue à l'être depuis des décennies dans de nombreux ouvrages. Tentons simplement de voir en quoi son usage peut avoir des impacts, bénéfiques ou néfastes, sur les pratiques DevOps.

git
de survie

Des outils déjà présents en entreprise

De notre expérience, rares sont les entreprises qui n'utilisent pas du tout de logiciels libres. Nous avons toujours fini par trouver, caché dans un coin, au moins un outil *open source* Certains le cachent comme s'il s'agissait d'une maladie inavouable. "Vous pensez bien, du logiciel gratuit, trouvé sur Internet...". Même si théoriquement, "libre" ne veut pas dire "gratuit", dans les faits les logiciels libres rencontrés le sont le plus souvent. Au palmarès des grands classiques, voici quelques exemples parmi ceux que l'on rencontre régulièrement :

- Système d'exploitation, en premier lieu GNU/Linux, tant sur des serveurs que sur les périphériques équipés d'Android.
- Serveurs Web : Apache httpd, NGINX
- Reverse-proxies / load-balancers / gestionnaires de caches : HAProxy, Varnish
- Serveurs d'application : Apache Tomcat, JBoss WildFly
- Services d'infrastructure: bind (DNS), ntpd (NTP), DHCP
- Langages : Python, Ruby, NodeJS
- Outils de développement : Git, Atom, Eclipse
- Solutions d'intégration continue : Jenkins, Gitlab-CI, Concourse, Drone
- SGBD : MariaDB, PostgreSQL
- Bases NoSQL : Cassandra, MongoDB, Redis, CouchBase, Elasticsearch
- Solutions " BigData" : la plateforme Hadoop (HDFS, Apache Kafka, Apache ZooKeeper, Spark, Storm etc.)
- Monitoring & Supervision : Nagios, Zabbix, Prometheus, InfluxDB, Grafana, Elasticsearch, Logstash, Kibana
- Outils d'infrastructure : Ansible, Puppet, Terraform

La liste se rallonge très aisément dès que l'on y réfléchit. Et devient obsolète le temps de le faire… Inutile de chercher à la rendre exhaustive, c'est peine perdue. Disons simplement que certains de ces logiciels libres font office de référence dans leur domaine et que leur prise en charge dans un contexte DevOps est nécessaire.

Des traits d'ADN séduisants

Sans tomber dans des généralités, la plupart des logiciels libres partagent des caractéristiques qui leur valent un certain succès et qui les rendent tout à fait éligibles à des pratiques DevOps.

Approche *KISS*[11]

Les logiciels libres sont généralement des produits simples, plus simples que leurs équivalents éditeur. On peut leur reprocher d'être trop simples, mais cette philosophie, appelée aussi la philosophie UNIX, est plutôt appréciée : **ils ne font qu'une chose, mais ils le font bien.** Ils n'annoncent les fonctionnalités que lorsqu'elles sont réellement présentes. **C'est un modèle que les éditeurs ont du mal à adopter.** Ils sont généralement friands de l'approche "étoile noire" : faire un produit qui répond à un besoin et très vite lui ajouter autant de fonctions que possible pour phagocyter le reste du SI du client. Ce pattern de *vendor lock-in*, très efficace, s'accompagne souvent d'une stratégie d'annonce toute aussi séduisante : "Ne vous inquiétez pas, ça sera dans la prochaine version".

Habituellement sous UNIX/Linux, un logiciel serveur (démon) est géré à travers 3 éléments classiques (PFS) :

- **P** : un **P**aquet (RPM, deb…) permettant le packaging et l'installation du logiciel
- **F** : un ou des **F**ichier(s) texte permettant de configurer son comportement
- **S** : un **S**ervice (au sens UNIX du terme) qui gère le démarrage et l'arrêt du logiciel

Le modèle PFS implique généralement le **respect des conventions du système hôte** (fichiers de configuration dans `/etc`, fichiers UNIT, données dans `/var/lib`[12], etc.), la disponibilité des paquets sur des dépôts publics... Entre la pression de la communauté et la capacité à simplement proposer une amélioration si nécessaire, les logiciels libres respectent souvent ces conventions ce qui facilite grandement **l'automatisation** et **l'exploitabilité du logiciel**.

⑪ "KISS : Keep It Simple, Stupid (en français, mot à mot : "garde ça simple, idiot", dans le sens de "ne complique pas les choses"), ou principe KISS, est une ligne directrice de conception qui préconise la simplicité dans la conception et que toute complexité non indispensable devrait être évitée dans toute la mesure du possible". (source : Wikipedia) ⑫ Conformément au FHS : https://fr.Wikipedia.org/wiki/Filesystem_Hierarchy_Standard

L'intégration de plusieurs logiciels simples sera peut-être nécessaire pour accomplir la tâche que l'on souhaite réaliser. Mais nous préférons cette approche, propice au **travail itératif**, plutôt que d'utiliser un logiciel complexe imposant une manière de travailler inadaptée.

Ouverture naturelle

Les logiciels libres sont généralement pensés pour s'intégrer. Ce sont d'ailleurs souvent les premières implémentations de référence de normes et de protocoles. Ça tombe bien, car si DevOps implique une contrainte sur les logiciels utilisés, c'est bien leur ouverture et leur **capacité à s'interopérer**. Il en va de même de la **portabilité** et de la **versatilité**. En étant écrits dans des langages largement répandus dans le logiciel libre et en utilisant des librairies également très standards (POSIX[13] en étant le meilleur exemple), ils prouvent leur capacité à être utilisés dans des contextes très différents. Par opposition, le manque d'interopérabilité d'une bonne part des logiciels propriétaires du marché tend à tordre les modèles d'intégration au SI. Nous pensons par exemple à l'utilisation détournée d'un ESB pour faire la glue entre deux produits non intégrables.

Les licences comme GPL, LGPL, BSD, Apache ou Mozilla garantissent que les logiciels concernés vont pouvoir être installés sans limitation du nombre de copies. C'est un réel confort dans un modèle DevOps où l'on souhaite pouvoir passer à l'échelle, gagner en réactivité, être en capacité de déployer des environnements, des machines supplémentaires sans impacter le *Time to Market* avec la gestion des licences. Finis les soucis de renouvellement de licence, de déblocage de budget pour anticiper l'ajout d'un serveur, d'une instance ou d'un nouvel environnement. L'économie de la gestion du serveur de licence associé est un bienfait supplémentaire, surtout quand il s'agit en plus de le mettre en haute-disponibilité. La dimension de plus en plus distribuée des architectures informatiques, avec sa logique de **scalabilité horizontale**, n'aurait pas été possible – pour des raisons économiques – en s'appuyant sur des logiciels propriétaires. Contraints par des logiques de coûts de licence, les DSI devaient mutualiser les machines, les OS, les applications. Les succès du logiciel libre sont donc aussi le reflet de la levée de cette contrainte économique, aussi bien pour les DSI que pour les acteurs du *cloud* qui ont construit leur plateforme sans logiciel propriétaire.

L'ouverture est également sensible en termes d'**accès à la documentation** : manuels, bases de connaissances. Des sites comme *Stackoverflow* nous rappellent régulièrement que nous sommes rarement les premiers à être tombés sur un obscur message d'erreur, ou à avoir besoin d'une configuration un peu particulière d'un logiciel libre.

[13] https://fr.Wikipedia.org/wiki/POSIX

Un modèle de collaboration hors du commun

Accès aux sources

Regarder les sources d'un logiciel est un confort dont nous profitons régulièrement. Lorsque plus rien ne se passe comme prévu, que l'on est arrivé à la limite de ce que les commandes *strace* ou *tcpdump* peuvent nous montrer, il est primordial de pouvoir regarder le code. Un DevOps se sentira toujours moins démuni s'il peut regarder les sources du logiciel défaillant, même si elles sont en *Perl.*

Par ailleurs, les capacités de recompilation et d'auditabilité du code restent une qualité que les organismes proches des sujets de sécurité ou de défense apprécient.

Pouvoir suivre la vie du projet

Au-delà du simple accès aux sources, c'est toute **la surveillance du modèle de collaboration** qui est un indicateur de la vie d'un projet. La page GitHub d'un dépôt est une mine de renseignements, d'un simple coup d'œil (voir schéma page suivante).

Voici quelques-uns des indicateurs que nous aimons regarder sur les projets *open source* :

- L'activité du dépôt sur les dernières semaines ou années : le projet est-il toujours vivant ? Y-a-t'il une augmentation, une baisse de l'activité de contribution ? Quel est son cycle de *release* ?
- Le nombre de contributeurs est-il important ? Baisse-t-il ? Augmente-t-il ?
- Quelles sont les entreprises contributrices ? Il est toujours très intéressant de voir l'énergie que les entreprises investissent sur les projets. En particulier cela permet de déterminer si une seule entreprise produit le plus gros effort de développement, si des entreprises se désintéressent ou, au contraire, commencent à contribuer. Il n'est pas rare de constater que des entreprises concurrentes parviennent même à produire des évolutions sur les mêmes logiciels.

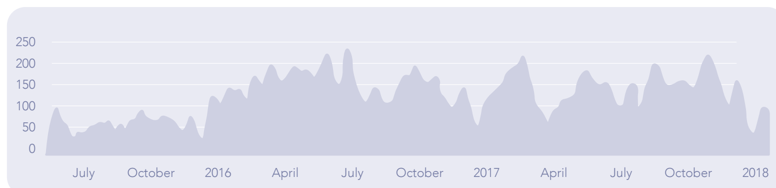
À travers ces éléments, nous sommes capables de mesurer **la sélection naturelle qui s'applique sur les projets**. Il s'agit là d'éléments de décisions importants.

kubernetes / kubernetes

Graphs

Contributors

① Jun 1, 2015 - Jan 20, 2018
Contributors to master, excluding merge commits

②


③

robinsoncrusoe
972 commits / 676,378++ / 56,153 --

vendredisursonîle
928 commits / 428 ++ / 277,204 --

① Âge du projet ② Information sur la dynamique de contribution au projet

③ Permet d'identifier les personnes et entreprises contributrices

Un coup de pouce de fournisseurs de services en ligne

Les fournisseurs de solutions de développement en mode SaaS fournissent un effort de soutien auprès des projets *open source*. Parmi les solutions les plus utilisées, citons :

- Des dépôts de code source gratuits
- Des gestionnaires de projet (wiki, suivi de demandes et de bugs, suivi de *pull requests*…)
- De l'outillage d'intégration continue avec tests automatiques
- Des services d'analyse de couverture de code
- De la génération de documentation

Etat de l'intégration du projet	Lien sur la documentation de référence du logiciel	Couverture des tests unitaires du projet
queue | running	godoc | reference	coverage | 84 %

Ces outils, à très forte valeur ajoutée, participent à améliorer la qualité du produit. Les projets qui disposent de tests unitaires, et dont les résultats des cycles d'intégration sont publics, nous rassurent sur la qualité et les risques de régression. Tous les logiciels propriétaires développés dans les entreprises ne peuvent pas se targuer d'un tel outillage, et encore moins d'une telle **transparence sur leur qualité.**

Généralisation du modèle de *pull requests*

Les projets *open source* ont majoritairement généralisé l'usage de Git comme gestionnaire de source. Un modèle de collaboration a émergé par la suite : *fork*[14] d'un dépôt, ajout de modifications dans une branche dédiée, proposition du correctif sous forme d'une *pull request*[15]. Les avantages de ce modèle sont nombreux : transparence des modifications, capacités à discuter de leur contenu, système d'approbation par un collège de relecteurs. Le modèle de contributions par *pull requests* encourage aussi quiconque ayant accès au dépôt à contribuer.

Devant ce succès, certaines entreprises sont en train d'adopter ce modèle collaboratif pour leurs dépôts internes de code source, conscientes que la qualité s'en voit tirée vers le haut. C'est en effet un **puissant levier de transformation** qui met l'accent sur la **responsabilité partagée du code**, plutôt que sur l'affectation des tâches à des individus par un tiers. Nous y voyons un bon moyen d'aider la transition des équipes : les anciens relisent le code des nouveaux qui gagnent progressivement en autonomie.

[14] Duplication d'une base de code permettant d'en réaliser une évolution différente de celle du code original. Plus d'infos sur Wikipédia : https://fr.Wikipedia.org/wiki/Fork_(d%C3%A9veloppement_logiciel) [15] Proposition de modification du code d'origine soumise auprès de ses mainteneurs, permettant d'en discuter ouvertement le contenu et de le modifier avant intégration à la base de code.

Tout n'est pas rose pour autant

Les qualités et avantages cités précédemment ne rendent pas systématiquement tous les logiciels libres exempts de reproches. C'est pourquoi le travail d'analyse de chaque projet (activité de la communauté, modèle de contribution, outillage, qualité de la documentation...) reste absolument nécessaire.

Certains problèmes récurrents persistent et doivent être traités avec attention.

Intégration avec les systèmes d'exploitation

Le packaging des logiciels libres est généralement un travail effectué séparément par deux types de contributeurs :

- **Le développeur du logiciel**, qui tente de fournir la dernière version dans les plus brefs délais. Avant tout concentré sur la fourniture de nouvelles fonctionnalités, il n'a pas toujours la capacité de s'assurer de la simplicité d'installation de son logiciel sur un nombre important de systèmes d'exploitation.

- **Le mainteneur de paquet d'une distribution**[16], qui a pour objectif d'assurer la cohérence du logiciel avec les paquets de la distribution. Cette recherche de cohérence l'amène parfois à faire preuve d'inertie et du fait d'incompatibilité de versions. Il ne peut pas toujours proposer la dernière version du logiciel.

Devant la divergence d'objectifs de ces deux contributeurs, il faut régulièrement faire un choix :

- Utiliser la version de la distribution, un peu ancienne mais simple à installer

[16] La notion de distribution est très liée au monde GNU/Linux : la variété des composants permettant de construire l'OS et la totale liberté laissée dans la manière de les intégrer ont fait émerger plusieurs grands courants. Chacun propose des choix orientés et cohérents dans la sélection des composants logiciels, du cycle de livraison, du format des paquets logiciels, de la gestion des services du système etc. Et chaque ensemble forme donc une distribution. Pour approfondir le sujet : https://fr.Wikipedia.org/wiki/Distribution_Linux

- Utiliser la dernière version créée par le développeur et payer le prix en termes de repackaging et de chasse aux bugs potentiels.

Un risque à piloter

Utiliser des logiciels *open source*, c'est aussi s'assurer de leur pérennité sur la durée. Or, après les premiers jours ou premières semaines d'utilisation de logiciel libre, il faut se rendre à l'évidence : au même titre qu'un logiciel éditeur, il est fondamentalement difficile de suivre l'évolution du produit. Que ce soit pour des mises à jour fonctionnelles ou pour corriger des failles de sécurité, **le travail d'installation et de mise à jour ne s'arrête jamais**. Et faire vivre les logiciels, libres ou non, demande de l'énergie.

Les entreprises sont par conséquent contraintes de faire un choix stratégique dans leur approche de l'*open source* :

Plan A : capitalisation sur des compétences internes

Certaines entreprises font le choix d'assumer cet effort de guerre et misent sur **quelques profils d'un haut niveau d'expertise** capables de faire le suivi des versions et des menaces de sécurité, le packaging de l'application, l'adaptation des fichiers de configuration, le redéploiement et bien entendu les tests. L'avantage d'une telle approche : une capacité à choisir très finement les logiciels et les versions qui sont utilisés, indépendamment d'une *roadmap* éditeur contraignante, avec une très forte autonomie, et sans avoir besoin de faire appel à des compétences externes. Ce sont ces entreprises qui finissent par maîtriser leur sujet au point de devenir contributrices aux projets *open source* dont elles n'étaient initialement qu'utilisatrices. Améliorer le produit dont on a besoin : la boucle est bouclée, le modèle de contribution fonctionne.

Plan B : souscription d'une offre de support "entreprise" de l'éditeur

En choisissant l'option "éditeur" de l'*open source* – ce qui peut paraître paradoxal – les entreprises tentent de **bénéficier du meilleur des deux mondes** : des garanties de résolution de problèmes contractuelles sur le produit et la force du logiciel libre notamment sur son prix "par machine". L'éditeur, en échange de cette rétribution, réalise le travail de veille et de packaging et contribue au projet *open source* pour le faire évoluer sur le long terme.

Toutefois, souvent, les contrats de support de produits *open source* ont des conséquences :

- Utilisation possible exclusivement de certaines versions sur certains systèmes d'exploitation eux-mêmes supportés. Cette restriction d'usage peut s'avérer très contraignante : version supportée trop vieille, inadaptée à l'OS standard de l'entreprise.

- Certains éditeurs réalisent les correctifs avec un circuit de validation tellement lent qu'en cas de faille de sécurité, les systèmes peuvent rester exposés longtemps, très longtemps.

Et côté DevOps ?

Même s'il n'existe pas à proprement parler d'outils "DevOps", les outils génériques d'industrialisation et d'automatisation que nous manipulons quotidiennement sont majoritairement *open source* :

- Les outils de développement dont nous avons déjà parlé précédemment
- Certains outils d'intégration continue (Jenkins, Gitlab-CI…)
- Des outils d' *Infrastructure as Code*[17] (Ansible, Puppet, Chef, CFEngine, Terraform...)
- Des orchestrateurs de ressources d'infrastructure (Docker, Kubernetes, Mesos...)

Nous apprécions ces outils pour les mêmes raisons : simplicité, ouverture, inter-opérabilité, souplesse.

Est-ce à dire que tous les outils DevOps sont *open source* ? Bien sûr que non. Les éditeurs ont rapidement adapté leurs offres pour **les rendre compatibles "DevOps"**, et chacun des acteurs historiques arrivera à vous convaincre (ou en tout cas essaiera) que sa solution apporte le meilleur ROI possible dans la transformation DevOps de votre entreprise. Sans nécessairement ranger tous les logiciels éditeur dans la même catégorie, disons qu'il n'est pas toujours facile de trouver toute la souplesse qu'un ou plusieurs logiciels libres bien intégrés pourront apporter : versatilité, gestion des versions, extensibilité, accès aux sources (pour déboguer). Au même titre que les autres logiciels éditeurs, les logiciels de déploiement d'automatisation et autres pâtissent des mêmes reproches. Avant de les choisir, posez-vous systématiquement les questions suivantes :

- La gestion du logiciel est-elle 100 % automatisable ?
- Est-il livré avec des recettes de déploiements automatiques (Puppet, Ansible ou autre) ?
- Est-il livré avec un Dockerfile ?
- Est-il disponible sous forme d'image Docker sur le hub public ?
- Est-il possible de gérer 100 % de la configuration du logiciel dans un système de gestion de version (ex. Git) ?
- Le logiciel peut-il s'étendre et s'intégrer avec d'autres outils, libres ou non ?
- N'existe-t-il pas une alternative plus économique ? Plus répandue ?
- Quelle est la qualité de la documentation ? Est-elle librement consultable en ligne ?

[17] Certaines personnes emploient encore parfois le terme d'outils de gestion de configuration.

Conclusion

LES RAISONS QUI NOUS FONT AIMER L'*OPEN SOURCE*

- Maturité et qualité des produits
- Innovation constante
- Ouverture
- Inter-opérabilité
- Transparence
- Modularité
- Esprit KISS

LES LIMITES DU MODÈLE

- Besoin de gérer la compétence *open source*
- Travail d'intégration de plusieurs outils parfois nécessaire
- La communauté n'acceptera pas une évolution qui est trop spécifique à un besoin d'une entreprise

CE QUI FAIT QUE CETTE TECHNOLOGIE TRANSFORME NOTRE MÉTIER

Nous ne sommes plus de simples utilisateurs d'une solution, mais également des contributeurs en capacité de l'améliorer.

Exemple de contribution à Terraform

À la croisée des chemins entre l'*open source* et le *cloud*

Il arrive de se sentir frustré par les fonctionnalités présentes dans un outil. Au travers de ce récit, montrons comment nous avons refusé cette fatalité et participé à l'amélioration du logiciel Terraform.

Qu'est-ce que Terraform ?

Terraform est un outil développé par Hashicorp qui permet de piloter le déploiement de son infrastructure pour un fournisseur cible (AWS, Azure, Openstack…) dans un langage de description (Hashicorp Common Language).

```
resource "aws_instance" "web" {
  ami = "${data.aws_ami.ubuntu.id}"
  instance_type = "t2.micro"
  tags {
    Name = "HelloWorld"
  }
}
```

Exemple de création d'une instance sur AWS

Il conserve un état de l'infrastructure déployée pour pouvoir identifier les changements entre le code et l'état de l'infrastructure déployée à partir de ce code (*pattern Resource* : description de l'état cible).

Afin de bénéficier de la puissance et des spécificités de chaque fournisseur (et non du plus petit dénominateur commun), chaque ressource créée dans Terraform ne peut fonctionner que pour un fournisseur.

Quelles étaient les limites ?

Dans la version 0.7.0 de Terraform, un nouveau type de primitives a été introduit : les *Data Sources*. Il permet d'aller chercher des objets déjà instanciés (par exemple des réseaux), via des éléments de recherche comme des tags ou un nom. Il est ensuite possible d'utiliser ces objets et les différentes informations

de ces objets (par exemple l'ID du réseau) pour instancier d'autres objets.

Pour une de nos missions, nous avons utilisé Terraform pour gérer notre infrastructure AWS. Nous avions une gestion par tags de notre infrastructure et avons souhaité bénéficier de la puissance des *Data Sources* sur un maximum d'objets. La primitive étant très récente, tous les objets AWS principaux n'étaient pas implémentés sous la forme de *Data Sources*. C'est donc au travers de l'ajout de *Data Sources* manquants que nous avons contribué au code de Terraform. Ces contributions ont pris la forme de trois *Pull Requests* sur la fin de l'année 2016.

Comment faire la contribution ?

Terraform est un projet *open source* hébergé sur Github et développé en Go. Nous nous sommes appropriés le code et avons ajouté les fonctionnalités manquantes (nous en avons profité pour apprendre à faire du Go dans un projet complet).

Nous avons ensuite suivi le processus classique d'une contribution : *Pull Request* sur le projet, application des corrections suite aux commentaires en retour (soit sur des compléments de fonctionnalités, soit sur des problèmes autour du code en lui-même).

Sur ce projet, comme dans de nombreux autres, l'ajout de fonctionnalités s'accompagne de l'écriture de nouveaux tests et de la documentation associée. La contribution, en termes de nombre de lignes, se décompose grossièrement ainsi : 1/3 de "vrai" code (en Go), 1/3 de tests (en Go) et 1/3 de documentation (en *Markdown*).

Les *Pull Requests* :

https://github.com/hashicorp/terraform/pull/9604

https://github.com/hashicorp/terraform/pull/9766

https://github.com/hashicorp/terraform/pull/10301

Les architectures distribuées : la magie des clusters applicatifs

Les architectures ont évolué pour traiter certaines problématiques (scalabilité, résilience) au niveau des ***couches applicatives****, alors qu'historiquement traitées par les couches de l'infrastructure. Même s'il peut paraître anodin, ce changement de paradigme n'est pas toujours bien appréhendé tant il déplace les frontières. Prenons quelques minutes pour détailler ce phénomène.*

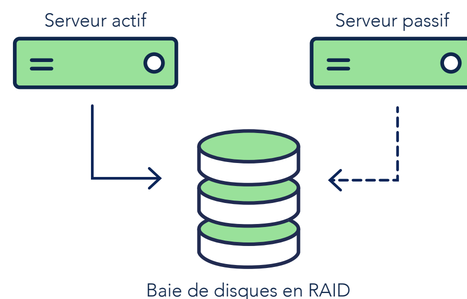
Problématique

Pour illustrer notre propos, prenons l'épineux exemple de la **gestion de la durabilité d'une base de données**. Pour rappel, la définition Wikipedia de la durabilité est la suivante :

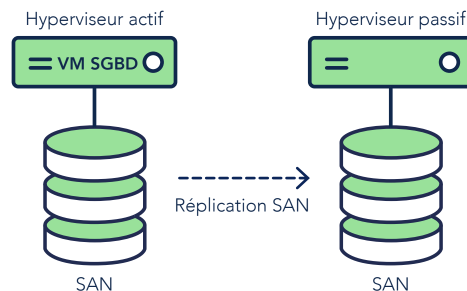
> "Dans le contexte des bases de données, la durabilité est la propriété qui garantit qu'une transaction informatique qui a été confirmée survit de façon permanente, quels que soient les problèmes rencontrés par la base de données ou le système informatique où cette transaction a été traitée. Par exemple, dans un système de réservation de sièges d'avion, la durabilité assure qu'une réservation confirmée restera enregistrée quels que soient les problèmes rencontrés par l'ordinateur qui gère le système de réservation (panne d'électricité, écrasement de la tête sur le disque dur, etc.)."

Plusieurs stratégies peuvent être adoptées pour atteindre cet objectif :

1. Utiliser une **baie de disques en RAID attachée à un serveur.** Si celui-ci défaille, débrancher la baie puis la rebrancher sur un second serveur.



2. Disposer de **deux SAN avec une réplication baie à baie** en place et deux hyperviseurs qui y sont attachés. La VM peut être déplacée en cas de besoin.



3. Mettre en place une **réplication entre deux volumes** au niveau du système d'exploitation de deux serveurs (DRBD par exemple).

SGBD Actif
O.S.
Réplication blocs
SGBD Éteint
O.S.
Disque local
Disque local

4. Le SGBD dispose d'un système de **réplication applicative** (*Log shipping* par exemple) qu'il s'agit de mettre en œuvre entre deux serveurs avec des disques internes.

SGBD Maître
O.S.
Réplication journaux de transactions SQL
SGBD Esclave
O.S.
Disque local
Disque local

Les entreprises qui se voulaient à l'état de l'art ont longtemps opté pour **l'approche "ceinture + bretelle"** avec un mélange des options 2 et 3 ou 2 et 4. Les coûts sont à l'image de la complexité de la mise en œuvre, mais la **promesse de résilience** – même en cas de grosse panne – est tenue. Seule ombre au tableau, une utilisation de mécanismes de haute disponibilité en mode actif/passif, ce qui n'est pas optimum pour la disponibilité : les bascules sont rarement transparentes.

Il est facile de se rendre compte qu'en fonction de la stratégie choisie, il va y avoir des différences majeures :

- En termes de qualité de service, dans les SLO [18] obtenus (RTO[19] et RPO[20]), la résilience à la perte d'un disque, d'un serveur, d'une baie, d'une salle, d'un centre de calcul
- Dans les compétences à solliciter pour mettre en œuvre la stratégie choisie (Expert stockage ? Expert hyperviseur ? Expert système ? DBA ?)
- Dans les opérations d'administration à appliquer en cas de désastre pour un retour à un fonctionnement nominal
- Dans les options à disposition pour augmenter les capacités de la base de données (en termes de volumétrie et du nombre d'entrées/sorties).

⑱ "Les *service level objectives* (SLOs), ou "objectifs de niveau de services" en français, sont un élément clé de l'accord de niveau de services (*service level agreement*, SLA) entre un fournisseur de service et un client. Les SLOs sont considérés comme un moyen de mesurer la (les) performance(s) du fournisseur de service et sont aussi là pour éviter des disputes entre les deux parties, basées sur un malentendu." (source : Wikipedia) ⑲ "Le *recovery time objective* (RTO), ou durée maximale d'interruption admissible (DMIA) est l'expression de besoin de disponibilité des différents métiers ou services, dans une organisation. Le terme RTO apparaît dans le document Special Publication 800-34 du NIST de juin 2002."(source : Wikipedia) ⑳ "Le *recovery point objective* (RPO), ou perte de données maximale admissible (PDMA), quantifie les données qu'un système d'information peut être amené à perdre par suite d'un incident. Usuellement, elle exprime une durée entre l'incident provoquant la perte de données et la date la plus récente des données qui seront utilisées en remplacement des données perdues" (source : Wikipedia)

Or, nos besoins en performance (volumétrie, entrées/sorties) augmentent et nos exigences en disponibilité ne diminuent pas (*24/7*, *follow the sun*...). Pour atteindre ces deux objectifs, **il faut arrêter de penser scalabilité verticale** (*scale-up*) et **pivoter à 90° vers la scalabilité horizontale** (*scale-out*). Sans ce pivot, les coûts deviennent assez vite déraisonnables et la faisabilité technique compromise.

Promesses et principes

Systèmes naturellement distribués et décentralisés. *Exit* le mono-machine. Ces solutions prennent tout leur sens sur plusieurs machines, plusieurs baies, plusieurs salles, plusieurs *datacenters*. La haute disponibilité du service est native, en mode actif/actif/actif[21]. La réplication de la donnée est un des principes de base de ces systèmes. Il n'y a plus une machine centrale qui décide pour les autres.

Systèmes naturellement scalables (*scale out*). L'ajout de capacités (machines avec du CPU, de la mémoire et du disque) augmente mécaniquement les capacités de stockage et/ou de traitement.

Design for failure. Les applications n'attendent presque plus rien des infrastructures de traitement et de stockage en termes de haute disponibilité (pas de RAID sur les disques, par exemple). Les serveurs sont banalisés. Ils peuvent et vont tomber en panne, c'est une certitude. C'est donc à l'application de garantir la copie des données via des mécanismes de réplication logicielle. De même, le réseau est faillible, parfois congestionné, source de pertes de paquets. C'est le travail de l'application de vivre malgré ces aléas.

Topology-awareness. Les applications sont conscientes de la topologie physique sous-jacente (quelle machine est dans quelle baie, quelle baie est dans quelle salle, quelle salle est dans quel centre de calcul). Elles adaptent en conséquence leur stratégie de réplication suivant la politique paramétrée (performance ou durabilité).

Banalisation du réseau. Pas de technologie réseau dédiée au stockage comme le FibreChannel, tout est sur IP. L'agrégation de liens permet d'assurer des débits maximums sur les interfaces.

Parmi les outils adoptant cette philosophie, plusieurs familles se détachent :

- **Les Nouvelles Architectures de Données.** Parfois regroupées derrière le terme NoSQL, ces solutions exposent des interfaces de type requêtes qui peuvent faire penser à nos SGBDR historiques : insertion, mises à jour, sélection... Quelques solutions du marché : Cassandra, Elasticsearch, MongoDB, Couchbase, Riak.

[21] Actif/actif/actif : évolution d'un déploiement actif/actif où les composants actifs du service sont répartis sur au moins trois *datacenters*, ce qui permet la perte de n'importe quel *datacenter* sans perte de service. Ce mode de déploiement est incontournable pour atteindre des SLO élevés.

- **Les solutions de stockage objet** (*object storage*). Plutôt orientées sur des accès type "fichiers", ces solutions reprennent dans leur ADN tous ces principes d'architecture (partitionnement, réplication…). Voici quelques produits du marché : Ceph, Openstack Swift, Scality...

- **Les systèmes de gestion de clusters applicatifs.** Centrés sur l'exécution de services distribués, ils ont un orchestrateur qui répartit au mieux les demandes de lancement de travaux en fonction de règles de placement et de l'occupation du cluster. Des exemples : Mesos, Kubernetes...

- **L'écosystème Hadoop.** À la fois porté sur la persistance de la donnée avec HDFS, Hadoop propose également un moteur d'exécution distribué du doux nom de YARN. Sa particularité : une co-localisation des traitements et des données pour minimiser les transferts de données.

Impacts sur les infrastructures

La mise en œuvre des solutions classiques de haute disponibilité couplées à du *scaling* principalement vertical impacte lourdement certains choix qui ont été faits sur des infrastructures stockage et serveur *best of breed*[22] : elles sont devenues trop riches fonctionnellement, trop chères. Bref, trop haut de gamme.

À quoi me sert un SAN connecté à des hyperviseurs, managé avec toute une suite de logiciels, le tout hors de prix, si mon cluster applicatif sait très bien se contenter de serveurs à bas coût sans même nécessiter de résilience sur ses disques ?

"Pas question de négocier sur la bande passante et la latence entre des machines d'un même rack "

Le réseau, au contraire, reste un organe essentiel. Pas question de négocier sur la bande passante et la latence entre des machines d'un même *rack*, elles doivent pouvoir synchroniser énormément de données et maintenir leur état cohérent.

[22] Ce qui se fait de mieux et de plus abouti dans le domaine.

Principes de fonctionnement

Les architectures distribuées partagent entre elles quelques traits d'ADN qui peuvent avoir des impacts non négligeables lors de leur mise en œuvre.

En premier lieu, ces architectures implémentent **des mécanismes d'élection de nœuds maîtres par une majorité des nœuds** (le *quorum*[23]) pour certaines fonctions critiques, ainsi qu'un **annuaire clé-valeur**. Pour ce faire, il est nécessaire de déployer des nœuds en nombre impair pour des raisons de quorum, généralement 3, 5, ou 7. Un cluster *Zookeeper* par exemple, fonctionne convenablement avec 3 nœuds au minimum. Comptez 5 instances si vous voulez pouvoir effectuer des opérations de maintenance sur un nœud tout en acceptant la défaillance d'un autre nœud en même temps. Même dans cette situation, le quorum de 3 nœuds opérationnels est respecté et les écritures possibles "durablement". L'impact pour maximiser la disponibilité de la solution dépend alors de la répartition de ces machines sur plusieurs baies, plusieurs salles, plusieurs *datacenters*. Un cluster réparti correctement sur 3 sites fortement interconnectés fonctionne nativement en mode actif/actif/actif, est capable de résister à la perte complète d'un site.

En second lieu, pour tirer le maximum de profit de ces technologies, il est nécessaire de **partitionner la donnée**. La littérature anglophone parle de *sharding*. La notion de partitionnement renvoie à l'idée de répartir l'ensemble des enregistrements d'une table (au sens d'une base de données) sur plusieurs machines. Ainsi, on choisira par exemple de stocker les clients de A à M sur une machine #1 et les clients de N à Z sur une autre machine #2. Le *sharding* horizontal nécessite une clé de répartition – la première lettre du nom dans cet exemple. Derrière ce modèle simple se cachent des questions structurantes, comme le choix de la clé de partitionnement qui dépend de la donnée à stocker, en vue de

[23] Nombre de nœuds minimum nécessaire pour prendre certaines décisions au sein du cluster, comme élire un nouveau nœud maître, ou accepter les écritures (Cf *Split Brain*). [24] "Les propriétés ACID (atomicité, cohérence, isolation et durabilité) sont un ensemble de propriétés qui garantissent qu'une transaction informatique est exécutée de façon fiable." (source : Wikipedia) [25] "Le modèle de cohérence à terme (ou *eventual consistency*) requiert que les écritures d'un processus soient vues par les autres, mais sans date limite ni contrainte d'ordre ; et que lorsque les mises à jour s'arrêtent, tous les processus observent le même état. C'est le modèle de cohérence le plus faible utilisé en pratique." (source : Wikipedia).
Pour aller plus loin http://www.allthingsdistributed.com/2007/12/eventually_consistent.html

trouver la meilleure distribution des clés sur les différentes partitions. Par exemple, le choix de la répartition selon la première lettre du nom s'avère souvent un mauvais choix car elle ne répartit pas les données de manière équilibrée.

Le *sharding* nécessite également d'adapter les architectures applicatives :

- Les **requêtes** doivent être adaptées pour tenir compte de la distribution, notamment en minimisant toute requête *inter-shard* car le coût d'accès à plusieurs nœuds distants est prohibitif. Les API de ces systèmes limitent ainsi les possibilités de jointure et d'agrégats à des données situées sur le même *shard*.

- Certaines fonctionnalités de **modélisation** sont complexes voire impossibles à implémenter nativement dans le cluster (exemple : les clés étrangères). L'implémentation d'une telle logique se doit alors de remonter côté applicatif, lors du développement.

- L'**atomicité** (le **A** de **ACID**[24]) est souvent limitée à la granularité de niveau clé (donc à la valeur/document) afin d'éviter des mises à jour atomiques incluant plusieurs clés, voire plusieurs *shards* et donc des transactions coûteuses en performance, ce qui est d'autant plus vrai si elles sont distribuées sur plusieurs machines.

- Le modèle dit de **"cohérence à terme"** permet des états transitoires pendant lesquels tous les nœuds n'ont pas nécessairement la même vision de la donnée. Une fois les réplications effectuées, le système finira par converger, mais entre temps, des lectures sur des réplicats peuvent retourner des informations obsolètes. Il existe plusieurs techniques pour contourner ce problème : toujours demander au nœud maître la donnée du *shard* concerné, au même possesseur d'un *shard* donné[25], jouer sur les variables de *quorum*, ou attendre "un certain temps".

Nos experts du sujet des Nouvelles Architectures de Données (NAD) chez OCTO se feront un plaisir de creuser le sujet si vous êtes intéressés, notre blog les mentionne très régulièrement. C'est donc naturellement que nous les avons intégrés dans nos réflexions sur les pratiques DevOps et que nous avons appris à en tirer le meilleur parti.

Risques

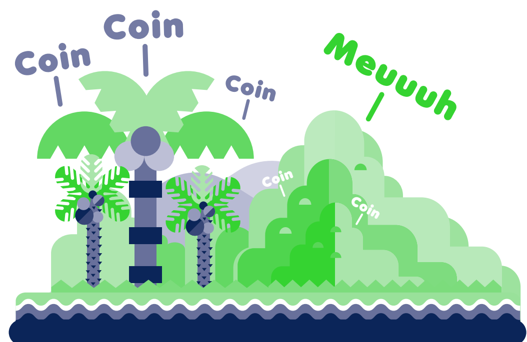
Les systèmes distribués sont donc extrêmement puissants et comme bien souvent, cela ne se fait pas sans contrepartie. Nous sommes face à des logiciels qui sont, par construction, complexes.

L'architecture est généralement composée de plusieurs rôles de machines qu'il faut identifier, dimensionner, agencer, connecter via différents réseaux, fournir en stockage.

Le caractère **auto-resilient** (*self-healing*) de ces architectures masque une certaine partie de la complexité puisque par construction, le système tolère la panne. Mais ce sont également ces comportements automatiques voire "automagiques" qui rendent les clusters plus complexes à administrer : **changements de leader** à cause de ré-élections spontanées, **rééquilibrage des données** à des moments mal choisis, etc. Il n'est pas toujours aisé de suivre l'état du cluster. De la même façon, il n'est pas aisé de savoir vraiment l'état d'un cluster en observant un seul nœud.

Par conséquent, les travaux d'architecture, de déploiement et d'administration sont des opérations généralement non triviales. Chaque technologie, au-delà de principes de fonctionnement identiques, demande un **haut degré d'expertise et d'expérience** pour être parfaitement maîtrisée. La mise en œuvre d'une telle technologie ne doit jamais se faire à la légère.

En gérant applicativement la résilience, les *middlewares* ne s'appuient plus sur les services d'infrastructure. Chaque outil va faire des choix d'implémentation spécifiques à son contexte pour optimiser le plus possible la distribution et la répartition de la donnée. Cette liberté acquise par ces *middlewares* présente tout de même une contrainte : les utilisateurs et exploitants de ces outils devront connaître, pour chacun d'entre eux, les choix qui s'offrent à eux et comment en tirer le meilleur parti.



Conclusion

De grands pouvoirs et donc de grandes responsabilités

LES RAISONS QUI NOUS FONT AIMER CETTE TECHNOLOGIE

- Des promesses très ambitieuses de tolérance aux pannes et de scalabilité
- Une certaine "automagie" qui traite de nombreuses problématiques complexes qui n'ont pas à être résolues manuellement

LES LIMITES DE CETTE TECHNOLOGIE

- Une modélisation du partitionnement de la donnée qui doit dépendre de son type et de son usage, donc finalement des besoins métiers
- Une certaine complexité d'administration
- Un risque d'avoir des clusters qui deviennent incontrôlables
- Vouloir gérer la disponibilité sur plusieurs couches à la fois (infra, hyperviseurs, système et applicatif) conduit irrémédiablement à des comportements erratiques

CE QUI FAIT QUE CETTE TECHNOLOGIE TRANSFORME NOTRE MÉTIER

- Utilisation des *datacenters* possible en mode actif/actif/actif. Un grand pas vers le PCA[26]
- Besoin accru de plusieurs *datacenters* (les traditionnels 2 ne suffisent pas pour un quorum)
- Moins d'exigences sur les SLO unitaires des machines (banalisation, commoditisation)
- Utilisation accrue du réseau

[26] "**PCA** : en informatique, un **P**lan de **C**ontinuité d' **A**ctivité, a pour but de garantir la survie de l'entreprise après un sinistre important touchant le système informatique." (source : Wikipedia)

L'orientation SDx

L'appellation SDx (Software Defined X) pourrait se traduire par le néologisme français ***logiciélisation des infrastructures.*** *L'émergence de ces technologies est le fruit d'un double constat :*

1. Les infrastructures sont de plus en plus dépendantes du logiciel. *Le matériel n'est plus rien sans son firmware, quel que soit le nom qui lui est donné. Le matériel vendu est souvent capable de tout faire, et seules des options logicielles (payantes ou non), viennent "activer" de nouvelles fonctions.*

2. La présence du logiciel libère des contraintes techniques*. Une reconfiguration n'est plus nécessairement liée à un changement physique sur un équipement. Elle devient théoriquement simple, rapide, réversible, agile.*

Contexte

La tendance a été initiée il y a longtemps, côté réseau, avec l'apparition des VLANs, sur le stockage avec les SANs et côté serveurs avec les hyperviseurs. La création et l'attribution de capacités (réseaux, disques, CPU et mémoire) n'imposent plus de réaliser un branchement, un câblage, ou un brassage la plupart du temps. Une fois ces pré-requis apparus, les offres sont restées propriétaires et fermées pendant de longues années. Leur usage est resté très traditionnel comme si l'affectation d'une ressource demandait encore de descendre en salle serveur, opération chronophage s'il en est. L'habitude a la peau dure et, même si créer des VLANs demande techniquement quelques secondes, leurs modes d'utilisation restent très statiques.

Le *cloud*, à nouveau, vient semer le trouble avec de nouveaux enjeux :

- **Beaucoup plus d'objets**. Là où la limite de 4096 VLANs est finalement assez rapidement atteinte, de nouveaux protocoles réseau, comme VxLAN par exemple, proposent de repousser la limite à 2^24 (plus de 16 millions) réseaux logiques.

- **Durée de vie des objets plus courte.** Il est tout à fait classique de créer une VM pour 1 heure, de lui attribuer 500 Go de stockage et de la placer dans un réseau spécifique avec des règles de filtrage ad-hoc.

- **Des API devant tous les services**. Il n'est plus optionnel de proposer une API. C'est même le premier moyen d'utiliser les ressources du *cloud*, loin devant les portails Web. La standardisation des API, leur facilité d'utilisation et la qualité de leur documentation deviennent des critères qui peuvent faire le succès ou l'abandon d'une offre.

- **Ouverture des services**. Dans la suite logique de l'approche API, de plus en plus d'intégration entre les services sont attendues : approvisionnement, gestion des événements, monitoring, facturation. Des systèmes fermés ne sont donc plus la réponse. Il devient facile d'ajouter de la logique propre à son entreprise dans l'allocation, l'autorisation, ou la facturation des ressources.

- **Banalisation des infrastructures**. Également appelée commoditisation, cette évolution privilégie le matériel de très grande série, à bas coût, standardisé, facilement remplaçable et réversible d'un constructeur à l'autre. La disponibilité ou le MTBF[27] d'une machine ne devient plus le premier critère de choix[28].

㉗"*Mean Time Between Failures* : Le temps moyen entre pannes ou durée moyenne entre pannes, souvent désigné par son sigle anglais le MTBF est une des valeurs qui indiquent la fiabilité d'un composant d'un produit ou d'un système." (source : Wikipedia) ㉘Paradoxalement, les grands acteurs du cloud comme Google manipulent tellement de serveurs qu'ils peuvent se permettre de produire leur propre matériel (parfois très spécialisé) en lieu et place du *commodity hardware* d'autres constructeurs. Cette recherche d'une optimisation économique n'a pas nécessairement pour but d'améliorer massivement le MTBF.

Nouveaux termes pour nouvelles offres

Même si les technologies de virtualisation *on-premise* (réseau, stockage, machine) sont donc historiquement capables de s'abstraire des réalités physiques, les nouveaux usages offrent un défi que toutes ne parviennent pas à relever.

C'est donc une nouvelle terminologie de solutions qui émerge depuis quelques années pour afficher clairement cette volonté de refondation par rapport aux offres historiques.

Parmi les termes les plus classiquement rencontrés dans le domaine des technologies SDx, nous trouvons :

"Les nouveaux usages offrent un défi que toutes les technologies de virtualisation on- premise ne parviennent pas à relever "

• **SDN (*Software-Defined Network*)** qui décrit un modèle d'intégration et de virtualisation réseau accessible de façon programmatique. La beauté de la démarche du SDN réside dans sa capacité à adresser à la fois les piles réseau des hyperviseurs et les équipements réseau physiques (commutateurs) en les pilotant via des protocoles comme OpenFlow. Ces composants – logiques ou physiques, peu importe – deviennent les bras armés (le *data plane*) d'un cerveau logiciel (le *control plane*) qui orchestre dynamiquement la reconfiguration complète du réseau en fonction de la vie des projets, des applications, des utilisateurs…

• **SDS (*Software-Defined Storage*)** qui propose des solutions de stockage dites *scale-out*, nativement scalables, résilientes, sachant tirer profit de serveurs à bas coût sans disque en RAID. La gestion de la performance, de la résilience (n copies ou du pseudo-RAID 5 logiciel dit *erasure-coding*) est traitée au niveau applicatif.

• **SDC (*Software-Defined Compute*)** ou **SDS (*Software-Defined Server*) :** simplement basé sur les technologies *on-premise* de virtualisation, où certains acteurs ont ajouté à leur offre historique une API standard pour ouvrir leur écosystème.

Une intégration d'une solution *open source* regroupant SDC, SDS et SDN, basée massivement sur OpenStack[29] pourrait ressembler à ceci :

- **Nova** pour le **SDC**. L'un des nombreux modules d'OpenStack, mais sans doute le plus important, offre les fonctions principales d'une offre de IaaS.

- **Neutron** pour le **SDN**. Après quelques années d'errance, la solution d'*OpenStack* permet de virtualiser le réseau de manière efficace, via un mélange de VLANs et de VxLANs pour dépasser la fameuse limitation historique des 4096 réseaux logiques.

- **Ceph** pour le **SDS**. Capable d'exposer un stockage sous différents protocoles d'accès (Network block device/RBD, CephFS, S3/ObjectStorage et NFS) sur un même cluster de machines disposant de stockage local. Il peut être exposé en direct ou masqué derrière l'offre de stockage Cinder d' OpenStack.

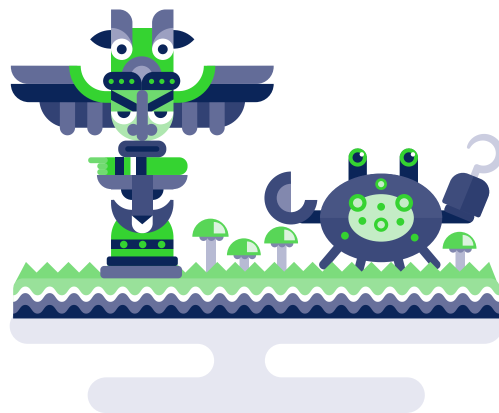
Par extension, quelques autres termes ont vu le jour, sans nécessairement offrir la même clarté dans leur définition :

- **SDI** : ***Software-Defined Infrastructure***

- **SDS** : ***Software-Defined Security***

- **SDDC** : ***Software-Defined DataCenter***

L'idée reste cependant la même : celle d'une **infrastructure banalisée**, physiquement déployée une fois et logiciellement **capable de se reconfigurer très rapidement** pour répondre aux enjeux de transformation des entreprises.



[29] *OpenStack* est une suite de logiciels libres qui permet de déployer une infrastructure *cloud* personnalisée. [30] Voir http://principlesofchaos.org/ pour une description complète de ces principes. Résumons simplement en disant qu'ils consistent à générer en continu et volontairement des crashes sur des ressources (machines virtuelles par exemple) pour valider au fil de l'eau que les systèmes sont tolérants à la panne.

Conclusion

LES RAISONS QUI NOUS FONT AIMER CETTE TECHNOLOGIE

- Une certitude de pouvoir tout automatiser
- Une intégration avec des API standardisées
- Une capacité à modéliser des architectures très complexes et surtout, à très rapidement réarchitecturer des solutions complètes. Un besoin nécessaire quand on cherche à "agilifier" son infrastructure.

LES LIMITES DE CETTE TECHNOLOGIE

- *Vendor lock-in* déguisé, ou même *rebranding* grossier d'une solution historique qui ne tiendra pas les nouvelles promesses des offres SDx : méfiez-vous des acteurs historiques qui essaient de prendre le train en route sans repenser leurs offres !
- Offrir du SDx à vos clients vous impose une rigueur à toute épreuve dans l'automatisation de votre infra, pour être toujours prêt à réagir aux imprévus.

CE QUI FAIT QUE CETTE TECHNOLOGIE TRANSFORME NOTRE MÉTIER

- Il faut penser programmation, code et configuration pour représenter une architecture
- La réorganisation agile des architectures techniques ouvre le champ des possibles
  - > Intégration de déploiements continus de composants d'infrastructure en plus du logiciel
  - > Mise en œuvre d'une stratégie de *Chaos Engineering*[30] pour tester la résilience des systèmes, en continu

EXEMPLE

Les limites

Quand le matériel se rappelle à notre bon souvenir

Attention, le "*Software-Defined Everything*" n'est pas magique. En effet, il s'appuie toujours sur du *hardware* qu'il faut continuer à gérer, et qui nous rappelle de temps en temps cette dure réalité.

Nous en avons eu un exemple flagrant début 2018, avec des failles de sécurité matérielles majeures : les déjà célèbres Meltdown[31] et Spectre.

Des chercheurs d'universités et de Google ont trouvé des failles majeures dans tous les CPUs modernes qui contournent les mécanismes de sécurité des OS. Ces failles sont tellement structurelles et répandues que toutes les catégories de machines sont touchées : les téléphones, les serveurs, presque tous les *desktops*... Seuls votre Raspberry PI et votre machine à laver semblent épargnés !

L'importance de ces failles de sécurité a pris l'ensemble de l'industrie par surprise. Les correctifs nécessitent **des modifications en profondeur et en urgence de nos OS**. Si les premiers patchs sont arrivés, de nouvelles variantes de Spectre n'arrêtent pas de sortir et continueront probablement à apparaître pendant des années.

Optimiser pour mieux régner (sur le marché)

Regardons ensemble le mécanisme de la faille Meltdown. Au cœur de l'attaque se trouvent les optimisations mises en place dans nos CPUs depuis 30 ans. Les constructeurs se heurtent en effet depuis longtemps aux limites de la physique, et doivent en permanence gérer deux problèmes majeurs :

1. Les processus de fabrication ne permettent pas d'augmenter les fréquences à l'infini, et les cycles de R&D sur ce sujet sont trop longs pour rester compétitifs.

2. Si les CPUs ont énormément accéléré, **la vitesse de la RAM n'a pas suivi**. Sans astuce, les CPUs modernes passeraient la majorité de leur temps à attendre la RAM, annulant tous les gains de performance développés.

[31] https://meltdownattack.com/

Afin de nous livrer toujours plus de puissance chaque année, les constructeurs développent donc de nombreuses techniques pour améliorer les performances en attendant que leurs chercheurs avancent sur les problèmes fondamentaux. Si une description détaillée remplirait des livres entiers, nous allons ici explorer deux mécanismes importants pour notre cas : le cache CPU et l'exécution *out-of-order*.

Le cache CPU

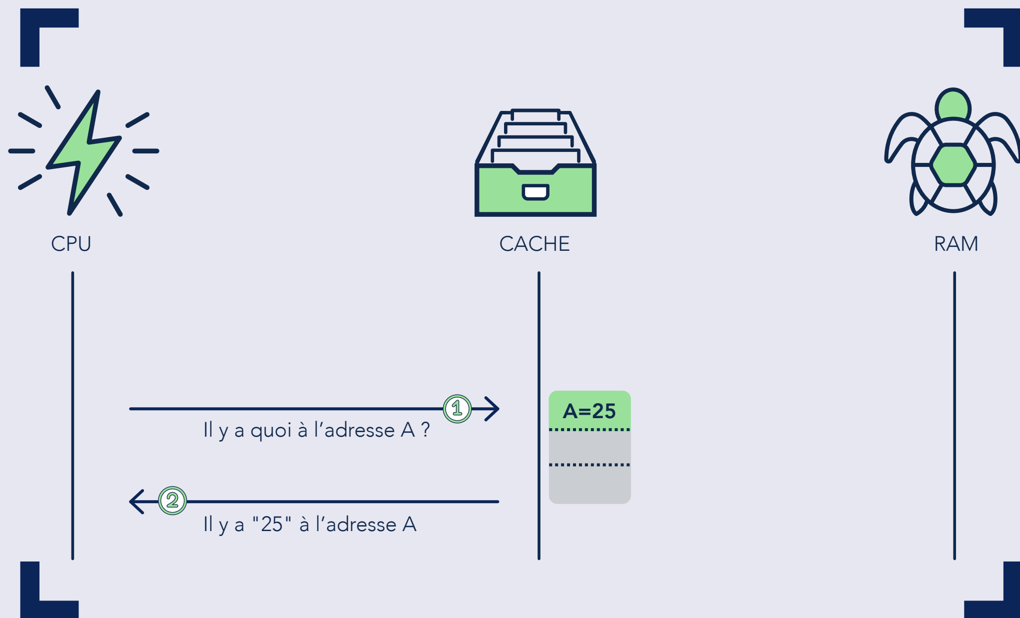
La première direction que les constructeurs ont prise est de mettre du cache dans leurs processeurs : de la mémoire beaucoup plus rapide que la RAM, en petite quantité (elle est très coûteuse), qui permet au CPU d'accéder très rapidement aux données utilisées récemment.

En pratique, quand le CPU a besoin d'une valeur en mémoire, il la demande au cache, qui la demande à la RAM.

CPU
CACHE
RAM
① Il y a quoi à l'adresse A ?
② Il y a quoi à l'adresse A ?
A=25
③ Il y a "25" à l'adresse A
④ Il y a "25" à l'adresse A

EXEMPLE

Le cache stocke la valeur récupérée et est capable de la renvoyer plus rapidement au CPU (jusqu'à 200 fois plus rapidement que la RAM[32]) s'il la demande à nouveau :



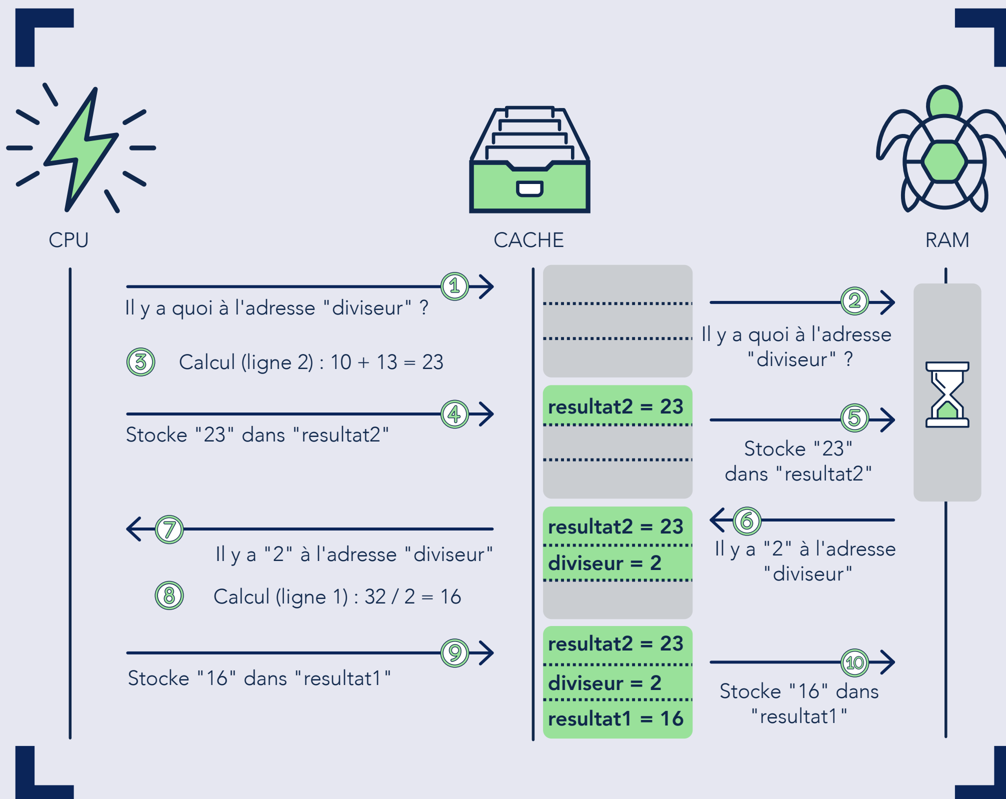
La présence de ce cache est transparente pour le programmeur : il voit seulement que certains accès mémoire sont plus rapides que d'autres, et que son programme s'exécute plus vite dans ce cas.

L'exécution *out-of-order*

Une fois le cache en place pour accélérer l'accès à la RAM, reste le problème de la fréquence du CPU. Comment exécuter plus d'instructions par seconde sans augmenter cette fréquence ? La réponse des constructeurs est d'en exécuter plusieurs en même temps[33]. Les processeurs modernes vont donc lire dans le programme plusieurs instructions à la fois pour les jouer toutes d'un coup. Un exemple avec ce bout de programme :

```
1 resultat1 = 32 / diviseur
2 resultat2 = 10 + 13
```

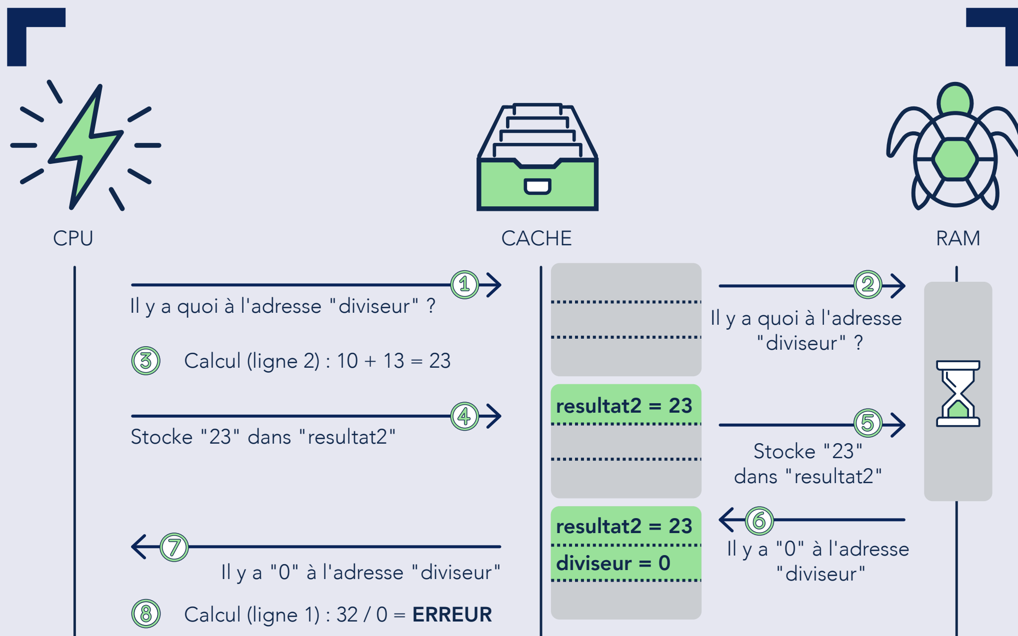
[32] https://gist.github.com/jboner/2841832 [33] On parle de plusieurs instructions par cœur CPU : dans un CPU multi-cœur moderne, chaque cœur peut traiter 6 instructions à la fois.



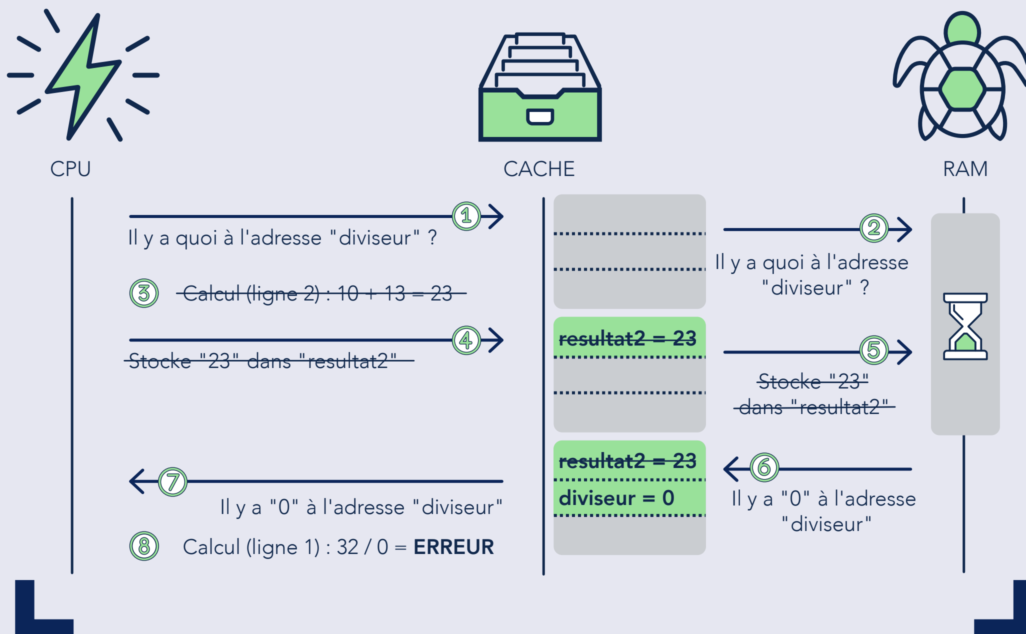
Tout comme le cache, cette optimisation est transparente pour le développeur : le CPU s'arrange toujours pour que le programme "voit" les instructions exécutées dans le bon ordre. Si une instruction génère au final une erreur, le processeur va annuler toutes celles qu'il avait déjà exécutées en avance.

EXEMPLE

Dans notre exemple, si le `diviseur` vaut 0, la division va créer une erreur :



Et le CPU va annuler toutes les modifications effectuées par les instructions suivantes :



Au-delà de notre exemple simple, le processeur va plus généralement analyser le flux d'instructions (notamment pour détecter des dépendances entre les instructions) et les réordonner pour en exécuter un maximum en un minimum de temps. D'où le nom de ce système : exécution *out-of-order*.

L'anatomie d'une faille : au cœur de Meltdown

Vous vous en doutez, Meltdown exploite les deux systèmes décrits ci-dessus. Pour autant, leur utilisation dans une attaque n'est pas évidente. Le cache en lui-même n'est pas directement lisible par un programme : le CPU l'utilise pour accélérer les accès mémoire, mais le programme doit toujours demander l'accès à la RAM en elle-même, et le CPU valide chaque accès, cache rempli ou pas. De même, en cas d'exécution *out-of-order*, le mécanisme d'annulation des instructions va masquer au programme tous les accès à de la mémoire protégée qui ont été effectués.

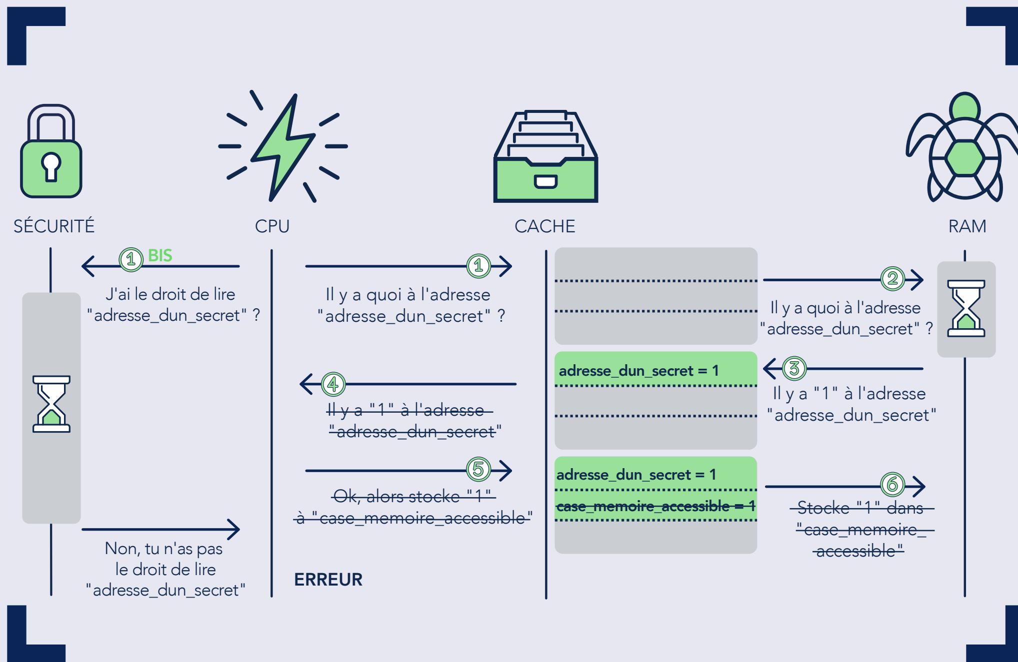
L'approche directe

Prenons un exemple simple. Supposons qu'un attaquant veuille lire le contenu d'un secret (qui peut valoir 0 ou 1) dans la mémoire du noyau, qui est protégée, pour l'utiliser :


```
1 secret = memoire_noyau[adresse_dun_secret] # Mémoire protégée,
illisible pour l'attaquant : ERREUR !
2 memoire_accessible = secret
```

Dans cet exemple, notre attaquant a naïvement essayé de lire un secret du noyau et de le stocker dans une case mémoire qui lui est accessible.

EXEMPLE



Le résultat est le même que pour notre division par zéro ci-dessus : dès que le CPU se rend compte que l'accès à `adresse_dun_secret` est interdit, il lève une erreur et annule les modifications des instructions suivantes.

À ce stade, les valeurs lues depuis la RAM restent chargées en cache. Est-ce là que Meltdown attaque ? Non, ça ne pose pas de problème de sécurité *a priori*, puisque les vérifications de sécurité sont toujours exécutées :

Les mécanismes de sécurité du CPU ont l'air bien implémentés. Comment Meltdown fait-il donc pour les contourner ?

L'approche subtile

Une variation de notre programme ci-dessus va rapidement nous donner la réponse :

```
1 secret = memoire_noyau[adresse_dun_secret] # ERREUR !
2 si secret est_egal_à 1
3     secret_indirect = memoire_accessible_1
4 sinon
5     secret_indirect = memoire_accessible_0
```

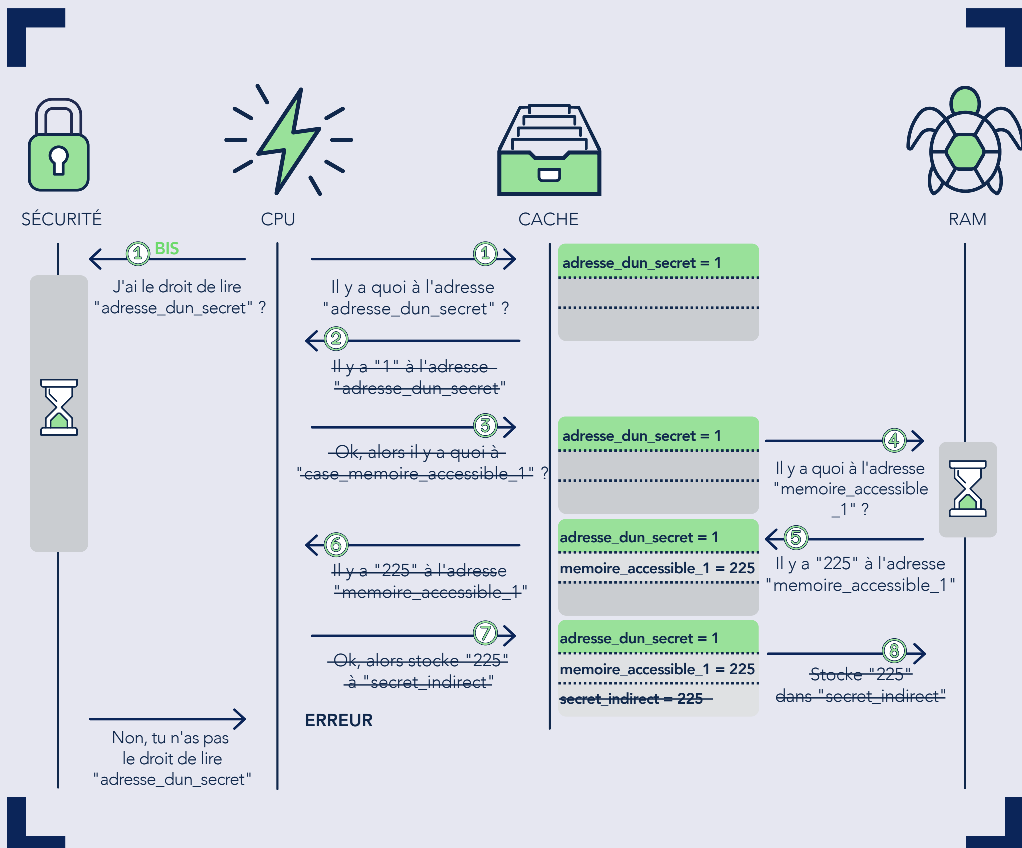
EXEMPLE

À première vue, ce programme ne nous avance pas plus :

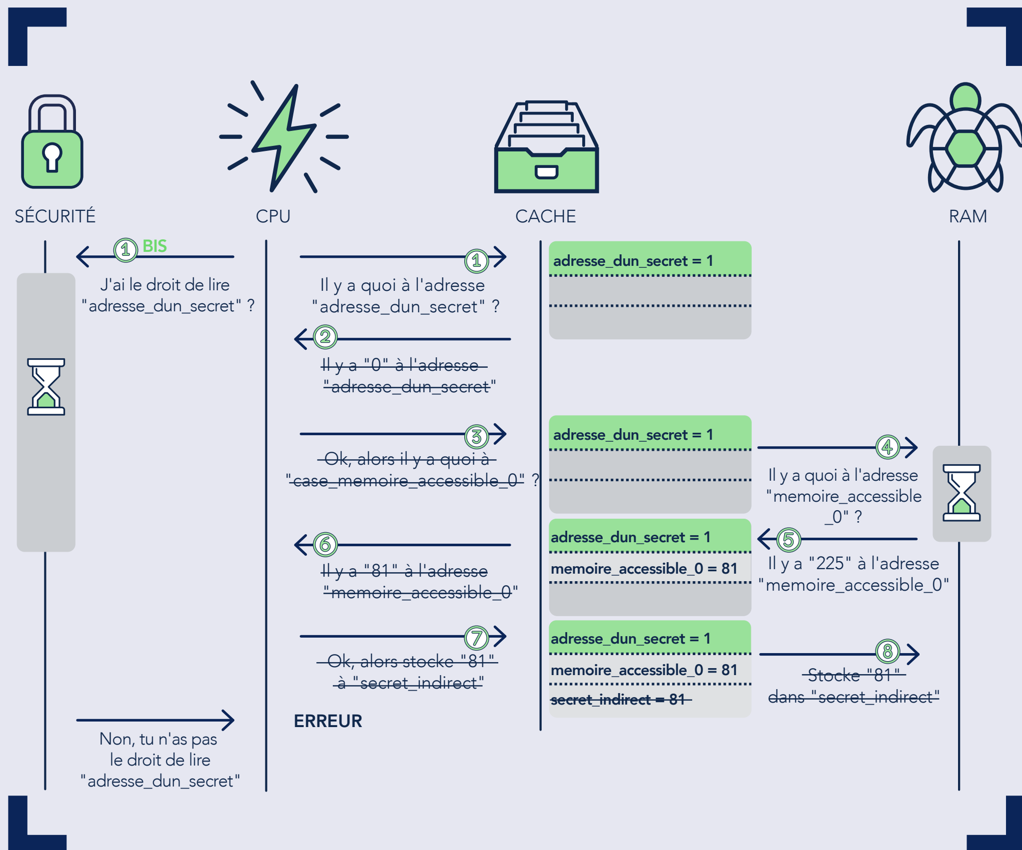
- Certes, l'exécution *out-of-order* peut exécuter les lignes 2 à 5 en avance.
- Certes, la case mémoire lue va dépendre de la valeur du secret.
- Pour autant le mécanisme d'annulation va entrer en jeu, et l'écriture dans `secret_indirect` sera supprimée.

À ce stade vous avez probablement l'impression que l'on vous fait tourner en rond. Pourtant, quelque chose a bien changé dans ce nouveau programme : selon la valeur de secret, la case mémoire chargée en cache (`memoire_accessible_1` ou `memoire_accessible_0`) n'est pas la même :

Si secret est égal à 1



Si secret est égal à 0



EXEMPLE

Et donc ? Eh bien, la vitesse d'accès aux cases mémoire 1 ou 0 a potentiellement été modifiée ! Complétons notre programme :

```
1 sortir_du_cache(memoire_accessible_1)
2 sortir_du_cache(memoire_accessible_0)
3 secret = memoire_noyau[adresse_dun_secret] # ERREUR !
4 si secret est_egal_à 1
5    secret_indirect = memoire_accessible_1
6 sinon
7    secret_indirect = memoire_accessible_0
```

Au démarrage, on s'arrange pour que `memoire_accessible_1` et `memoire_accessible_0` ne soient plus dans le cache : l'accès aux deux sera lent. En revanche, une fois le reste du code passé, seule l'une des deux cases aura été chargée en cache : son accès sera plus rapide. Et on y est : il suffit de vérifier la vitesse d'accès à la mémoire en question. Voyez donc :

```
1 sortir_du_cache(memoire_accessible_1)
2 sortir_du_cache(memoire_accessible_0)
3 secret = memoire_noyau[adresse_dun_secret] # ERREUR !
4 si secret est_egal_à 1
5    secret_indirect = memoire_accessible_1
6 sinon
7    secret_indirect = memoire_accessible_0
8 si vitesse(memoire_accessible_1) est rapide
9    afficher("le secret vaut 1")
10 si vitesse(memoire_accessible_0) est rapide
11   afficher("le secret vaut 0")
12 sinon
13   afficher("l'attaque a échoué")
```
[34]

(34) L'attaque ne réussit pas toujours à 100 % même sur un CPU vulnérable : l'ordre réel des opérations effectuées par le CPU varie, et le contrôle de sécurité peut revenir avant que les instructions suivantes soient exécutées. Un échec est en revanche facilement identifiable par l'attaquant, et il peut facilement réessayer.

Le résultat est là : sans avoir directement lu la valeur de secret, l'attaquant a pu deviner son contenu à travers l'analyse du cache. Cette technique peut facilement être étendue pour lire des octets complets : l'attaquant doit simplement préparer 256 cases mémoire judicieusement placées en RAM, et s'arranger pour que chaque valeur possible du secret charge une case différente en cache.

Préparez-vous à l'impact : les conséquences de Meltdown

L'attaque présentée ci-dessus permet en pratique de lire l'ensemble du contenu du noyau du système d'exploitation, que ce soit Linux ou Windows. Les chercheurs qui l'ont découverte ont réussi à atteindre 500 Ko/s en vitesse de lecture, ce qui est suffisant pour lire de grandes parties du noyau en quelques minutes. Le noyau en lui-même contient de nombreux secrets[35], et est donc une cible de choix pour un attaquant.

Pire encore, le noyau inclut une bonne partie de la mémoire des autres processus de la machine[36] : un attaquant peut se servir de l'attaque pour lire la mémoire des autres programmes de la machine. Ainsi, **tout secret chargé en mémoire** (clé privée associée au certificat SSL d'un service web, mot de passe, etc.) **devient vulnérable**.

Les conséquences de Meltdown sont donc désastreuses pour la sécurité informatique. Tous les OS se sont basés sur les mêmes **mécanismes de protection mémoire** pour se protéger d'un attaquant, et Meltdown fait complètement fondre ces mécanismes. L'ampleur de la faille est également grande : toutes les générations de CPUs Intel depuis 2010 sont concernées, et certains ARM aussi[37], rendant vulnérables ordinateurs de bureau, serveurs et téléphones.

Un facteur limite légèrement l'impact : l'attaquant doit pouvoir exécuter du code très spécifique sur la machine cible pour exploiter la faille, et doit donc avoir au préalable **un point d'entrée sur sa victime**. Ce n'est malheureusement pas très rassurant pour les *desktops* et téléphones : une attaque peut aisément se cacher dans une application Android ou iPhone discrète, voire dans le Javascript d'une page web... Si les serveurs sont également vulnérables, ils exécutent généralement du code vérifié, et un attaquant devra déjà trouver une autre faille avant de pouvoir bénéficier de Meltdown.

[35] Par exemple, l'attaquant pourrait récupérer l'état du générateur de nombres aléatoires du noyau (qui alimente /dev/random, OpenSSL et autres programmes de sécurité), lui permettant ainsi de prédire les futurs nombres renvoyés par ce générateur. [36] Pour une raison totalement légitime : les programmes passent par le noyau pour toute opération de lecture/écriture sur le disque ou le réseau, et le noyau doit donc accéder aux pages des programmes pour les remplir. [37] La liste n'est pas définitive : la faille est théoriquement possible sur quasiment toutes les gammes de CPUs depuis des années, même si les chercheurs n'ont pas réussi à les exploiter ailleurs… Pour l'instant.

EXEMPLE

Une rupture difficile, mais nécessaire

Les protections contre Meltdown passent dans leur ensemble par des mises à jour des systèmes d'exploitation. Tous les grands fournisseurs d'OS ont déjà publié des patches, vous trouverez de nombreux pointeurs sur le site de Meltdown[38]. La nature de la mise à jour se résume à séparer le noyau dans son propre processus, totalement isolé des autres. **Chaque processus a en effet une vue totalement différente de la mémoire** : le CPU est limité à cette vue de la mémoire pour toutes ses opérations. Pour accéder au noyau, il faut désormais changer entièrement de vue mémoire, ce qui n'est pas possible depuis un programme, même via l'exécution *out-of-order*.

La séparation du noyau dans son propre processus permet de totalement bloquer Meltdown, mais avec un coût majeur : l'accès au noyau est fortement ralenti. Le changement de processus est en effet coûteux, beaucoup plus qu'un simple accès à la mémoire protégée. Or, les programmes ont besoin du noyau pour de nombreuses opérations, comme l'accès au réseau et au disque dur : les impacts de performance sont donc très variables en fonction du type de programme. En pratique, les résultats des benchmarks sont pour l'instant très variables : l'impact va fortement varier d'une application à l'autre[39].

La sécurité est un processus, pas un état

Meltdown et sa petite soeur Spectre ont pris l'ensemble de l'industrie par surprise de par leur nature : les mécanismes utilisés sont présents depuis des dizaines d'années dans les CPUs sans que personne ne se soit aperçu de leur possible détournement. Par son ampleur également : quasiment tous les ordinateurs (dans une définition large) sont impactés.

Leur correction complète va d'ailleurs s'avérer très longue : Spectre est tellement subtile et complexe que ses ramifications vont générer de nombreux patches dans les prochains mois ou années. C'est une nouvelle catégorie d'attaques que les chercheurs en sécurité commencent seulement à explorer[40]. Greg Kroah Hartmann, l'un des développeurs centraux de Linux le dit de manière éloquente :

"Mettez à jour vos noyaux, n'attendez pas, et ne vous arrêtez pas. Les mises à jour pour résoudre ces problèmes vont continuer à apparaître pendant très longtemps".[41]

---

(38) https://meltdownattack.com/#faq-advisory (39) https://access.redhat.com/articles/3307751 (40) Les chercheurs s'intéressent depuis longtemps aux bugs des CPUs, comme le résume bien cette vidéo : https://www.youtube.com/watch?v=e2vPp0fQUkM. Ces bugs vont certainement être analysés à nouveau dans les prochains temps à la lumière de Meltdown et Spectre (41) Traduction par nos soins de "Again, update your kernels, don't delay, and don't stop. The updates to resolve these problems will be continuing to come for a long period of time." (http://www.kroah.com/log/blog/2018/01/06/meltdown-status/)

Nous proposons même d'aller plus loin :

"Soyez toujours prêts à mettre à jour vos systèmes : immédiatement, en masse, et du jour au lendemain".

En effet, Meltdown et Spectre ne sont que les dernières failles d'une longue série : Poodle[42], Heartbleed[43], Dirty COW[44]… Toutes ces failles ont en commun d'avoir touché toutes les couches de l'infrastructure, et d'avoir nécessité des mises à jour en masse et en urgence.

"Alors que nos systèmes d'information se complexifient à vue d'œil, nous devons redoubler d'efforts pour suivre l'état de l'art de sécurité."

L'état des lieux de la sécurité nous oblige donc à être toujours plus réactifs, toujours plus… agiles. Alors que nos systèmes d'information se complexifient à vue d'œil, nous devons redoubler d'efforts pour suivre l'état de l'art de sécurité.

L'automatisation des déploiements et de la maintenance nous paraît donc incontournable pour conserver un SI fiable et maîtrisé. Il nous paraît en effet illusoire d'espérer gérer manuellement une infrastructure à la fois utile, à l'échelle et sécurisée. Heureusement, une autre tendance est en pleine ébullition et prête à nous aider : l'*Infrastructure as Code*.

42 https://en.Wikipedia.org/wiki/POODLE 43 http://heartbleed.com/ 44 · https://dirtycow.ninja/

L'Infrastructure as Code

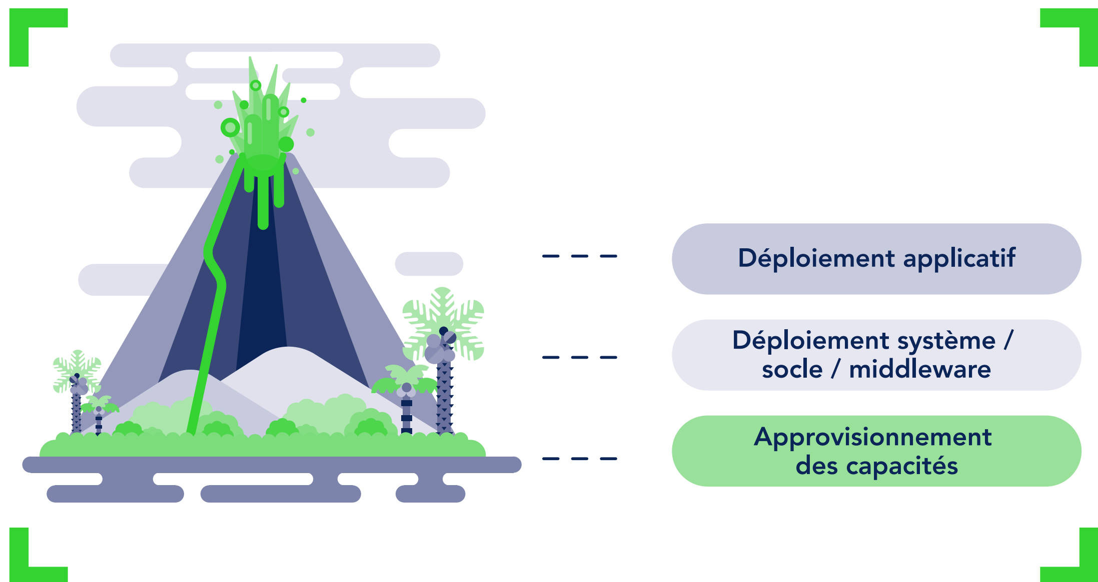
*Derrière le terme **IaC** se cache donc l'Infrastructure as Code. L'**IaC** permet d'automatiser la gestion des infrastructures et des systèmes : approvisionnement, déploiement, configuration, gestion des services...*

Introduction

L' *Infrastructure as Code* est vue comme un empilement de 3 strates permettant de passer de rien à un système intégralement déployé.

Les 3 étages se veulent relativement isolés les uns des autres permettant une certaine abstraction entre les couches. Le déploiement système et *middleware* n'a, en théorie, que peu d'adhérence avec la technologie utilisée pour fournir les ressources (machines physiques, VMs…).

L'automatisation se fait de façon programmatique : du code décrit le quoi, le où, le combien, parfois le comment. L'**IaC se veut la branche exécutable du fameux DAT (Dossier d'Architecture Technique)**. Et c'est bien la vertu de cette approche. Si un DAT est régulièrement faux, pas à jour, incomplet, imprécis, une plateforme fraîchement bâtie par de l'IaC est nécessairement à l'exacte image du code qui l'a produite. Le terme "fraîchement" prend ici toute son importance. Si l'on commence à modifier les machines à la main, cette assertion ne reste pas vraie très longtemps.



Historique

L'*Infrastructure as Code* est née bien avant l'apparition du terme DevOps. À titre d'exemple, le projet *CFEngine*, père de tous les autres langages de gestion de configuration (Puppet, Chef, Ansible, Saltstack et bien d'autres) pointait le bout de son nez dès 1993.

Le besoin d'automatisation dans les déploiements de parcs de machines n'est pas apparu avec la virtualisation. Les pratiques DevOps l'ont simplement rendu indispensable. Ces pratiques préconisent de proscrire les opérations manuelles sur les machines au profit de l'IaC. Les SI étant de plus en plus gros et intégrés, leur utilisation tend donc à se généraliser à l'échelle de tout un produit, un pan de l'organisation voire l'entreprise en entier. Les langages de type IaC sont venus proposer une alternative au scripting classique (sh, ksh, bash, Python ou Perl...). Ils sont particulièrement adaptés pour des travaux d'installation et de configuration de systèmes ou d'applications.

"Des légendes urbaines racontent que certains consultants OCTO seraient nés après 1993, mais ne nous égarons pas !"

Approche déclarative vs. impérative

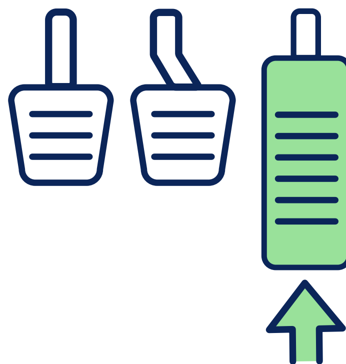
Comme tout langage de programmation digne de ce nom, les langages d'IaC sont gérés comme du code. Ils ne coupent pas à des pratiques d'utilisation tout à fait classiques dont nous reparlerons quand il s'agira de garantir la qualité du code de type IaC. Les langages d'**IaC** sont généralement conçus autour du paradigme de **programmation déclarative**.

Approche déclarative

"Dis-moi l'état que tu attends, je saurai comment l'atteindre."

Cela pourrait être en substance le leitmotiv du modèle de programmation déclaratif de l'IaC. Le développeur décrit l'état attendu et le moteur d'exécution connaît les opérations à effectuer pour vérifier si cet état est déjà présent et l'atteindre dans le cas contraire. Les anglophones parlent de DSC pour *Desired State Configuration*[45]. Pour illustrer la différence de fonctionnement entre les **langages impératifs** et les **langages déclaratifs**, faisons l'exercice périlleux de la métaphore en comparant la pédale d'accélérateur et le régulateur de vitesse.

La pédale d'accélérateur adopte le comportement d'une **programmation impérative**. C'est à l'usager (i.e. le conducteur) de savoir si sa vitesse est en-dessous ou au-dessus de celle recherchée pour décider s'il faut appuyer, maintenir ou relâcher le champignon. C'est également à lui de faire des ajustements par vent de face, dans les montées, les descentes, etc, tout en gardant en permanence un œil sur le compteur de vitesse.



[45] C'est d'ailleurs ainsi que Microsoft a appelé son système d'IaC propriétaire pour Windows !

Le régulateur de vitesse adopte le comportement d'une **programmation déclarative** : nous lui donnons simplement l'ordre de la vitesse à atteindre et il s'occupe du reste. Il regarde s'il n'est pas déjà à la bonne vitesse, auquel cas il ne change rien à la puissance moteur (concept d'idempotence). Il porte toute la logique d'observation de la vitesse actuelle, d'adaptation (vers le haut ou vers le bas) de la puissance moteur, etc.

C'est ce dernier principe de fonctionnement qui s'applique à l'IaC en mode déclaratif. Il suffit de fixer des objectifs à l'automate qui va fonctionner comme régulateur de vitesse :

- Assure-toi que tel logiciel est installé
- Assure-toi que tel fichier est présent et avec tel contenu
- Assure-toi que tel service est éteint.

Limites de l'approche purement déclarative

Ce principe de fonctionnement est extrêmement intéressant pour des tâches unitaires et simples de déploiement et configuration. Fort heureusement celles-ci représentent facilement 80 à 90 % des cas.

Quelques cas très récurrents apparaissent de façon désagréable pour venir heurter ce principe de fonctionnement d'une pureté biblique. En premier lieu, la plupart des déploiements sont certes déclaratifs, mais doivent également être ordonnés et conditionnés. Démarrer un service alors que le logiciel n'est pas encore installé n'a aucun intérêt. De même, il est nécessaire de redémarrer un service lorsque l'on change son fichier de configuration, mais superflu voire néfaste si le redémarrage n'est pas nécessaire. Entrent alors en jeu **les notions de dépendances et de notifications** qui permettent de remettre de l'ordre dans la maison et d'appliquer une logique quasiment procédurale.


```
S'assurer que le paquet truc est installé
S'assurer que le fichier de conf. de truc a bien le bon contenu
Si le service truc n'est pas démarré, alors
  S'assurer que le service truc est démarré
Sinon
  Si le fichier de conf. de truc a été modifié, alors
     S'assurer que le service truc est redémarré
```

Les solutions d'IaC proposent un panel de réponses plus ou moins élégantes pour gérer ce genre d'algorithme finalement plutôt simple ; tant sur l'ordre des étapes (via des dépendances par exemple) que sur l'arbre de décision des actions à opérer (via des mécanismes de notification).

Un peu de procédural pour des cas plus complexes

Restent toujours des cas plus complexes que l'on rencontre généralement en montant dans les étages de la fusée, vers le **déploiement applicatif**. L'approche déclarative devient alors de moins en moins pertinente et un bon vieux retour au **code impératif** permet d'implémenter des algorithmes *ad hoc*, comme dans le cas d'un déploiement d'une application sans interruption de service :

La gymnastique algorithmique est ici assez poussée car elle impose non seulement de boucler sur un certain nombre d'éléments (les serveurs applicatifs et les *load-balancers* dans le cas présent), mais aussi de travailler de façon très précise sur des groupes de machines différents au travers d'une même exécution. Ici, la reconfiguration concerne à tour de rôle les *load-balancers* et les serveurs applicatifs qui sont *a priori* des machines différentes. Il s'agit par conséquent d'orchestrer une chorégraphie assez subtile entre plusieurs machines.


```
Pour chacun des serveurs applicatifs s parmi s1, s2 et s3:
   Retirer s des load-balancers lb1 et lb2
   Attendre que l'application a sur le serveur s ait fini ses traitements en cours
   Éteindre l'application a du serveur s
   Mettre à jour l'application a sur le serveur s
   Démarrer l'application a sur le serveur s
   Faire des tests de santé de l'application a sur le serveur s
   Remettre s dans les load-balancers lb1 et lb2
```

Différentes familles d'Infrastructure as Code (IaC)

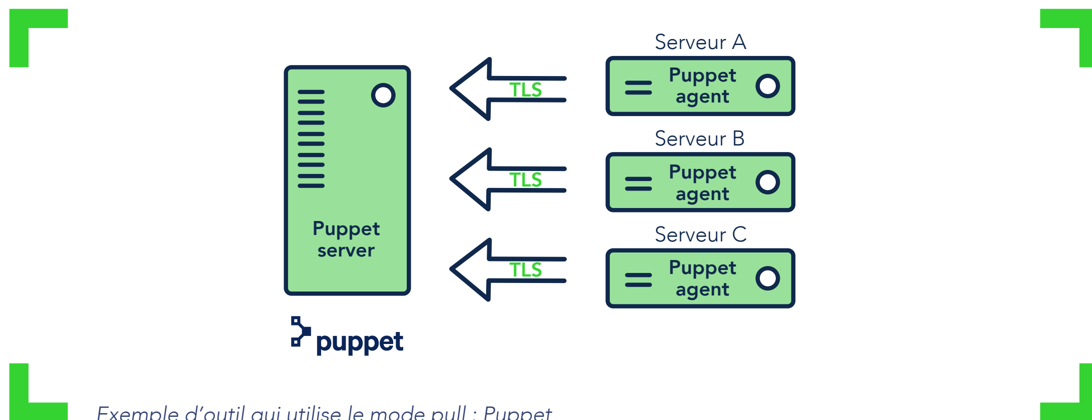
Derrière le terme d'*Infrastructure as Code* (IaC) se cachent donc des solutions assez différentes implémentant soit des approches purement déclaratives, soit des approches purement impératives, soit un mélange plus ou moins élégant des deux.

À titre d'exemple, nous pouvons citer :

- Des solutions centrées essentiellement sur le déclaratif : Terraform, Puppet, Chef, CFEngine
- Des solutions spécialisées dans l'impératif comme Fabric et Capistrano
- Des solutions hybrides comme Ansible.

Un autre critère permettant de classifier les solutions d'IaC est leur mode d'exécution. Là où certaines solutions adoptent un fonctionnement en mode "*push*" à partir d'une machine de contrôle, certaines font le choix inverse d'une approche de type "*pull*" où des agents locaux viennent chercher auprès d'un serveur central une liste de tâches à accomplir.

Exemple d'outil qui utilise le mode push : Ansible



Exemple d'outil qui utilise le mode pull : Puppet

Dans ce modèle, il est nécessaire d'installer un agent sur chacune des machines managées. Cet agent interroge régulièrement le Puppet Master pour obtenir l'ensemble des états attendus.

Ces choix de fonctionnement ont des impacts majeurs sur plusieurs critères : mode d'installation (serveur central vs. machine de contrôle), mode de communication (client -> serveur, server -> client), scalabilité, vision synchrone ou asynchrone de l'exécution… Sans rentrer dans un grand comparatif des avantages et inconvénients des deux approches, tâchons de classifier les produits par leur mode de fonctionnement :

- Mode agent de type *pull* : Puppet, Chef, CFEngine
- Mode *push* à partir d'une machine de contrôle : Fabric, Capistrano, Ansible, Terraform

À noter toutefois que même s'ils fonctionnent naturellement en mode *pull* avec un serveur de référence, les outils de type Puppet ou Chef peuvent être déclenchés manuellement voire fonctionner en mode sans serveur à partir de code local (puppet apply, chef-solo).

Dernier critère pour différencier nos belligérants : leur **positionnement naturel** sur les trois strates, entre approvisionnement des capacités, gestion des socles et déploiement applicatif.

Et là, il faut se rendre à l'évidence, tous les produits ne ciblent pas le même objectif. Certains sont très spécialisés sur des fonctions qu'ils font très bien, là où d'autres tentent de travailler plus en largeur en couvrant le maximum de périmètre possible.

D'autre part, certains acteurs du marché se déplacent et tentent d'augmenter leur champ d'application.

Le choix des outils n'est par conséquent pas nécessairement une tâche triviale, et va faire naître des débats d'experts, impliquer des stratégies d'intégration différentes.

Déploiement applicatif
Déploiement système / socle / middleware
Approvisionnement des capacités
Terraform
Puppet
Chef
CFEngine
Capistrano
Ansible

Abstraction

Pour pouvoir masquer une grande partie de la complexité de travail de l'IaC, il est nécessaire de gagner en abstraction. Pour ça, les logiciels d'IaC modélisent les concepts de **ressources** et de **fournisseurs**. La terminologie employée ici est celle de Puppet. L'équivalent des **ressources** Puppet chez Ansible s'appelle les **modules**[46].

La **ressource** est le concept unitaire sur lequel nous allons chercher à définir un état attendu. En IaC, les ressources les plus usuellement rencontrées sont les paquets, les fichiers, les services, les utilisateurs. Voici un exemple Ansible pour demander à ce qu'un paquet soit installé ntp dans le cas présent :

```
- name: Ensure NTP package is installed
  package:
    name: ntp
    state: present
```

Le **fournisseur** est la logique cachée qui implémente le déploiement de la ressource en fonction du contexte. Si l'on reprend l'exemple précédent, l'implémentation de la ressource diffère d'une distribution Linux à une autre :

- Pour Debian ou Ubuntu
  - > `dpkg-query` pour vérifier si le paquet est déjà installé
  - > `apt-get install` pour installer le paquet
- Pour RedHat et CentOS
  - > `rpm -q` pour vérifier si le paquet est déjà installé
  - > `yum install` pour installer le paquet

Pour des **fonctions standard**, les fournisseurs (aussi appelés *providers*) sont déjà implémentés dans l'outil d'IaC et il n'est pas nécessaire de les coder. Toutefois, les langages d'IaC prévoient de pouvoir être étendus, soit pour ajouter un nouveau *provider* pour une ressource existante, soit pour implémenter un nouveau type de ressource et au moins un provider associé. C'est aussi dans l'extensibilité de ses solutions que se trouve leur puissance. Reste à comprendre le modèle de programmation de ces providers, d'utiliser le langage dans lequel il est prévu (Ruby pour Puppet par exemple, Python pour Ansible).

[46] Nous verrons par la suite que les modules Puppet désignent un autre concept.

Composition

L'autre fonction très appréciée des outils d'IaC repose sur leur capacité de composition avec la possibilité de créer des niveaux d'abstraction "métiers". En fonction des technologies, on parlera de **modules et classes** (Puppet), **rôles** (Ansible), **cookbook** (Chef). Un **module** se veut une encapsulation de plusieurs **ressources** pour en faire un tout cohérent répondant à un besoin de plus haut niveau. L'exemple le plus simple est l'installation d'une application standard.

Pour décrire l'**anatomie d'un rôle**, prenons l'exemple de NTP : le service standard de synchronisation de l'horloge des machines UNIX.

L'installation de ntp est un exemple très simple et tout à fait classique de **PFS** :

- **P** : le **P**aquet NTP doit être installé
- **F** : le **F**ichier de configuration ntpd.conf doit être produit sur le serveur avec une liste de serveurs NTP de référence
- **S** : le **S**ervice *ntpd* doit être démarré ou redémarré en cas de changement du fichier de configuration

Quelques paramètres permettent de configurer le rôle, mais sa ré-utilisation se veut très simple : **le module masque la complexité d'implémentation**. Dans le cas du rôle ntp, un unique paramètre pourrait être une liste de serveurs NTP de référence avec lesquels la machine cible doit se synchroniser. Ce paramètre pourrait même être optionnel avec une valeur par défaut de bon sens comme l'utilisation d'un serveur de temps public sur Internet : pool.ntp.org.

Nos chouchous du moment

Difficile de parler du plaisir de la technologie sans nommer ces outils qui font partie de notre quotidien, ceux que nous allons naturellement utiliser ou recommander, même s'ils sont imparfaits.

Puppet

Celui que l'on pourrait surnommer la **"Rolls des IaC" de type déclaratif**, Puppet est le projet de référence, tant au niveau de son fonctionnement qu'au niveau de l'écosystème qu'il draine : pratiques de codage et de qualité, outillages divers et variés, communautés d'utilisateurs, quantité de modules communautaires disponibles. Sa forge est parmi les plus fournies. Nous aimons à le préconiser sur des parcs très volumineux (plusieurs milliers de nœuds), mais pas forcément très intégrés entre eux. Il doit souvent être utilisé de pair avec un autre outil comme Terraform pour prendre en charge l'approvisionnement des capacités voire avec Capistrano pour l'orchestration des déploiements applicatifs. Puppet définit une syntaxe (appelée aussi DSL[47]) qui lui est propre, très déclarative et permettant une modélisation très avancée. Il est extensible par le développement de code Ruby.

```
class
  configure_user(
    $test_user = 'test'
  ) {
  user {$test_user:
    ensure => absent,
  }
  file {"/home/${test_user}":
    ensure => absent,
  }
  group {'admin':
    ensure => present,
  }
}
class
  httpd {
  package {'httpd':
    ensure => 'installed',
  }
}
```


47 DSL : *Domain Specific Language.*

Cet extrait de code permet de supprimer un utilisateur ainsi que son dossier dans /home sur l'OS, ajouter un groupe nommé admin si non présent et s'assurer que le paquet httpd (apache) est bien installé. Pour chacun de ces éléments, Puppet ne changera rien au système si celui-ci est déjà dans le bon état. Si ce n'est pas le cas, il exécutera les commandes d'installation ou de configuration nécessaires.

Ansible

Le couteau Suisse de l'IaC, à la fois capable d'adresser l'**approvisionnement des machines** que **l'orchestration "futée" du déploiement des applications**, c'est la solution unique qui parvient à elle-seule à démarrer quasiment n'importe quel projet d'automatisation de déploiement. Nous apprécions sa simplicité de premier abord, qui permet d'être productif en quelques heures seulement, et sa capacité native à orchestrer le déploiement de plusieurs machines. Nous regrettons cependant une **plus faible capacité** (par rapport à Puppet) à appliquer de bonnes pratiques de qualité de code (Tests unitaires notamment). Quelques régressions d'une version à l'autre viennent également ternir le tableau. Ansible cible des machines listées (statiquement ou dynamiquement) dans un inventaire :

```
[load-balancers]
lb1-srv
lb2-srv

[web-servers]
app1-srv
app2-srv

[db-masters]
db1-srv

[db-slaves]
db2-srv

[db-servers:children]
db-masters
db-slaves
```

Cet extrait de code correspond à l'extrait d'un fichier d'inventaire Ansible. Il permet de définir des groupes de machines et pour chacun de ces groupes les machines qui le composent. Ainsi, dans le reste du code Ansible, on applique des tâches sur des groupes de machines plutôt que sur une liste exhaustive de machines. L'avantage évident est qu'il est possible de faire varier le nombre de machines juste en changeant le fichier d'inventaire.

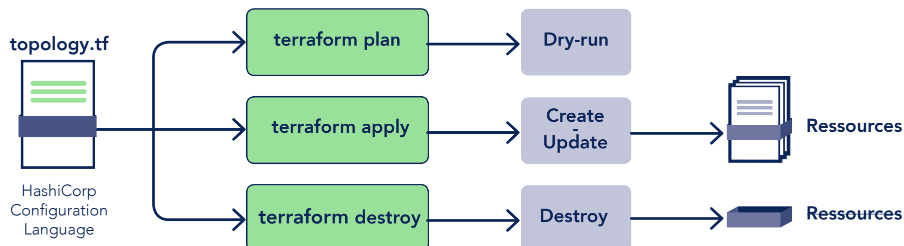
Ansible utilise en outre une syntaxe en YAML pour définir son code. Il est extensible par du développement en Python.

```
# playbook install.yml
---
- hosts: all
  become: true
  gather_facts: true
  tasks:
  - name: Installation NTP
    package:
      name: ntp
  - name: Activation NTP
    service:
      name: ntp
      state: started
      enabled: true
```

Cet extrait de code s'assure que le paquet ntp est présent sur toutes les machines, qu'il est actif au boot et démarré : si ce n'est pas le cas, il réalise les actions nécessaires pour se mettre en conformité avec cet état attendu.

Terraform

Nous l'avons évoqué dans la partie logiciel libre, Terraform de la société HashiCorp est spécialisé dans l'approvisionnement des infrastructures sur des *clouds* ou diverses technologies de virtualisation. Bien que non fourni par Amazon, il se paye le luxe de gérer plus efficacement les ressources AWS que ne le fait l'offre Amazon officielle, CloudFormation. En **constante évolution** (parfois par nos modestes contributions), Terraform se diversifie en prenant en charge de plus en plus d'applications et de *middlewares* en permettant de définir leur configuration comme le ferait Puppet ou Ansible. Il gagne progressivement en abstraction et devient mission après mission un équipier de choix.



```
resource "aws_security_group" "site1-allow-external" {
  name = "site1-allow-external"
  description = "allows external SSH, ICMP, and HTTPS"
  vpc_id = "${aws_vpc.site1_vpc.id}"
  egress {
    from_port = 443
    to_port = 443
    protocol = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port = 22
    to_port = 22
    protocol = "tcp"
    cidr_blocks = ["88.77.66.55/32"]
  }
  ingress {
    from_port = 443
    to_port = 443
    protocol = "tcp"
    cidr_blocks = ["88.77.66.55/32"]
  }
  ingress {
    from_port = -1
    to_port = -1
    protocol = "icmp"
    cidr_blocks = ["88.77.66.55/32"]
  }
  tags {
    Name = "site1-security-group-external"
  }
}
```

Autrement dit :

- Autoriser toutes les connexions vers l'extérieur
- Les connexions entrantes ne sont autorisées que depuis l'IP "88.77.66.55", uniquement sur les ports 22 (SSH) et 443 (HTTPS) en TCP
- Le protocole ICMP (le ping) est également autorisé depuis cette IP.

```
resource "aws_instance" "front-0" {
  ami           = "ami-4d46d534"
  instance_type = "t2.large"
  key_name = "${var.aws_keypair}"
  vpc_security_group_ids = [ "${aws_security_group.site1-allow-internal.id}",
"${aws_security_group.site1-allow-external.id}" ]
  associate_public_ip_address = true
  subnet_id = "${aws_subnet.main.id}"
  tags {
    Group = "APP1_FRONT"
    Name = "front-0"
  }
}
resource "aws_instance" "back-0" {
  ami = "ami-4d46d534"
  instance_type = "t2.large"
  key_name = "${var. aws_keypair}"
  vpc_security_group_ids = [ "${aws_security_group.site1-allow-internal.id}" ]
  associate_public_ip_address = false
  subnet_id = "${aws_subnet.main.id}"
  tags {
    Group = "APP1_BACK"
    Name = "back-0"
  }
}
```

Cet extrait de code permet de créer 2 instances AWS (équivalent à 2 VMs) : une nommée back-0 et une nommé front-0. Elles possèdent quasiment les mêmes caractéristiques hormis que celle nommée front-0 reçoit une adresse IP publique et est attachée à 2 security groups.

Conclusion

L'IaC a atteint un niveau de maturité qui la rend incontournable de nos jours. Déployer des parcs de machines manuellement n'est jamais paru aussi désuet et risqué. Le risque n'est toutefois pas éliminé à 100 % si la manipulation de l'IaC ne s'accompagne pas d'une hygiène irréprochable dans la gestion et la qualité du code. Certains *twittos*[48] se font un malin plaisir de décrire les ravages d'une IaC mal maîtrisée. Le troisième volume de cette série – relatif au goût du travail de qualité dans un environnement DevOps – décrit les stratégies de test à impérativement associer à l'IaC.

LES RAISONS QUI NOUS FONT AIMER CETTE TECHNOLOGIE

- L'approche état désiré / idempotence
- Le côté extensif de solutions
- Des constructions / déploiements d'infrastructures automatiques, répétables, rapides
- Du code versionnable dans Git pour comprendre l'historique de l'infrastructure

LES LIMITES DE CETTE TECHNOLOGIE

- Comme tous les logiciels, ils ne sont pas parfaits (complexité de certains, qualité d'autres)
- Dualité des besoins impératifs / déclaratifs dans la vraie vie
- Écrire de l'*Infrastructure-as-Code* ne se fait pas sans écrire des tests !

CE QUI FAIT QUE CETTE TECHNOLOGIE TRANSFORME NOTRE MÉTIER

- Toute l'infrastructure peut être décrite, installée, testée et versionnée
- Les administrateurs peuvent se concentrer sur l'amélioration du SI plutôt que sur des tâches facilement robotisables, sans valeur ajoutée à l'intelligence humaine

[48] https://twitter.com/devops_borat/status/41587168870797312

La conteneurisation

L'arrivée de Docker et de son écosystème propose un nouveau modèle de gestion des applications. Il est suffisamment différent des modèles précédents pour provoquer des changements à plusieurs titres.

Introduction

En lice pour le plus gros *buzzword/silver bullet* de ces cinq dernières années, il y a évidemment Docker avec son moteur d'exécution et son format de conteneurs. Docker est arrivé en 2013 avec une promesse : "si ça marche en dev', ça marche en prod'".

Le projet a connu un succès fulgurant, et a généré une énorme traction. Il a mis en lumière **des problématiques liées à l'infrastructure** (durée de vie des unités de déploiement et donc des applications qu'elles hébergent), **au déploiement** (hétérogénéité des formats de packaging, et des moyens de distribution), **à la sécurité**, **à la séparation des responsabilités entre les équipes**... Si tant est qu'aujourd'hui, la conteneurisation véhicule un fantasme en particulier : il s'agit de La Solution – technique et unique – à tous les problèmes, qui saura nécessairement trouver sa place dans tous les SI.

"Notre conviction est que les conteneurs véhiculent des concepts vecteurs de transformations profondes de notre industrie, mais qui ne s'arrêtent pas à la simple technologie. "

Notre conviction est que les conteneurs véhiculent des concepts vecteurs de transformations profondes de notre industrie, mais qui ne s'arrêtent pas à la simple technologie.

Pourtant, les technologies de conteneurisation sous UNIX/Linux remontent à de nombreuses années, au rang desquels BSD (jails), Linux (Linux-vServer), Solaris (zones), AIX (WPAR). Certaines d'entre elles ont plus de 15 ans.

Comme bien souvent avec les innovations informatiques, la technologie de conteneurisation "Docker" n'est pas issue d'une invention majeure. **Elle est le fruit de l'assemblage – génial, convenons-en – de plusieurs concepts pré-existants**. C'est dans les vieux pots qu'on fait la meilleure soupe !

La conteneurisation en quelques lignes

Tentons de donner une définition simple au principe de conteneurs. Un conteneur est **une abstraction** qui garantit qu'un groupe de processus s'exécutant sur une machine :

- Croit être le seul à utiliser le système d'exploitation (et donc le matériel sous-jacent)
- Ne peut pas voir ses voisins (les autres conteneurs et processus système)
- Ne peut "s'échapper" de l'espace dans lequel il est contenu.

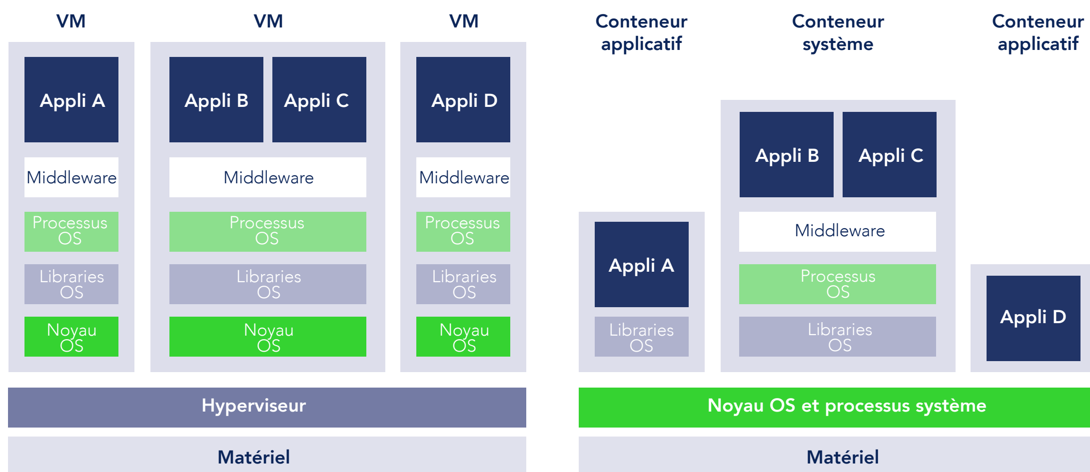
Bien souvent, on utilise l'analogie entre machine virtuelle et conteneur pour introduire le concept. Elle n'est pas dépourvue d'intérêt mais, comme toutes les analogies, elle a ses limites.

Points communs : conteneurs et machines virtuelles permettent *in fine* d'isoler des applications ou des processus.

Différences :

- **Conteneur** : l'isolation s'effectue via une abstraction de ressource de l'OS[49] (hiérarchies de processus, interfaces réseau, utilisateurs…), le démarrage s'effectue en quelques secondes (une fois le conteneur distribué sur la machine). Tous les conteneurs d'une même machine partagent le même système d'exploitation.
- **Machines virtuelles** : l'isolation s'effectue par une abstraction (virtualisation) du matériel (processeur, mémoire, disques, cartes réseau…). Un système d'exploitation complet démarre, ce qui prend entre quelques secondes et quelques minutes, et consomme davantage de ressources.

[49] *Operating System* ou Système d'exploitation



Les conteneurs ont donc cette particularité de **partager le même noyau** ; la portabilité n'est possible que si le conteneur et la machine qui l'héberge partagent la même architecture (amd64, arm…) et le même système d'exploitation. Initialement développé sur Linux, Docker est également disponible sous Windows et implémente des mécanismes similaires pour produire des effets équivalents : isoler les conteneurs à la fois du système hôte et entre eux.

La particularité de Windows est de proposer deux modes de fonctionnement : faire tourner des conteneurs Windows Natifs ou démarrer des portions de système Linux virtualisé pour permettre l'exécution de conteneurs Linux.

En pratique : comment emprisonne-t-on un processus ?

Les technologies de conteneurisation permettent d'**emprisonner un processus en une ligne de commande**, mais en pratique les choses sont évidemment un peu plus compliquées. Docker par exemple, dans sa version la plus commune sous Linux, se base principalement sur **cinq composants du noyau** pour faire son travail.

Un système de fichiers dédié

Enfermer un processus (on utilisera même le terme "conteneuriser" pour décrire le phénomène) nécessite de séparer son système de fichiers de celui de l'hôte. Deux raisons principales à cela :

- Empêcher le conteneur de "voir" le reste du système de fichiers

- Lui fournir les librairies, exécutables, fichiers de configuration dont l'application a besoin pour s'exécuter (une webapp Java n'a pas les mêmes besoins qu'une base de données PostgreSQL).

Chaque conteneur dispose donc de son propre système de fichiers plus ou moins complet – qu'on appellera par la suite *rootfs* – et qui ressemble à s'y méprendre à celui de l'OS cible. Par exemple, pour les conteneurs qui s'exécutent sur Linux, on y retrouve des répertoires bien connus tels que `/etc`, `/usr`, `/bin`.

Chroot

Une fois munis de notre *rootfs*, nous pouvons commencer à y emprisonner un processus via la commande *chroot* (pour "*change root*"). Sorte de "conteneur du pauvre", c'est une commande toute simple qui a pour effet de changer la racine vue du processus qui subit le *chroot*.

C'est un début, mais avec quelques manipulations assez simples, il est possible de sortir de cette prison. L'isolation est d'autant plus partielle qu'il est possible de :

- Voir **tous les processus qui s'exécutent sur l'hôte**, et donc ceux de nos voisins

- Voir **toutes les interfaces réseau de l'hôte**

- Monter des périphériques comme la partition /dev/sda1 dans notre *chroot*, ce qui revient à **avoir accès au *filesystem* de l'hôte en entier**.

Les *namespaces*

Le noyau Linux dispose d'un système de virtualisation des ressources système : les *namespaces*. Par "ressource système" nous entendons : les utilisateurs, IDs de processus, IDs d'utilisateurs et de groupes, points de

montage, interfaces réseaux, noms de machine, communications interprocessus…

En créant des *namespaces* n'ayant qu'une vue partielle sur les ressources d'un système et en affectant des processus à ces *namespaces*, il devient possible de **masquer une partie de la réalité à un logiciel** : il ne voit pas les autres logiciels, pense qu'il n'y a qu'une interface réseau, ne voit pas l'intégralité des périphériques de la machine etc.

Les *capabilities*

Dans le monde Linux, nous sommes habitués à gérer les permissions au niveau du *filesystem*, les fameux modes présents sur chaque fichier, et qui indiquent les droits fournis au propriétaire, au groupe, et aux autres utilisateurs sur ledit fichier.

Le kernel propose également un mécanisme de permissions appellé *capabilities*. Il s'agit d'**un ensemble de rôles, chacun associé à des privilèges sur l'hôte**. L'utilisateur *root* les possède tous, et il nous est possible de les attribuer de manière discrétionnaire à un programme, ou de les lui retirer.

Dans le cas d'un conteneur, c'est ce qui nous intéresse : enlever des capacités à un processus, pour que même depuis son *chroot*, et avec sa vue limitée du système offerte par les *namespaces*, ce dernier n'hérite pas des *capabilities* de l'utilisateur qui l'a démarré.

Prenons l'exemple du logiciel Vault de Hashicorp[50], dont le rôle est de stocker toutes sortes de secrets. Pour les protéger correctement, il a besoin de pouvoir verrouiller de la mémoire pour l'empêcher d'être envoyée en swap. Avec Docker, il est possible de lui fournir cette capacité, et uniquement celle-ci :

```
$ docker run --cap-add=IPC_LOCK \
  -d --name=dev-vault vault
```
[51]

Le conteneur dev-vault, lancé par cette commande, n'a aucun autre droit sur le système que verrouiller de la mémoire à l'aide de la capacité IPC_LOCK.

Les *cgroups*

Dernière étape dans la conteneurisation : la limitation dans la consommation des ressources. Pour que les conteneurs se côtoient correctement, il est en effet important de s'assurer qu'un gourmand ne vole pas toutes les ressources aux autres. Les *cgroups* (pour *control groups*) sont une fonctionnalité du noyau permettant de limiter et de monitorer un ou plusieurs processus dans son accès aux ressources matérielles : temps CPU, quantité de mémoire utilisable, nombre d'entrées/sorties… Appliqués aux conteneurs, ils permettent de gérer chacun d'entre eux comme une unité distincte, à laquelle on alloue des ressources et des priorités. Même si un conteneur lance plusieurs processus, ils font tous partie du même *cgroup*, et ont les mêmes limites.

[50] https://www.vaultproject.io/ [51] https://hub.docker.com/_/vault/

Docker : ces petits plus qui font toute la différence

Le moteur Docker utilise massivement les capacités du noyau pour isoler et maîtriser les applications contenues. Mais ce n'est paradoxalement pas ce qui a fait le succès de Docker. C'est bien la facilitation de l'usage, associée à des à-côtés qui entourent le moteur d'exécution, qui changent la donne.

Focalisation sur l'application, pas sur le système d'exploitation

À la différence de Docker, Linux-VServer ou LXC ont pour vocation d'isoler un système d'exploitation complet. On parle de conteneurs système. L'objectif est de **reproduire** – de façon légère et rapide – (presque) **tous les fonctionnements d'une machine virtuelle** : système de démarrage des processus (sysVinit ou systemd), démarrage des démons.

Isolation système VS Isolation applicative

Conteneur système
Appli A
Processus OS
Libraries OS

Conteneur système
Appli B
Appli C
Middleware
Processus OS
Libraries OS

Conteneur système
Appli D
Processus OS
Libraries OS

Librairies et processus système
Noyau OS
Matériel

Conteneur applicatif
Appli A
Libraries OS

Conteneur applicatif
Appli A
Middleware
Libraries OS

Conteneur applicatif
Appli A
Middleware
Libraries OS

Conteneur applicatif
Appli D

Librairies et processus système
Noyau OS
Matériel

Docker, *a contrario*, annonce une ambition toute autre : faire tourner une unique application. Il est par conséquent souhaité de n'avoir qu'un processus en cours d'exécution dans chaque conteneur Docker. Les impacts de cette approche sont nombreux :

Utilisation d'un système de fichiers minimaliste, qui même s'il est basé sur une distribution classique (Ubuntu, CentOS, ou Alpine Linux pour optimiser sa taille) n'a besoin de contenir que le minimum pour exécuter l'application cible : la base du système de fichiers et un gestionnaire de paquets pour y installer les dépendances (exemple : une JVM pour une application Spring Boot). Certains langages, comme C, C++, Go[52] ou Rust[53] permettent même de créer des binaires compilés statiquement. Ils ne dépendent d'aucune librairie et ne s'appuient que sur des appels système pour fonctionner. Il en résulte des images Docker ne contenant qu'un seul fichier : l'application. Le *nec plus ultra* de la légèreté !

Gestion simpliste du démarrage des services puisqu'il se résume au démarrage d'une application. Difficile de faire plus rapide, le coût de démarrage du conteneur est négligeable, le surcoût de la conteneurisation l'est tout autant.

Conteneurs immuables. Dans l'esprit, Docker suit le principe du *rebuild* plutôt que celui d' *upgrade*. Une fois construite, l'image d'un conteneur est figée. Une mise à jour (du code de l'application, d'une librairie, d'un exécutable...) se fait en reconstruisant une nouvelle image et en remplaçant un ancien conteneur par un nouveau basé sur cette image. Il en résulte de nombreux avantages à cette approche dans la gestion des conteneurs et de leurs images.

Gestion avancée d'images

Puisque l'on parle de construire, détruire, redéployer souvent – et rapidement – des images, celles-ci doivent être faciles à manipuler. Docker prévoit plusieurs mécanismes à cet effet. Un mécanisme de **modélisation d'images par couches.** Chaque couche décrit un changement dans le système de fichiers (ajout, modification, retrait de répertoires ou de fichiers) et intervient comme un calque qui vient se superposer à la couche précédente suivant le principe du *Copy on Write*[54].

[52] https://golang.org/ [53] https://www.rust-lang.org [54] https://fr.Wikipedia.org/wiki/Copy-on-write

Système de fichiers en couches

Couche du conteneur	Couche en écriture à l'exécution
Couche 3	Image Add **Tomcat**
Couche 2	Image Add **OpenJDK**
Couche 1	Base Image **Ubuntu**
Couche 0	bootfs cgroups, namespace, devicemapper kernel

Exemple de découpage en couches d'une image

L'intérêt de ce découpage en couches est la possibilité pour deux images (ou plus) de partager les mêmes couches, réduisant ainsi l'espace de stockage nécessaire.

Couche en écriture	Couche en écriture
Image Add **Tomcat8**	Image Add **Tomcat9**
Image Add **OpenJDK8**	Image Add **OpenJDK9**
Base Image **Ubuntu**	
bootfs cgroups, namespace, devicemapper kernel	

Illustration de deux images qui utilisent les mêmes premières couches

Un système de dépôt d'images appelé une *registry*. Publique ou privée, elle est en charge d'héberger les différentes couches des images de manière versionnée. Une API (REST, qui l'eût cru) permet d'interagir avec elle : déposer des images, les lire, poser des labels... Les *registries* Docker viennent se placer à mi-chemin entre :

- Les dépôts de code comme Git : modèle de ***diff*** qui décrit des changements du système de fichiers à appliquer et référence la couche (le commit) précédent pour reconstruire toute une arborescence de changements, notions de ***labels*** / ***tags*** pour versionner les images avec des noms intelligibles par un humain.
- Les dépôts binaires historiques comme Nexus ou Artifactory dans leur capacité à stocker et versionner de nombreux objets binaires. Ces derniers ont d'ailleurs évolué pour parler désormais couramment l'API *registry* et garder leur place dans l'entreprise.

Un mécanisme de construction d'images au travers d'un format de fichier dit Dockerfile.

Celui-ci décrit les étapes à exécuter pour créer une nouvelle image à partir d'une précédente :

- Image de base sur laquelle les modifications seront ajoutées
- Commandes à exécuter (installation de logiciel par exemple)
- Fichiers à copier (fichier de configuration par exemple)
- Commande de démarrage à exécuter
- ...


```
FROM openjdk:8-jre-slim

ARG JAR_VERSION
ARG APP_NAME

LABEL version= "${JAR_VERSION}"
LABEL app= "${APP_NAME}"

EXPOSE ${PORT:-8080}

ADD build/libs/integrator- ${JAR_VERSION}.jar app.jar

WORKDIR /

ENTRYPOINT
[ "java","-Djava.security.egd=file:/dev/./urandom","-jar","app.jar" ]
```

Exemple de fichier Dockerfile, permettant de packager le jar d'une application Spring Boot dans une image Docker

Pourquoi cette technologie est-elle disruptive ?

Un nouveau paradigme de partage

À première vue, Docker semble simplement se trouver **à la confluence de la virtualisation et de l'*Infrastructure as Code*** : on se trouve en effet devant un système qui va générer des images binaires (les images Docker) à partir de code d'infrastructure (le Dockerfile). Ce serait toutefois **une vue très réductrice** de la puissance de Docker, tant la légèreté des images Docker est éloignée de l'image d'une VM classique.

Les mécanismes de Docker sont en effet optimisés pour réutiliser au mieux les images existantes et en générer de nouvelles. Dans l'exemple ci-dessus, 8 lignes de Dockerfile ont permis de transformer une image Docker Java générique en celle d'une application spécifique. La création de cette nouvelle image n'aura pas pris plus de quelques secondes, et le résultat pèse à peine quelques méga-octets. Un bug ? Générer une nouvelle image corrigée ne prendra pas plus de temps. Un *framework* Java spécifique ? Générer une nouvelle image Docker l'intégrant et la distribuer sera aussi simple, permettant aux utilisateurs de l'intégrer à leurs projets tout aussi facilement.

Appliquer la même approche sur des images de machines virtuelles nécessiterait des processus lourds : démarrer une image Linux, se connecter dessus en SSH, copier le code d'infrastructure à l'intérieur puis le lancer, et repackager l'image. Le processus prendrait plusieurs minutes, pour un résultat de plusieurs gigaoctets. Cette lourdeur fait qu'en pratique tout le monde[55] préfère générer un tout petit nombre d'images de VMs basiques sur lesquelles on va travailler ensuite en *Infrastructure as Code*.

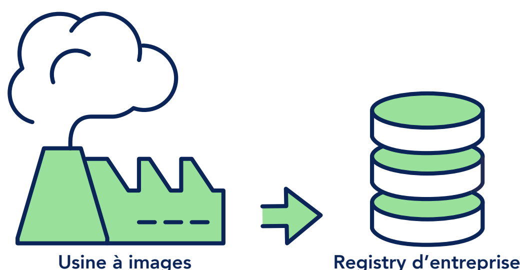
Ce changement d'approche apporté par les conteneurs va **décupler les capacités de partage et de réutilisation entre équipes**, mais va évidemment **impacter les processus** : architecture, conception, construction et déploiement vont être refondus pour s'adapter à la rupture apportée par Docker.

[55] À part Netflix, qui a un tel volume de VMs à déployer que ce modèle est viable pour eux : https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15

La construction des images au cœur de l'intégration continue de l'entreprise

Le changement du code applicatif, comme l'installation d'un correctif de l'une des dépendances de l'application (patch de la JVM par exemple) conduit inexorablement à la reconstruction de l'image. Si comme on l'a vu cette reconstruction est particulièrement simple, l'intégration des couches OS, *middleware* et applicatives dans une même image va forcément déplacer les périmètres de responsabilité. Les Dev et les Ops doivent donc parvenir à trouver **un modèle de collaboration** pour garantir la dualité Dev/Ops intrinsèque au contenu des images Docker.

Tout se prête à l'automatisation du processus de construction de l'image. L'intégration continue garde tout son sens. En plus de ses fonctions classiques, elle se voit désormais en charge de tâches supplémentaires : analyse de vulnérabilité des images, publication dans une *registry*, signature d'images, pose de labels..



Les applications doivent être repensées ou adaptées

En plus des impacts déjà abordés sur la construction d'applications (liés au caractère immuable des conteneurs par exemple), la conteneurisation d'applications implique d'autres conséquences.

La règle du jeu est simple : pour une application donnée, un nombre variable de conteneurs (au minimum 2) vont être instanciés, et de façon très dynamique :

- Les conteneurs peuvent être amenés à mourir (crash de l'application, du nœud qui héberge le conteneur) : ils doivent pouvoir être ré-instanciés sans autre forme de procès. L'infrastructure est faillible[56], ***Design for Failure*** est notre credo.
- Les conteneurs vont être volontairement tués et ré-instanciés (dans de nouvelles versions) pour appliquer des mises à jour (dans le code ou les dépendances techniques, comme nous l'avons déjà dit, mais aussi dans la configuration si celle-ci ne peut s'appliquer à chaud)
- Des conteneurs supplémentaires seront ajoutés (ou supprimés) pour absorber des pics de charge (scalabilité horizontale)
- Des conteneurs supplémentaires dans des versions spécifiques vont être ajoutés pour tester un nouveau comportement sur un échantillon d'utilisateurs (*Canary Release*[57])

Voici donc les principales caractéristiques attendues de ces applications. La littérature emploie les termes de 12-*factors applications*[58] ou de *cloud-natives applications*.

Lorsqu'elles suivent ces préconisations, elles peuvent tirer le meilleur parti du *cloud*, et deviennent, mécaniquement, les premières éligibles à la conteneurisation. À noter que si elles sont nécessaires pour la conteneurisation, elles conservent tout leur sens dans des environnements sans conteneur.

Arrêtons-nous quelques instants sur les principales pratiques que nous recommandons et que nous tentons de mettre en œuvre dès que possible. N'hésitez pas à consulter nos publications[59] qui sont bien plus exhaustives sur le sujet !

Applications dites *Stateless* : les applications conteneurisées doivent être jetables et donc sans état. Terminées les affinités de session sur les *load-balancers* et le stockage local des sessions Web. C'est à ce prix que l'on obtient une bonne tolérance à la panne et une scalabilité horizontale efficace. Par exemple, s'il y a besoin de stocker des informations de session, elles le seront dans un cache distribué (tel que Redis par exemple).

Séparation du code et de la configuration : les images Docker doivent être construites suivant le principe du ***build once, run everywhere***. Il doit être possible d'injecter simplement de la configuration dans le conteneur, via variables d'environnement ou fichiers de configuration qui surchargent les valeurs par défaut.

Démarrages rapides, tolérants et arrêts gracieux : les applications doivent être pensées pour être proprement et fréquemment arrêtées et redémarrées. La technologie de conteneurisation promet un démarrage technique très rapide, mais n'est d'aucune utilité si l'application met ensuite 3 minutes à s'initialiser. L'application doit également être résistante à un démarrage désordonné de ses dépendances. Dans un écosystème massivement distribué, il est en effet difficile de garantir qu'une base de donnée, un cache ou une autre application sera bien présente avant le démarrage d'une application. D'un point de vue plus applicatif, la mise en œuvre de mécanismes de type *Graceful Degradation*[60] ou *Circuit Breaker*[61] est préconisée pour gérer l'indisponibilité de services secondaires.

[56] Au même titre que les Illusions de l'Informatique Distribuée [https://fr.Wikipedia.org/wiki/Illusions_de_l%27informatique_distribu%C3%A9e], les infrastructures tombent en panne. Les systèmes doivent donc accepter cet état de fait et être conçus pour tolérer la panne [57] https://martinfowler.com/bliki/CanaryRelease.html [58] https://12factor.net/fr/ [59] https://blog.octo.com/cloud-ready-applications/ [60] Adaptation automatique de l'application à une situation dégradée (perte d'une partie de l'infrastructure, taille d'écran de l'utilisateur réduite, fonctionnalité non supportée par le navigateur cible, bande passante réduite, etc.). [61] En fonction d'un certain nombre de critères d'erreur (*timeout*, nombre d'erreurs), ce *pattern* permet de désactiver l'envoi de requêtes au Service 2 et de renvoyer immédiatement une réponse alternative. Il agit comme un proxy implémentant une machine à états (Ouvert, Passant (fermé), Semi-ouvert) pour l'apprentissage de l'état du service.

Présence d'URL de *healthcheck*. Aussi appelées lignes de vie, elles permettent de refléter un état de fonctionnement le plus précis possible de l'application. Généralement exposées en HTTP, elles permettent d'aller bien au-delà de la simple présence d'un processus sur une machine ou un port en écoute sur un serveur.

Présence d'URL de métriques. Chaque application a pour mission d'exposer (à nouveau en HTTP de préférence) un jeu de métriques pour aider à sa supervision. Parmi les métriques exposées, on cherche à la fois à exposer des métriques techniques liées aux serveurs d'application (Tomcat), à la plateforme d'exécution (JVM) ou au langage (Java), mais aussi à exposer des métriques métier : taux de transformation du tunnel de vente, 95 centiles des temps de réponses des appels à une API…

Des *logs* produits simplement. Tout comme pour les points précédents, Docker cherche la simplicité. Plutôt que d'écrire des *logs* dans des fichiers, Docker attend simplement des applications qu'elles produisent des *logs* sur les sorties et erreurs standard, et ne prennent en aucun cas en charge leur écriture dans un fichier (et *a fortiori* pas sa rotation), ni leur transmission vers un puits de journaux centralisé. C'est le travail de la technologie de conteneurisation de s'en charger, et le développeur du conteneur n'a plus à y réfléchir.

La situation à date

Prod-ready ?

À l'heure où nous écrivons ces lignes, force est de constater que Docker a passé l'épreuve de la production. Certains villages gaulois résistent encore et toujours à l'envahisseur, mais progressivement les freins sont en train de disparaître : maturité de la technologie, sécurité, professionnalisation de l'écosystème, préparation au monde de la production (monitoring, administration…).

La clusterisation est mature

Après une longue période d'incertitude, et à l'issue d'un combat acharné[62] entre belligérants cherchant à gérer une flotte de conteneurs à l'échelle, le marché est en train de se consolider autour d'offres de plus en plus matures : Kubernetes, ECS, Fargate en sont quelques unes. C'est principalement autour de Kubernetes que se portent aujourd'hui tous les regards. Il offre en effet une solution crédible, réversible (à la fois sur le *cloud*, managée ou non et *on-premise*), et avec un niveau d'abstraction permettant de modéliser des architectures riches.

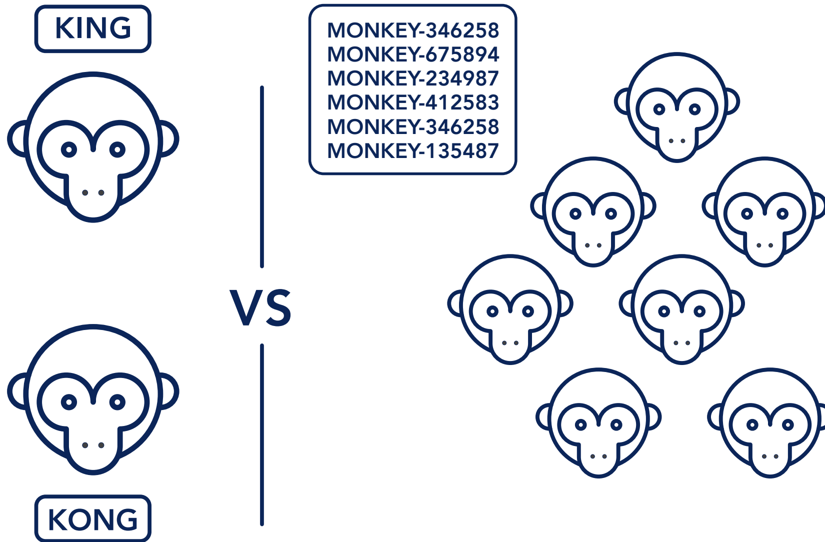
Les conteneurs à états (persistants) en ligne de mire

Si la question des conteneurs applicatifs sans état dits *cattle* n'est plus un sujet, il reste l'épineuse question des conteneurs de type *pet*[63]. Elle est infiniment plus complexe à traiter et les travaux, même s'ils sont déjà largement engagés (gestion de volumes de stockages persistants, concepts de groupes de conteneurs à état dits *StatefulSets*[64]) impliquent toujours une très forte intégration entre les applications concernées et les orchestrateurs de conteneurs. L'histoire reste donc à écrire et elle s'annonce passionnante !

---

(62) Revivez cette épopée sur notre blog : Épisode 1 : https://blog.octo.com/docker-en-production-la-bataille-sanglante-des-orchestrateurs-de-conteneurs/ Épisode 2 : https://blog.octo.com/containerus-bellum-ou-la-chronique-des-hostilites-dans-lecosysteme-docker/ (63) (lit. animaux de compagnie vs. bétail). Les serveurs ne sont plus nommés "King" ou "Kong" mais par un identifiant unique (monkey-xxx). Cette approche industrielle de gestion d'environnements consiste à ne plus nommer, réparer un serveur et le garder longtemps (*Pet*), mais à systématiquement le détruire et le recréer en cas de problème (*Cattle*).
(64) https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

Pet VS Cattle



Conclusion

LES RAISONS QUI NOUS FONT AIMER CETTE TECHNOLOGIE

- Une masse de nouveaux paradigmes qui facilitent la gestion, l'automatisation, le paramétrage, le déploiement, la résilience, la scalabilité des applications
- Une légèreté à l'exécution

LES LIMITES DE CETTE TECHNOLOGIE

- Ce n'est qu'un outil. Mal employé, il ne résoudra pas les problèmes organisationnels. Attention aux fantasmes cachés derrière cette technologie !
- La gestion des conteneurs à états n'en est qu'à ses balbutiements

CE QUI FAIT QUE CETTE TECHNOLOGIE TRANSFORME NOTRE MÉTIER

- Les domaines de responsabilité sont rebattus
- Il y a une réelle nécessité de repenser les applications et les processus pour utiliser correctement les conteneurs

Le Cloud

À la croisée de toutes les thématiques abordées précédemment se trouve enfin ce fameux concept protéiforme qu'est le cloud. Et c'est bien parce que tous ces concepts (API, open source, SDx, Infrastructure as Code, applications cloud-natives voire conteneurisées...) sont parfaitement assimilés et combinés que l'usage du cloud permet effectivement de déplacer des montagnes. Le cloud sans une API n'est rien, le cloud sans Infrastructure-as-Code n'est que l'ombre de lui-même, le cloud sans ...Bref, vous avez compris le principe.

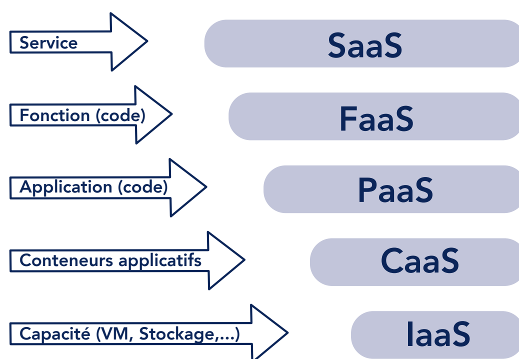
Introduction

Nous ne ferons pas l'affront d'expliquer dans cet ouvrage ce qu'est le *cloud*. Des années de littérature ont déjà pavé tout ce qui pouvait être écrit. Synthétisons simplement quelques points saillants qui le définissent.

Les caractéristiques du Cloud Computing telles que décrites en 2011.

Le NIST (*National Institute of Standards and Technology*), institut indépendant américain, définit le *Cloud Computing* comme étant "une nouvelle façon de délivrer les ressources informatiques et non une technologie". Dans la publication du NIST[65], sont définies cinq caractéristiques des services *cloud* :

- Un service en libre service et à la demande : il doit être accessible par simple demande automatisée, sans interaction humaine
- Un accès ubiquitaire au réseau : le service doit être disponible sur le réseau par tous, via des protocoles standards (REST…)
- Une mutualisation des ressources : ces dernières sont partagées et attribuées dynamiquement aux clients
- Une élasticité rapide : les capacités du service sont adaptées en fonction des besoins



- Un service mesurable : le service doit être mesurable tant en termes d'usage que technique

Une offre de services basée sur une fusée à plusieurs étages.

Même s'il existe un relatif consensus sur l'appellation des types de services *cloud*, les périmètres de chacun sont loin d'être limpides, avec de magnifiques zones floues, particulièrement autour de la définition du PaaS.

Attardons-nous juste quelques instants sur les derniers arrivés dans le modèle :

FaaS (***F****unction as* ***a*** ***S****ervice*).

L'objectif est de pouvoir exécuter des fonctions (des "bouts de code") directement sur le *cloud*, sans avoir besoin de provisionner de machines virtuelles, de conteneurs

ou... de plateformes ! Simplement avec du code source. Ces fonctions peuvent être déclenchées soit par des événements (liés à la plateforme *cloud* elle-même ou applicatifs), soit manuellement. Si du code *stateless*[66] est écrit, plusieurs "instances" de la fonction peuvent être lancées simultanément. Cela permet d'**adapter la quantité de code** en train de tourner **par rapport à la quantité d'événements déclencheurs**. Étant donné le fait que la facturation est faite par rapport au temps d'exécution des fonctions, nous parlons vraiment d'une expérience "*Pay only for what you use*" : une fois que la fonction a fini son travail, son instance est terminée, et la facturation s'arrête.

Exemples : Amazon Lambda, Google Cloud Functions, Azure Functions, Fission.

CaaS (*Container as a Service*).
Cette appellation désigne les offres d'exécution de conteneurs Docker. L'API exposée pour piloter ces orchestrateurs de conteneurs est soit basée sur un standard du marché (Swarm, Kubernetes, Mesos…), soit spécifique au *cloud provider*. Certaines offres proposent un modèle de facturation basé sur les machines virtuelles utilisées pour constituer un cluster. D'autres offres basent leur modèle directement sur le nombre de conteneurs à exécuter et les quotas de ressources qui leur sont associés.

Exemples : Amazon EC2 Container Service, EKS, Fargate, Google Container Service, Azure Container Service.

La promesse du *cloud* est donc la mise à disposition d'un catalogue de services exceptionnel, le tout, avec un temps de mise en œuvre proche de zéro et avec uniquement quelques centaines ou milliers d'euros en poche pour démarrer. Il n'est plus nécessaire d'avoir son propre *datacenter* pour être sur le marché et rivaliser avec les acteurs déjà en place. C'est par essence le paradis des startups et plus généralement des initiatives digitales des entreprises : aller vite[67], se confronter rapidement au marché, pivoter si besoin est. *Time-to-Market first.*

"La promesse du cloud est donc la mise à disposition d'un catalogue de services exceptionnel, le tout, avec un temps de mise en œuvre proche de zéro et avec uniquement quelques centaines ou milliers d'euros en poche pour démarrer"

[65] http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf [66] *Stateless* : sans état, qui peut s'exécuter indépendamment, sans avoir recours à une session. [67] Mais aussi échouer vite (*fail fast*). Quitte à se tromper, autant le faire le plus vite possible avec le moins d'investissement possible.

"Élève en progrès, mais peut mieux faire"

Nous sommes donc en théorie en possession de l'**outil le plus puissant qui soit** :

- Rapidité de mise en place
- Coût de démarrage quasi-nul
- Élasticité des infrastructures pour optimiser la facture en fonction de l'usage, avec une capacité à réagir aux saisonnalités, aux succès aussi brusques qu'inattendus d'un produit
- Automatisation de bout en bout de tous les processus (construction de l'application, déploiement, administration/exploitation, maintien en condition opérationnelle)
- Tout ça avec une disponibilité proche des fameux 100 %, grâce à l'implantation des *cloud providers* sur plusieurs régions et plusieurs zones de disponibilité, combinée à une architecture applicative résiliente et scalable

Et pourtant, malgré un potentiel qui rend l'usage du *cloud* théoriquement immédiat et infini, **notre constat est tout autre**.

Le débordement "transparent" existe-t-il vraiment ?

Le *cloud* public n'est que très rarement utilisé comme une stratégie de débordement "transparent". L'idée qu'une application puisse simplement et efficacement passer à l'échelle en ajoutant des VM sur le *cloud* pour venir épauler celles déjà présentes dans le SI (*cloud* privé ou VM classiques) reste de **l'ordre de la mythologie**. Et c'est pourtant un des *use-cases* majeurs qui est mis en avant dans la littérature. Ce que nous constatons plus généralement est **l'utilisation du *cloud* pour initier les projets**, avec une tendance à **ré-internaliser les plateformes** à l'approche de la mise en production.

Ségrégation par environnements (réalité)

DATACENTER PRIVÉ

Environnement de production

Lien Sécurisé

CLOUD PUBLIC

Environnement de développement

On parle du déplacement complet d'une application du *cloud* vers le SI interne (ou dans l'autre sens), mais rarement d'une application en mode hybride : à la fois sur les deux mondes de façon transparente.

Débordement IaaS et CaaS (mythe)

DATACENTER PRIVÉ

CLOUD PRIVÉ

Applications Legacy

Applications Cloud Ready **IaaS / CaaS**

Lien Sécurisé

Extension du datacenter

CLOUD PUBLIC

Applications Cloud Ready **IaaS / CaaS**

Le lâcher-prise vis-à-vis des services managés

Utiliser des services managés est certes plus cher sur la facture du fournisseur *cloud*, mais plus économe en temps passé aux (basses) besognes de gestion dudit service : déploiement, mise à jour, maintien en condition opérationnelle. Pour réellement profiter de ces services, il est nécessaire de faire preuve d'un certain lâcher-prise en acceptant le modèle du *cloud provider*. C'est lui qui fixe les règles du jeu en termes de stratégie de montée de version, du nombre de versions supportées… Le laxisme que l'on a pu s'autoriser sur les SI d'entreprise n'est plus applicable. Il devient très compliqué, voire impossible de conserver des versions obsolètes de logiciels (systèmes d'exploitation, versions de *middlewares*) si le *cloud-provider* ne les supporte plus.

C'est le modèle de responsabilité partagée, entre autre, qui tire ce principe. Le fournisseur étant responsable de l'infrastructure d'un service managé, il s'octroie le droit de ne gérer qu'un nombre limité de versions, privilégiant les plus récentes.

L'ombre du *vendor lock-in*

Il pourrait être séduisant de systématiquement opter pour les offres de services à plus forte valeur ajoutée. C'est en effet un excellent levier pour aller plus vite, avec le minimum de frais de *build*.

Toutefois, ce choix vient avec un coût supplémentaire, plus sournois et difficile à évaluer : **la difficulté (voire l'impossibilité) de changer d'hébergeur**. Deux critères doivent être évalués :

- **Le choix des outils de déploiement, et d'administration des applications**. Se baser sur la solution de *monitoring*, de déploiement, de sécurité spécifique à un *cloud-provider* conduira à la nécessité de ré-écrire des pans entiers d'outillage lors du changement d'hébergeur.

- De la même façon, **l'utilisation de services complémentaires spécifiques** (cache, bases de données managées, stockage objet, etc) nécessitera un travail conséquent de réécriture des applications.

L'un des challenges importants est donc d'**être capable d'arriver à trouver le bon équilibre** entre la quantité de services dits génériques à toutes les plateformes et les services spécifiques à la plateforme utilisée, sachant qu'un niveau d'abstraction plus élevé entraîne mécaniquement une capacité de changement plus réduite.

À titre d'exemple, évoquons les offres *FaaS* ou *Serverless*. Celles-ci impliquent presque mécaniquement une adhérence très forte au fournisseur de services : dans les SDK ou *frameworks* utilisés, mais également dans le modèle d'intégration qu'elles proposent (bus de messages spécifiques, type d'événements spécifiques…). Une solution peut être alors de mettre en œuvre un *framework* portable multi-*cloud provider*, sachant que la portabilité implique malheureusement de se contenter du plus petit dénominateur commun.

"Chez nous, c'est pas pareil"

C'est une maladie très classique – dans les grands groupes en particulier – de faire du "pareil, mais pas pareil[68]". Cette propension à utiliser une application ou un service à la limite – voire en dehors – de son cadre est un frein à son passage sur le *cloud*. Elle interdit en effet de s'appuyer sur des services de haut niveau et contraindra à simplement reproduire sur le *cloud* – au travers de l'offre de plus bas niveau : le IaaS – l'architecture (et donc ses spécificités) en mode *Lift and Shift*[69]. Peu de gain au final et une lourdeur dans la gestion qui se rapproche des modes d'hébergement historiques. Nous invitons chaleureusement ici notre lecteur à **toujours revenir aux usages standards des applications, *frameworks* et autres services** ; c'est à la fois un gage de portabilité et de réversibilité, mais aussi la promesse qu'il sera possible de s'appuyer massivement sur des services managés standard.

"Cette propension à utiliser une application ou un service à la limite – voire en dehors – de son cadre est un frein à son passage sur le cloud. "

Faire du vieux avec du neuf

Les machines virtuelles sur le *cloud* sont (encore trop souvent) **gérées comme des serveurs physiques**. Ce modèle de consommation très statique est un frein majeur à l'optimisation économique et à l'optimisation de la gestion des machines (déploiement notamment). Trop peu d'entreprises adoptent par exemple une stratégie industrielle de gestion de l'arrêt des VM de développement pendant la nuit. Rares sont celles qui suivent l'approche des infrastructures immuables[70] comme une stratégie de gestion des machines virtuelles. Au même titre que les conteneurs, des modèles comme le *blue/green deployment*[71] s'appuyant sur la re-création de nouvelles machines plutôt que leur mise à jour peut apporter énormément de rapidité, maîtrise et simplicité. De notre analyse, cela relève plus de vieilles habitudes qui ont la vie dure que d'une réelle impossibilité technique. À partir du moment où l'*Infrastructure as Code* est en œuvre à tous les niveaux, il est même envisageable de détruire bon nombre de ressources tous les soirs pour mieux les re-créer au petit matin.

[68] Lisez cet article avant de partir sur du non-standard : https://blog.octo.com/plaidoyer-pour-les-configurations-par-defaut/ [69] Lift and Shift est une stratégie de cloudification qui consiste simplement à porter sans modification une application ou un service. Cette stratégie est la plus simple car elle permet de ne pas devoir repenser l'architecture, les offres de services à consommer. [70] Tel que décrit dans le chapitre relatif à la conteneurisation [71] Le déploiement *blue/green* permet de déployer une nouvelle version d'un système sans interrompre le bon fonctionnement du service. https://blog.octo.com/zero-downtime-deployment/

La loi de Conway conduit à reproduire les modèles historiques sur le *cloud*

C'est finalement à nouveau la Loi de Conway[72] qui revient jouer les trouble-fêtes, telle une implacable fatalité qui nous poursuit jusqu'au fin-fond des *datacenters* de nos très chers *cloud providers*. Reproduire une organisation silotée telle qu'elle peut l'être dans nos SI historiques aura irrémédiablement la même conséquence : ralentir nos réalisations – même sur le *cloud* – en dissociant les objectifs de chaque équipe. Nous n'allons pas ici répéter le propos tenu dans le premier opus de cette trilogie, mais simplement rappeler cette vérité : **la question de l'organisation s'applique tout autant sur le *cloud***. Son impact négatif sera d'autant plus visible que l'on sait pertinemment que sur le *cloud*, l'approvisionnement de la plupart des ressources se fait en quelques secondes, quelques minutes au maximum…

Si vous songez à mettre votre *cloud* derrière un outil de *ticketing*, sous la responsabilité d'un centre de service, pour en maîtriser ses usages, alors, vous ne tirerez jamais parti des réels avantages du *cloud*. Où est l'élasticité ? L'autonomie ? Le libre-service ? L'agilité ? La simplicité ? La responsabilité ?... Vous n'avez, au final, que déplacé vos serveurs dans le *datacenter* de quelqu'un d'autre.

Tuer le mythe du calcul simple du ROI

Le **calcul prédictible d'une facture *cloud*** [73] **relève de l'utopie.** Les modèles de facturation impliquent de connaître très finement – et *a priori* – tout un tas d'indicateurs qui n'étaient historiquement pas ou peu mesurés (débit sortant sur les réseaux, quantité de données échangées entre zones de disponibilité ou régions, taux de changement des données sur les disques, etc.). Aujourd'hui quasiment aucun acteur ne tente plus de prévoir la consommation de services *cloud*. Au lieu de ça, la démarche majoritairement appliquée est basée sur **une période d'observation des coûts**, suivie, si le besoin se fait sentir, d'**une phase d'optimisation** (engagement de location d'instances à l'avance pour les payer moins cher[74], enchères sur le prix des instances[75] par exemple, ou architectures alternatives).

Oubliez les vieux réflexes ; remettez le travail d'architecture au centre de la démarche d'adoption du *cloud*

Rares sont les politiques de sécurité qui passent sereinement l'épreuve du *cloud*. Entre laxisme outrancier et modèle de défiance systématique, il est difficile de placer le curseur tout en dépassionnant les débats. C'est donc par un **travail d'architecture** tout à fait classique – même s'il s'appuie sur un catalogue de services

[72] les organisations qui définissent des systèmes... sont contraintes de les produire sous des designs qui sont des copies de la structure de communication de leur organisation." Voir notre premier tome *Culture DevOps Vol.01*. [73] Voir à ce sujet le calculateur d'AWS : https://www.awstcocalculator.com/ [74] · *Reserved Instances* chez AWS et Azure, *Committed Use* chez Google par exemple. [75] · *Spot Instances* chez AWS, *Preemptible Instance*s chez Google,...

cloud bien plus vaste qu'auparavant – que doit se faire le travail de recueil d'exigences (fonctionnelles, techniques, sécuritaires...), d'analyse de scénarios, d'arbitrage, de recherche d'optimum… C'est aussi au travers de cette démarche qu'il sera possible d'y voir clair **entre promesses et réalité**.

Bien que l'on parle de migration vers le *cloud*, il faut en réalité voir une réelle **re-conception de nos manières de penser et de faire nos infrastructures et nos applications**. Le métier nouveau d'Architecte *cloud* est au cœur de cette transformation : il incarne à la fois la rupture avec l'ancien monde, mais également les *trade-offs* nécessaires à la bonne cohabitation entre l'ancien et le nouveau.

"Rares sont les politiques de sécurité qui passent sereinement l'épreuve du cloud..."

Le *cloud* n'est pas en soi un prérequis au digital, mais il coche sans difficulté toutes les cases d'un produit "compatible DevOps". *API First, Software Defined, Infrastructure As Code…*
Le *cloud* est le porte-drapeau du DevOps.

Et à part ça ?

Les pratiques DevOps n'ont pas fini d'évoluer au rythme des nouvelles innovations technologiques. À la fois source de réels progrès, l'adoption future de ces technologies va irrémédiablement entraîner des ***remises en cause de nos modes de fonctionnement actuels****. Blockchain, re-spécialisation des machines et des processeurs (ARM, GPGPU*[76]*), IoT*[77]*, ordinateurs quantiques, unikernels*[78]*, la liste est longue. Voici quelques exemples de technologies qui vont – probablement – venir semer le trouble dans nos pratiques. Attardons-nous sur trois tendances pour illustrer notre propos.*

[76] *General-purpose processing on graphics processing units* : utiliser la puissance de calcul spécialisée (nombres flottants, parallélisme...) de la carte graphique pour des usages qui ne sont pas forcément du rendu graphique [77] Internet des Objets : https://fr.Wikipedia.org/wiki/Internet_des_objets
[78] "Les *unikernels* sont des images systèmes spécialisées, créées en utilisant des systèmes d'exploitation bibliothèques où tous les processus partagent le même espace mémoire. Le développeur sélectionne, à partir d'un ensemble modulaire, un sous ensemble minimum de bibliothèques qui correspondent aux services du système d'exploitation nécessaires à l'exécution de son application. Ces bibliothèques sont alors compilées avec l'application et des configurations pour créer des images fixes, à but unique, qui fonctionnent directement sur un hyperviseur ou sur du matériel sans intervention d'un système d'exploitation tel que Linux ou Windows." (source : Wikipedia)

Crypto everywhere

"***Encrypt everything***" est devenu le mot d'ordre sur internet. Les révélations régulières des lanceurs d'alerte ont montré l'importance du chiffrement des flux, et les acteurs du web poussent à une sécurité toujours plus avancée afin de maintenir la confiance nécessaire au business sur internet.

Mozilla et Google ont annoncé via leurs navigateurs respectifs que les sites non chiffrés doivent disparaître[79] et Google booste déjà les sites chiffrés dans ses résultats de recherche[80]. HTTP/2 ne sera probablement pas utilisable sans chiffrement, et les standards de sécurité poussent de plus en plus à tout protéger : disques durs des serveurs, et surtout l'ensemble des flux internes des plateformes.

Au-delà du web, en Europe, c'est la réglementation GDPR[81] qui entérine la nécessité de chiffrement des données dans tous ses états (*at rest, in use, in transit*). L'entreprise est désormais responsable légalement de la protection par design (*privacy by design*) et par défaut des données de ses utilisateurs. Si prise à la légère, cette responsabilité peut coûter chère : "20 000 000€ ou, dans le cas d'une entreprise, **jusqu'à 4 % du chiffre d'affaires annuel mondial total** de l'exercice précédent, le montant le plus élevé étant retenu"[82].

"Si cette poussée de la cryptographie a du bon, elle va remettre en question certaines pratiques dans la gestion de l'infrastructure. "

Si cette poussée de la cryptographie a du bon, elle va remettre en question certaines pratiques dans la gestion de l'infrastructure.

En premier lieu, la génération et le suivi des secrets (certificats, mots de passe...) va devoir passer à l'échelle. Les demandes manuelles ne seront pas viables quand il faudra renouveler cinq cents certificats à la fois, sans parler de leur déploiement !

Ensuite, le trafic devient naturellement opaque et impossible à lire de l'extérieur. Si c'est bien là le but recherché, cela va aussi rendre aveugles certains équipements de sécurité comme les sondes de détection d'intrusion[83] (IDS[84], IPS[85]), à moins de leur fournir l'ensemble des clés privées utilisées dans l'infrastructure.

Enfin, n'oublions pas les difficultés pour l'administrateur à investiguer des pannes quand tous les flux sont chiffrés (*tcpdump* ne montre plus grand-chose), sans compter la complexité supplémentaire dans la configuration des serveurs pour gérer le chiffrement.

Alors, va-t-on devoir embaucher une armée de stagiaires pour renouveler des certificats ? Bien sûr que non. Plusieurs groupes avancent sur l'**automatisation des PKIs**[86] et de bonnes solutions existent déjà. *Let's Encrypt*[87] permet de générer et renouveler automatiquement des certificats pour des sites web publics, et les solutions pour **monter une PKI interne automatisée**[88] se multiplient : Vault[89], Conjur[90], CFSSL[91]…

Le débogage de problèmes de PKI reste en revanche encore un art pour initiés, basé sur d'obscures lignes de commande OpenSSL et une connaissance fine des certificats et des protocoles. Vus les enjeux sur ce sujet, gageons cependant que l'outillage nécessaire ne va pas tarder à arriver !

(79) https://blog.mozilla.org/security/2015/04/30/deprecating-non-secure-http/ et https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html (80) https://webmasters.googleblog.com/2014/08/https-as-ranking-signal.html. (81) General Data Protection Regulation, ou RGPD (Réglement Général de Protection des Données) en français : http://eur-lex.europa.eu/legal-content/FR/TXT/HTML/?uri=CELEX:32016R0679 (82) Article 83 du Règlement n°2016/679 du Parlement Européen et du Conseil du 27 avril 2016 (83) Certaines industries s'alarment notamment de la disparition de l'algorithme RSA dans la prochaine version de TLS : https://www.ietf.org/mail-archive/web/tls/current/msg21275.html (84) *Intrusion Detection System*: "Un système de détection d'intrusion (ou IDS: *Intrusion Detection System*) est un mécanisme destiné à repérer des activités anormales ou suspectes sur la cible analysée (un réseau ou un hôte). Il permet ainsi d'avoir une connaissance sur les tentatives réussies comme échouées des intrusions."(source : Wikipedia) (85) *Intrusion Prevention System* : "Un système de prévention d'intrusion (ou IPS, intrusion prevention system) est un outil des spécialistes en sécurité des systèmes d'information, similaire aux IDS, permettant de prendre des mesures afin de diminuer les impacts d'une attaque. C'est un IDS actif, il détecte un balayage automatisé, l'IPS peut bloquer les ports automatiquement. Les IPS peuvent donc parer les attaques connues et inconnues. Comme les IDS, ils ne sont pas fiables à 100 % et risquent même en cas de faux positif de bloquer du trafic légitime." (source : Wikipedia) (86) "Une infrastructure à clés publiques (ICP) ou infrastructure de gestion de clés (IGC) ou encore *Public Key Infrastructure* (PKI), est un ensemble de composants physiques (des ordinateurs, des équipements cryptographiques logiciels ou matériel type HSM ou encore des cartes à puces), de procédures humaines (vérifications, validation) et de logiciels (système et application) destiné à gérer les clés publiques des utilisateurs d'un système."(source : Wikipedia) (87) https://letsencrypt.org/ (88) Rappelez-vous, pilotable par un API ! (89) https://www.vaultproject.io/ (90) https://www.conjur.org/ (91) https://github.com/cloudflare/cfssl

L'IA partout ?

L'IA sera la prochaine révolution dans l'IT. C'est du moins ce que nous annoncent les prédicateurs. D'après Kevin Kelly[92], cette révolution se passera sur le même modèle que le numérique ("nous avons numérisé <votre activité préférée>") ou le digital ("nous avons digitalisé <le service de votre choix>"). Les prochaines années verront fleurir des milliers de startups avec pour elevator pitch : "voici <le produit de votre choix> avec de l'IA".

Et chez nous, OPS, que peut-on faire de l'IA ?

Nous voici donc avec notre activité, l'OPS, et notre technologie disruptive. À nous de trouver un cas d'usage. Avant de s'affoler devant la possibilité de nous faire remplacer par des machines, bon nombre de sujets vont changer avec l'explosion des données, du *Machine Learning*[93] et du *Deep Learning*[94] Et des données, nous en possédons énormément dans notre infrastructure, ne serait-ce que dans nos serveurs de supervision. Souvenez-vous, ces *logs* qui saturent vos disques durs… OK, mais que pouvons-nous en faire ?

"Nous voici donc avec notre activité, l'OPS, et notre technologie disruptive. À nous de trouver un cas d'usage."

Prédire les pannes

Pouvoir anticiper la chute de son cluster de serveurs en avance, nous savons déjà le faire notamment quand il s'agit de suivre une courbe linéaire de remplissage de disque. Parfois, les problèmes sont plus complexes, et vont réussir à déjouer notre surveillance pour faire tomber un à un tous nos serveurs. Nous ferons l'analyse de la situation après la panne, et trouverons une nouvelle valeur à ajouter à notre outil de supervision. Nous ne nous ferons pas avoir deux fois. **Et si, avec nos données de supervision nous nous construisions un *dataset*[95] ?** Avec ce dernier, nous pourrons entraîner différents modèles de *Machine Learning* et trouver des dépendances entre les différentes variables. Il s'agira de les ajouter à notre outil de supervision. Dès lors, nous serons plus précis et nous pourrons **anticiper les pannes sur notre cluster.**

---

[92] *The Inevitable*, 2016, éditions Viking Press [93] *Machine Learning* : apprentissage automatique basé sur un ensemble de données et de modèles mathématiques. [94] *Deep Learning* : sous ensemble du *machine learning* qui consiste en un empilement de réseaux de neurones. [95] *Dataset* : jeu de données.

Détecter les signaux faibles d'une attaque

Une attaque bien exécutée est une attaque discrète. L'analyse à froid d'une intrusion est longue, et n'offre pas de garantie de résultat. La masse de données nous empêche de nous focaliser sur certains signaux faibles, un accès non autorisé, des appels d'API inhabituels, un DDOS en cours...

L'IA est en mesure d'intervenir dans ces cas pour **faire ressortir des profils d'attaques là où d'autres formes d'analyse s'écroulent** sous la quantité de données à traiter. Avec *GuardDuty*[96], Amazon propose déjà un outil de détection de menace intelligent qui s'appuie sur l'apprentissage des milliards d'événements existants dans nos comptes AWS. Chez Azure, *l'Azure Security Center* propose la même approche[97].

Gérer nos SLO/SLA

Ce que recherche un utilisateur, c'est un service de qualité, réactif et disponible. Cette qualité doit être mesurable pour déterminer les SLO pour lesquels nous sommes engagés. Il peut être question de ne jamais rendre un service au-dessus des 100ms, ou de ne pas passer en-dessous 99,99 % de disponibilité. Connecter tous nos éléments de mesure à notre intelligence artificielle nous permettra d'anticiper les fluctuations, en fonction de la charge ou d'autres éléments que nous ne connaissons/maîtrisons pas encore. L'IA se chargera de mettre en mode dégradé les services secondaires, non éligibles à notre SLA, ou bien d'ajouter de nouvelles ressources à notre cluster. **En vérité, peut-être choisira-t-elle de répondre à l'urgence d'une façon que nous ignorons aujourd'hui**, afin de tenir nos engagements.

Le futur, c'est maintenant

Sans parler des données relatives à la supervision, nous imaginons aussi des **infrastructures intelligentes**, qui pourront déployer du code en production sans configuration amont de la part d'un humain. Nous ne nous poserons plus les questions suivantes : quelle version de conteneur PHP vais-je déployer ? Comment vais-je redonder, rendre *scalable* ma base de données ? Puis-je ouvrir tel flux de *backend* depuis cette zone pour consommer ce service ?

Avant d'arriver dans ce futur, d'autres façons de consommer l'infrastructure apparaissent, et œuvrent toujours dans le même sens : la simplicité à déployer du code en production de façon fiable et rapide. Les agents conversationnels (plus communément appelés chatbots) permettent à des non-initiés de demander la mise à disposition d'un environnement, d'un pipeline de façon simple :

[96] https://aws.amazon.com/fr/guardduty/ [97] https://azure.microsoft.com/fr-fr/services/security-center/

- "Déploie un environnement PHP / MySQL"
- "Déploie-moi un cluster Kubernetes"
- "Créé-moi un pipeline dev / préprod / prod"

Le futur métier de l'Ops se trouve à la jonction du développement logiciel, de la gestion des systèmes, et de la data-science.

Nous l'avons vu, le mouvement DevOps a **remis le génie logiciel au cœur du monde de l'exploitation informatique**. Depuis quelques années, les méthodes de développement et la culture de la qualité sont au cœur de la transformation des métiers d'ingénierie des systèmes.

Cette mutation des emplois est à l'origine de nouveaux rôles. À titre d'exemple, la matérialisation des concepts DevOps se traduit chez Google par la mise en place de *Site Reliability Engineer* (SRE). Ben Treynor (*Vice President Google Engineering, founder of Google SRE*) définit le SRE comme étant "ce qui arrive quand on demande à un ingénieur du logiciel de concevoir une fonction d'exploitation"[98].

Ces nouveaux ingénieurs ne se contentent plus de développer de simples outils pour automatiser leurs propres tâches ; ils sont les concepteurs de solutions complètes construites autour d'un besoin de plus en plus centré sur le business. **L'agilité et l'élasticité** qu'apporte une plateforme telle que Kubernetes est un exemple de **développement centré sur le besoin des utilisateurs** et est un vrai succès logiciel… tout en restant un produit d'infrastructure.

À propos du cycle de développement d'un logiciel 2.0

Contrairement au cycle de développement classique, la réponse au besoin métier (l'algorithme) n'est plus implémentée directement dans le code, mais sous forme de modèle mathématique. Ce modèle mathématique est alors à son tour transformé en programme informatique (dans un langage "classique" tel que le C++ ou le Python par exemple). Enfin, pour terminer la conception du programme, il faut alimenter le code avec des grands volumes de données. C'est seulement une fois cette phase d'apprentissage terminée que le logiciel est prêt à être utilisé.

[98] https://landing.google.com/sre/

SysAdmin
Valence OPS
Expertise système

DevOps
SRE
Valence OPS
Expertise système
+
Pensée programmatique
Infrastructure-as-Code
Algorithmie

Aujourd'hui, le monde du logiciel est en pleine mutation. Andrej Karpathy, directeur du service d'intelligence artificielle chez Tesla, a même récemment introduit la notion de "logiciel 2.0"[99].

En tant qu'outil pour le *Machine Learning*, **le logiciel 2.0 est une nouvelle façon de concevoir les applications**. Certes, les méthodes de développement classiques restent utilisées comme base de conception ; cependant le moteur d'exécution change au profit de fonctions mathématiques dont les paramètres sont, pour la majeure partie, déterminés grâce à un apprentissage (la plupart du temps supervisé).

Le "Codeur 2.0" est un savant mélange de *data-scientist* et de développeur logiciel.

Face aux nouveaux challenges que doivent relever les "Ops" pour soutenir le TTM et réduire le TTR, il y a fort à parier que les SRE 2.0 seront des citizen *data scientists*[100]. L'Ops de demain incarnera probablement la réponse à cette question :

"Que se passe-t-il quand nous demandons à des codeurs 2.0 de concevoir ces mêmes fonctions d'exploitation ?"

SysAdmin
Valence OPS
Expertise système

DevOps
SRE
Valence OPS
Expertise système
+
Pensée programmatique
Infrastructure-as-Code
Algorithmie

DevOps 2.0
SRE
Valence OPS
Expertise système
+
Pensée programmatique
Infrastructure-as-Code
Algorithmie
+
Data Science
Machine Learning

[99] https://medium.com/@karpathy/software-2-0-a64152b37c35
[100] https://www.gartner.com/newsroom/id/3570917

La conception responsable

L'état de la planète

Le changement climatique est global car son étendue géographique est planétaire. Ses caractéristiques et conséquences affectent l'ensemble du vivant et de nos sociétés.

Les climatologues estiment probable de voir l'océan Arctique libre de glace durant l'été d'ici à 2050[101]. Fin février 2018 (situation exceptionnelle me direz-vous), il faisait 12°C de plus à Nuuk au Groenland qu'à Paris[102]. Un refroidissement rapide de l'Atlantique nord nous menace-t-il ? Toutes les projections des modèles climatiques actuels détectent le ralentissement de la circulation océanique de retournement, dont fait partie le fameux Gulf Stream qui apporte la chaleur de la Floride jusqu'aux côtes européennes. Ce phénomène pourrait entraîner un **bouleversement climatique sans précédent** accélérant d'autant plus la **chute de la biodiversité**.

Est-il déjà trop tard ? Les collapsologues[103] le pensent. Ils se préparent déjà aux conséquences de la chute de notre civilisation. D'autres s'accrochent au conte transhumaniste[104] et pensent qu'il est inutile de changer quoi que ce soit à notre mode de vie, à notre économie ou à la manière dont nous exerçons notre empire sur la nature, puisque des solutions aux problèmes que nous causons aujourd'hui se présenteront nécessairement demain. L'étude Mission 2020 publiée en avril 2017[105] nous révèle que le point de non retour se situe à l'horizon 2020.

Il faut donc garder espoir et agir tous ensemble, sans attendre, pour espérer **transmettre à nos enfants et aux générations futures une planète vivante.**

---

(101) Étude *Ice-free Arctic at 1.5 °C?* de James A. Screen & Daniel Williamson publiée en Mars 2017 dans la revue *Nature Climate Change*: https://www.nature.com/articles/nclimate3248 (102) https://www.accuweather.com/fr/gl/nuuk/186497/february-weather/186497 (103) La collapsologie est "l'exercice transdisciplinaire d'étude de l'effondrement de notre civilisation industrielle et de ce qui pourrait lui succéder, en s'appuyant sur les deux modes cognitifs que sont la raison et l'intuition et sur des travaux scientifiques reconnus" (Servigne et Stevens, 2015). (source : Wikipédia) (104) "Le transhumanisme est un mouvement culturel et intellectuel international prônant l'usage des sciences et des techniques afin d'améliorer la condition humaine notamment par l'augmentation des caractéristiques physiques et mentales des êtres humains." (source : Wikipédia) (105) http://www.mission2020.global/2020%20The%20Climate%20Turning%20Point.pdf (106) Les chiffres de ce paragraphe sont issus de GreenIT.fr, 2015

L'empreinte du numérique[106]

Nous, métiers du numérique, impactons le climat à hauteur de 4 % des émissions mondiales de gaz à effet de serre (GES)[107] soit autant que l'aviation civile.

Nous affectons également la biodiversité avec 60 Millions de tonnes de DEEE[108] rejetées par an dont 70 % du tonnage fait l'objet d'un trafic occasionnant des risques sanitaires majeurs par pollution des sols, de l'air et de l'eau.

Nous épuisons également les ressources naturelles et essentiellement non renouvelables (minerais, métaux, terres rares[109], pétrole, eau...), le rapport matière première / matière finie est de l'ordre de 16000:1 dans le cas de la fabrication d'une puce électronique (soit 16kg de matière première pour fabriquer 1g de puce). Dans le cas de la fabrication d'une voiture, le rapport est de l'ordre de 54:1.

La grande majorité des médias nous rabâchent que la consommation électrique des *datacenters* est la cause principale des impacts de l'IT sur l'environnement. En réalité, selon l'étude du club GreenIT[110] (portant sur 530 000 utilisateurs et 1,7 million d'équipements), **la cause principale d'impacts sont les terminaux sur l'ensemble des critères analysés** (énergie, GES, ressources et déchets).

Internet

3 Mds

6 Mds

200:1

9 milliards ← 800 millions → 44 millions

	Terminaux	Réseau	Data Center
Énergie	40 %	32 %	28 %
GES	47 %	28 %	25 %
Ressources	60 %	20 %	20 %
Déchets	50 %	25 %	25 %

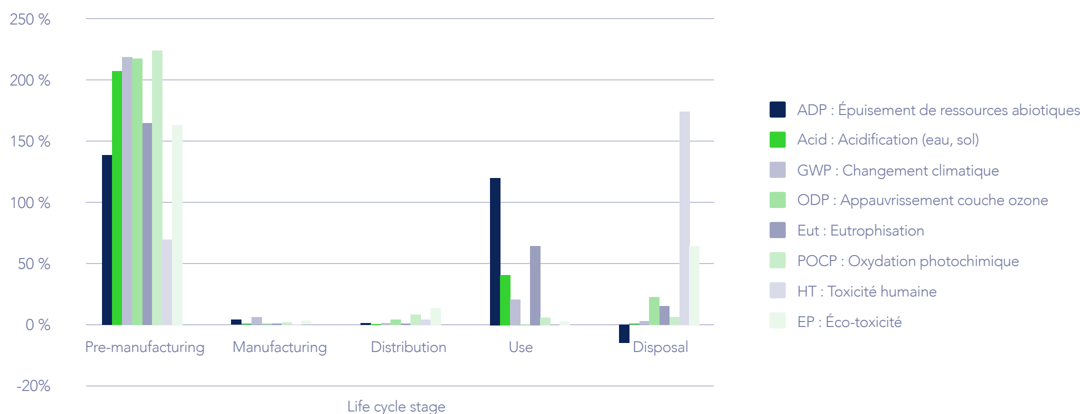
Source : GreenIT.fr, 2015

[107] "Les gaz à effet de serre (GES) sont des composants gazeux qui absorbent le rayonnement infrarouge émis par la surface terrestre et contribuent à l'effet de serre. L'augmentation de leur concentration dans l'atmosphère terrestre est l'un des facteurs à l'origine du réchauffement climatique." (source : Wikipédia) [108] "Les déchets d'équipements électriques et électroniques (DEEE, D3E ou PEEFV — produits électriques et électroniques en fin de vie —, en anglais *Waste Electronic and Electrical Equipment WEEE*) "(source : Wikipédia) [109] Liste et utilisation des terres rares : https://fr.Wikipedia.org/wiki/Terre_rare#Liste,_%C3%A9tymologie_et_utilisations_des_terres_rares (source : Wikipédia) [110] https://club.greenit.fr/benchmark2017.html

D'ailleurs, selon l'analyse de cycle de vie (ci-dessous)[111] de l'unité centrale d'un ordinateur, la phase d'extraction des minerais, transformation en composants électroniques et phase de recyclage sont de très loin les étapes les plus polluantes.

Pour diminuer l'empreinte du numérique, le principal objectif est donc de **lutter contre l'obsolescence programmée** et d'utiliser plus longtemps, moins d'équipements. Dans la plupart des cas, nous ne changeons pas notre matériel parce qu'il ne fonctionne plus, mais parce qu'il rame. L'obsolescence du matériel est accélérée par la couche logicielle, on parle d'**obésiciel**. Le nerf de la guerre est de **réduire ce "gras numérique"** via une conception responsable.

Analyse du cycle de vie d'une unité centrale



Source : Unité centrale de PC coréen, taux de recyclage de 46 %, Choi et al, 2006

111 "L'analyse du cycle de vie (ACV) est une méthode d'évaluation normalisée (ISO 14040 et ISO 14044) permettant de réaliser un bilan environnemental multicritère et multi-étape d'un système (produit, service, entreprise ou procédé) sur l'ensemble de son cycle de vie." (source : Wikipedia)

Vers une transition écologique

"La transition écologique est l'affaire de tous. Nous sommes tous concernés. Les 3 piliers pour y arriver : changement personnel, changement collectif et technologies" (Fabrice Boissier, directeur général délégué de l'ADEME).

La transition écologique est un horizon incontournable et, aujourd'hui, bien des métiers, contrairement à l'IT, ont déjà une certaine maturité (bâtiment, emballage…). La transition numérique, pour sa part, est l'une des grandes forces transformatrices de notre société. Nous allons nous appuyer sur tout ce potentiel de transformation et y intégrer les 3 piliers pour impacter en profondeur notre métier.

La réglementation est en train de se construire. Un livre blanc *Numérique et Environnement*[112] a été remis au gouvernement en mars 2018 et propose 26 mesures[113] pour **faire converger numérique et écologie**. Entre autres, celui-ci propose de rendre obligatoire **l'écoconception des services numériques**. De plus, l'intégration de l'**accessibilité** sera prochainement obligatoire via l'art. 106 de la loi pour une république numérique.

Pour cela deux démarches s'offrent à nous, le ***green IT*** et la **conception responsable**.

Le ***green IT*** est une démarche de réduction de l'empreinte environnemental d'un SI en favorisant le choix de matériels éco-conçus[114], l'utilisation d'énergie renouvelable, les économies d'énergie et de papier, le réemploi par économie circulaire sans oublier le recyclage.

La **conception responsable** est une démarche qui consiste à **penser et concevoir** des services numériques en intégrant nativement toutes les facettes du développement durable, tant en termes de réduction d'impact que de création de valeur. Au-delà de l'accessibilité et de l'écoconception, il s'agit de prendre en compte d'autres facettes telles que **la diversité, l'éthique, le respect de la vie privée** (GDPR / RGPD) et la **fracture numérique**. Plus de 70 organisations (dont OCTO Technology) se sont regroupées au sein du *Collectif Conception Numérique Responsable*[115] pour affirmer l'intérêt de cette démarche et construire ensemble des outils concrets qui aident le plus grand nombre à passer à l'action (ecoindex[116], ecometer[117], le référentiel des 115 bonnes pratiques[118]).

Et concrètement dans notre métier d'OPS, nous allons :

- **Évangéliser** jusqu'à une prise de conscience de nos clients que "chaque octet a un impact dans le monde réel" (Frédéric Bordage, greenIT.fr)

[112] https://www.greenit.fr/wp-content/uploads/2018/03/2018-03-Livre_blanc-numerique_et_environnement.pdf [113] https://www.greenit.fr/2018/03/19/26-actions-concretes-faire-converger-numerique-ecologie/ [114] reconnu par exemple par un Écolabel Nordique, un label Ange bleu ou EPEAT. [115] https://collectif.greenit.fr/ [116] http://www.ecoindex.fr/ [117] http://ecometer.org/ [118] Le livre : https://www.greenit.fr/2015/09/09/eco-conception-web-les-115-bonnes-pratiques/ ou disponible gratuitement sur le site de l'Opquast : https://checklists.opquast.com/ecoconception-web/

- Intégrer, en amélioration continue, plus de frugalité (souvent du bon sens pour réduire le "gras numérique"), plus d'efficience (notion à différencier de la performance) et de l'**éco-innovation** (l'IA et le *monitoring* vont être des secteurs clés et les Géants du Web l'ont déjà compris[119])

- Ajouter à nos UDD **des outils d'analyse d'efficience logicielle**, d'analyse d'impact environnementale[120] et d'analyse de respect des référentiels (éco-conception[121] et accessibilité[122])

- Favoriser la **mutualisation de ressources d'infrastructure** via de l'hébergement *cloud*[123] de l'orchestration de conteneurs[124] et du serverless

- Pousser **les bonnes pratiques de TDD** d'infrastructure car la qualité ne se discute pas et la non-qualité entraîne de la dette donc du "gras numérique"

- Proposer des **audits d'infrastructure** et définir des *patterns* pour optimiser l'empreinte environnementale, par exemple :

  > En identifiant des serveurs peu ou pas utilisés

  > En mettant en place des mécanismes d'arrêt/démarrage automatiques en fonction d'horaires ou simplement via chatbot ou assistant vocal pour démarrer les instances d'un environnement

  > En mettant en place des mécanismes de nettoyage d'infrastructure via par exemple *serverless*

- Creuser en R&D **l'utilisation massive de RAM non volatile** (NVRAM)[125] et l'utilisation de *quantum computing*[126] qui promet, à terme, une puissance de calcul phénoménale.

Les facteurs de ROI sont :

- Économique (moins de gras, moins de serveurs, moins de dépense)

- L'amélioration de l'image de marque sur les aspects sociétaux et écologiques

- L'attraction de jeunes talents ; les jeunes générations sont sensibles à rejoindre un employeur responsable

- L'anticipation sur la réglementation qui est en cours de construction.

[119] Par exemple : Google utilise *Deepmind* pour optimiser l'efficience de ses *datacenters* [120] https://gitlab.com/ecoconceptionweb/ecometer [121] Le livre : https://www.greenit.fr/2015/09/09/eco-conception-web-les-115-bonnes-pratiques/ ou la check-list gratuite sur le site de l'Opquast : https://checklists.opquast.com/ecoconception-web/ [122] RGAA et W3C-WCAG [123] Classement des hébergeurs *cloud* selon le rapport Greepeace 2017 : http://www.clickclean.org/downloads/ClickClean2016%20HiRes.pdf [124] https://kubernetes.io/ [125] https://blog.octo.com/la-vision-des-octos-pour-les-5-a-10-prochaines-annees/ [126] "Un calculateur quantique (anglais *quantum computer* parfois traduit ordinateur quantique, ou système informatique quantique), utilise les propriétés quantiques de la matière, telle que la superposition et l'intrication afin d'effectuer des opérations sur des données." (source : Wikipédia)

ÉCO-
RESPONSABLE

Conclusion

Les technologies que nous avons explicitées dans ce second volume ne sont pas exhaustives. Il en existe beaucoup d'autres et probablement encore plus dans les années à venir. Si nous avons choisi de mettre en lumière celles qui nous semblent incontournables, c'est parce qu'elles sont représentatives d'un **nouveau paradigme** dans les démarches DevOps, mais cela ne veut pas dire pour autant qu'elles sont nécessaires.

En effet, vouloir absolument mettre en œuvre tout le catalogue de ces *frameworks* dans son SI reviendrait à faire du "*Cargo Cult*", c'est à dire à **procéder par mimétisme** en s'imaginant être mature sur le sujet DevOps uniquement parce qu'on a choisi tous les outils à la mode. Les bénéfices apportés par chaque outil ne se concrétiseront que si **les savoir-faire**, et **les pratiques de qualité** (que nous développerons dans le troisième volume) sont également au rendez-vous.

Par abus de langage, ces outils sont souvent présentés comme des "solutions", mais nous nous méfions de ce discours "solutionniste" : en aucune manière ces *frameworks ne* sont des solutions au sens de "baguettes magiques". La réalité est qu'ils apportent de nouveaux problèmes, mais nous pensons que les problèmes qu'ils posent sont plus intéressants que les problèmes que nous aurions sans eux. Puisque "nouveaux outils" veut dire "nouveaux problèmes", l'enjeu est donc de **déplacer les problèmes** plutôt que d'imaginer les éradiquer une bonne fois pour toute. Nous trouvons notre plaisir dans ces nouvelles problématiques, plus nobles, plus excitantes. Les technologies qui y répondent nous attirent, et **l'élégance de leur conception** nous décroche un sourire en coin, appréciateur. Attirer les talents qui vibrent au cœur des technologies innovantes est également un enjeu, qui ne saurait trouver de réponse si nous occultons l'aspect "purement technique" de la technique.

"Notre goût pour la technologie ne doit nous rendre ni naïfs, ni aveugles."

L'histoire nous l'a montré, les innovations technologiques comportent systématiquement deux facettes, **elles sont ambivalentes**. À la fois source de progrès, elles ont toujours une part d'ombre qui les rend également dangereuses lorsqu'elles sont mal employées. Notre goût évident pour la technologie ne doit nous rendre ni naïfs, ni aveugles. Alors, remède et poison ? Sans une **réelle discipline** et une **recherche perpétuelle de qualité** dans leur mise en œuvre, ces innovations sont vouées à faire de gros dégâts dans nos SI.

Enfin, il est généralement admis que la production et les opérations sont, par nature, des lieux où le conservatisme est de mise. Or, il faut définitivement remettre en cause ce mythe qui n'a plus lieu d'être à l'heure du numérique : les produits logiciels, leur architecture et leur infrastructure sont en évolution permanente. Que ce soit pour des raisons de sécurité, pour des raisons légales ou tout simplement pour des évolutions fonctionnelles : tout évolue tout le temps. Et c'est bien là l'enjeu que portent les démarches DevOps ; **accompagner l'automatisation de ces évolutions**, sans quoi n'importe quel système moderne est menacé d'obsolescence à très court terme.

Le mot de la fin :

⊙ À propos de l'auteur :

Ce second volume de la trilogie à été écrit par Arnaud Mazin et grâce aux contributions inestimables de Daniel Baudry, Nicolas Bordier, Joy Boswell, Nelly Grellier, Salim Boulkour, Sebastián Cáceres, Eric Fredj, Christian Fauré, Aurélien Gabet, Mathieu Garstecki, Mathieu Herbert, Arnaud Jacob-Mathon, Yohan Lascombe, Victor Mignot, Thomas Pepio, Rémi Rey, Alexandre Raoul, Borémi Toch et Olivier Wulveryck.

L'aventure continue avec ce second volume, à nouveau fruit du travail de l'intégralité de la Tribu OPS d'OCTO Technology : AMZ, ANDR, ARP, ASC, AUB, AUG, BBR, BGA, CHL, CQU, CYPA, DABA, DUE, EDE, EFR, EPE, ETC, FXV, JGU, KSZ, LBO, LCH, LDU, MAMI, MAT, MBO, MCY, MDI, NBO, PYN, RAL, RRE, SBO, SCA, TPA, TWE, VMI, YATI et YOL.

La direction artistique et les illustrations sont l'œuvre de Sophie Delronge.

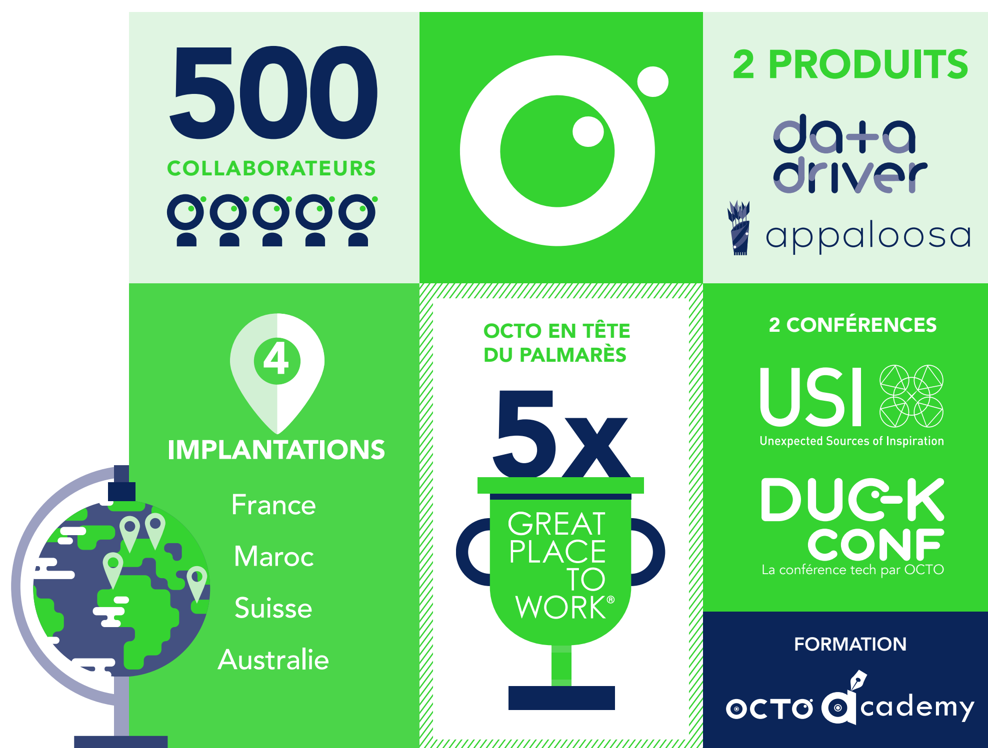
"DevOps is a human problem"
Patrick Debois

OCTO Technology

CABINET DE CONSEIL ET DE RÉALISATION IT

"Nous croyons que l'informatique transforme nos sociétés. Nous savons que les réalisations marquantes sont le fruit du partage des savoirs et du plaisir à travailler ensemble. Nous recherchons en permanence de meilleures façons de faire. THERE IS A BETTER WAY !"

– Manifeste OCTO Technology



Dépot légal : Octobre 2018

Conçu, réalisé et édité par OCTO Technology.
Imprimé par IMPRO
98 Rue Alexis Pesnon, 93100 Montreuil

© OCTO Technology 2018

Les informations contenues dans ce document présentent le point de vue actuel d'OCTO Technology sur les sujets évoqués, à la date de publication. Tout extrait ou diffusion partielle est interdit sans l'autorisation préalable d'OCTO Technology.

Les noms de produits ou de sociétés cités dans ce document peuvent être les marques déposées par leurs propriétaires respectifs.

www.octo.com — blog.octo.com

FRANCE I MAROC I SUISSE I AUSTRALIE