CULTURE DEVOPS

VOL. 03

THERE IS A BETTER WAY

PYRAMIDE DE DEVOPS :
À LA DÉCOUVERTE DES PRATIQUES DE QUALITÉ POUR UNE INFRASTRUCTURE À L'ÉPREUVE DU TEMPS.

octo.com | blog.octo.com

Pyramide de Devops : à la découverte des pratiques de qualité pour une infrastructure à l'épreuve du temps.

Jaloux des développeurs, nous souhaitons nous aussi fiabiliser notre code et aspirer à l'excellence. Nous devons copier sans honte leurs bonnes pratiques et y appliquer les spécificités de la production. Hautement disponible, scalable, observable et sécurisée : l'infrastructure sera de qualité ou ne sera pas !

Les trois histoires de DevOps : Organisation, Technologie et Qualité

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.

① Notre vision est celle d'OCTO Technology et de sa Tribu OPS qui œuvre depuis plus de sept 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 : https://www.octo.com/fr/publications/23-cloud-ready-applications

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/

Voici donc le troisième volet de la trilogie. Il présente notre vision de la qualité, adaptée aux contraintes de la production. Nous avons choisi de le découper en quatre parties représentant les différents acteurs dans l'émergence des bonnes pratiques. Dans une première partie, nous mettrons en relief ce que cela signifie de "tester" une infrastructure et les limites que nous y voyons. Nous présenterons ensuite comment instaurer au sein d'une équipe une atmosphère propice à l'amélioration continue qui devrait rappeler l'ouvrage Culture Code à nos fidèles lecteurs. Enfin, nous contextualiserons les pratiques DevOps au sein de l'entreprise et les moyens de faire émerger une culture de la qualité.

Asseyez-vous confortablement et préparez-vous pour un voyage au pays des pyramides !

Sommaire

01 **Introduction** 07-08

02 **L'Individu** 09-46
Automatisation 13
Infra as Code 15
TDD : de nouveaux outils de tests d'infrastructure 17
Idempotence comme indicateur de qualité 29
Limites et stratégie 35
Les outils de nos rêves 42

03 **L'Équipe** 47-66
Collective Code Ownership 50
Clean Code 54
La nécessité d'une approche continue de la qualité 57
Pilotage de dette : Principes d'archi 65

04 **L'Entreprise** 67-90
Continuous Delivery 70
Prod-awareness : ça veut dire quoi d'avoir bien fait son travail ? 76
Le rôle du management : accompagner plutôt qu'imposer 87

05 **Annexes** 91-96
Exemple : plateforme de conteneurs 92

06 **Conclusion** 97-100

Introduction

Les changements de ces dernières années n'ont pas épargné le métier de *sysadmin*[1]. Pourtant, la nouveauté n'a pas été accueillie à bras ouverts. Rien à faire. "La prod[2]" est devenue quelque chose de changeant, là où régnait la stabilité de l'inamovible. Tout va bien, tant que l'on n'y touche pas.

Pour mesurer la qualité d'une production, le premier indicateur qui vient en tête est *l'uptime*[3]. Pour garantir que cet indicateur soit élevé, la règle d'or consiste à ne pas toucher à quelque chose qui fonctionne. Ou en tout cas, à y toucher le moins souvent. Et quand on y touche, on s'attend à ce que cela se passe mal.

C'est souvent de cette manière que les infrastructures ont été gérées ces dernières années. Puis est arrivé le DevOps, avec cette drôle d'idée selon laquelle il faut "livrer souvent" ; que notre infrastructure ne doit plus être seulement solide, mais également élastique ; que l'on ne doit plus être chêne mais roseau.

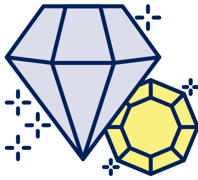
Heureusement pour les acteurs de la production, les développeurs ont aussi eu à subir ces changements quelques années plus tôt avec l'arrivée de l'agile. Ne pas réinventer la roue étant l'un des principes de base de l'informatique, l'infrastructure s'y est essayée et a adopté les "bonnes" pratiques des "bons" développeurs. Quel gain de temps ! Des décennies de *try and fail* applicables à ce domaine !

"Dans une organisation produit, la qualité doit être l'affaire de tous et de toutes."

Grâce aux épaules de ces pionniers de l'agilité, le travail d'évangélisation est plus facile. L'incompréhension autour du DevOps, souvent réduit à l'automatisation, rend néanmoins ce mouvement incomplet. On peut (on doit !) faire de la qualité avec l'automatisation. Mais si nos contraintes organisationnelles demeurent, comment livrer plus souvent ? Comment prendre en main la sécurité de son application ? Si nous ne maîtrisons pas les patterns liés à l'architecture, si nous souhaitons livrer plus souvent et avec le moins d'interventions humaines possibles, comment s'assurer que notre service sera toujours hautement disponible ?

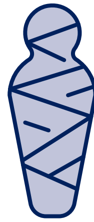
① *Sysadmin* : contraction de "system" et de "administrator". En Français "Adminsys", pour "administrateur système". Personne en charge de gérer l'infrastructure informatique au sein d'une organisation. ② La production : environnement sur lequel évoluent les application à destination des utilisateurs finaux. ③ *Uptime*, ou durée de fonctionnement, en français désigne le temps depuis lequel un système tourne sans interruption.

Le coût de la non-qualité



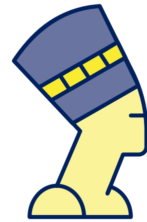
UN COÛT FINANCIER

- Des évolutions de plus en plus chères à produire.
- Des clients insatisfaits qui ne payent plus.



UN COÛT HUMAIN

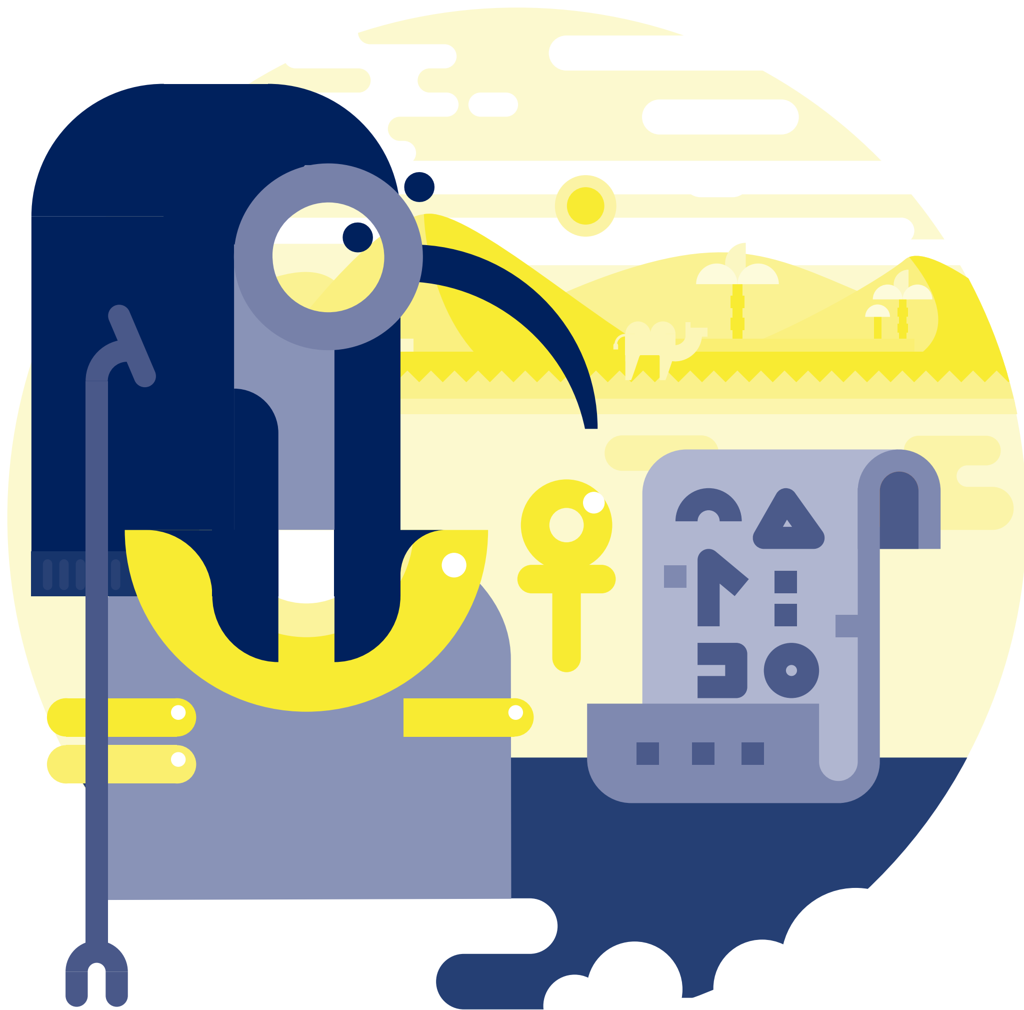
- Des développeurs démotivés et désengagés.
- Les meilleurs démissionnent.



UN COÛT STRATÉGIQUE

- Des difficultés à répondre au "*time to market*".
- Une perte de confiance dans la marque.
- Des risques d'échec ou de retard élevé.

L'Individu



Thot *: Doté d'une grande intelligence, il était considéré comme le dieu de la science, de l'écriture, de l'art et de la sagesse. Il est l'inventeur de l'écriture, du langage et des chiffres.*

Depuis quelques années, les développeurs ne travaillent plus qu'en agile. Cela se passe bien, et nous avons bien moins de bugs en production. Une fois par mois, nous devons livrer leur code sur les serveurs, pour intégrer les évolutions.

En tant qu'administrateur système, j'ai été attaché à l'une de ces équipes, pour les aider à développer des applications qui se sentiront bien sur nos serveurs. Je suis également en charge des déploiements et des scripts d'automatisation que je développe avec *Ansible*. Petit à petit, je leur laisse la main sur ces scripts et je pense que, dans quelques mois, ils pourront se passer de moi.

Je sais qu'ils aimeraient que nous fassions des livraisons plus souvent, mais c'est compliqué sur notre infra. Quand nous déployons, nous devons rester tard le soir, afin de nous assurer que le moins d'utilisateurs possible seront impactés en cas de pépin. Je dis "en cas de pépin", mais pour être honnête, nous en avons la plupart du temps. Souvent, nos scripts de déploiements ne sont plus à jour, ou créent des effets de bord que nous n'avons pas su déceler plus tôt. J'ai quand même l'impression que cela s'améliore avec le temps...

Pendant leur transformation agile, nous n'avons pas chômé non plus de notre côté. Passage de notre infra "*bare metal*" au tout virtuel. *Exit* les progiciels, bonjour l'*open-source*. Nous avons même pris le chemin de l'automatisation grâce à des scripts qui orchestrent désormais nos déploiements ! Même si nos environnements ne sont pas totalement identiques, on s'en approche.

Malheureusement, ces travaux ont un coût : nous passons beaucoup de temps à faire évoluer notre infrastructure. À cela s'ajoutent les soucis liés aux applications que l'on gère. La *morning checklist* occupe souvent le reste de la journée : disques trop pleins, scripts qui ne marchent plus, conséquences imprévues de nos travaux…

"Tout au bout de la chaîne de valeur, l'agile a marqué un tournant dans notre façon de travailler, pour nous, les "OPS"."

Revers de la médaille : le travail à fournir pour faire évoluer notre infrastructure se complexifie avec le temps. Notre base de code d'infrastructure est de plus en plus grosse et il est difficile de tout maîtriser ! Surtout avec les problèmes habituels, comme les documentations obsolètes ou notre PIC (Plateforme d'Intégration Continue) "constamment en carafe".

Heureusement, Jean est toujours capable de nous dépanner. C'est lui qui est à l'origine de toutes nos automatisations : d'abord en installant un Jenkins, afin de piloter nos scripts de monitoring, puis en automatisant la création de notre infrastructure (via d'autres scripts, en bash). Alors quand on a un souci, il est fort probable qu'il ait la solution ! Par contre, quand il n'est pas là...

Il faut avouer que je me sens un peu "à la ramasse" parmi les développeurs. J'ai l'impression que de leur côté, quand ces problèmes se posent, ils ont rapidement la solution. Ils parlent de "*software craftsmanship*[4]" et quand j'assiste à leurs points, leur base de code semble vraiment *clean* et maîtrisée ! J'ai longtemps pensé que ces principes, aussi séduisants soient-ils, étaient pour beaucoup des "*buzz-words*" ou tout simplement pas applicables au contexte de l'infrastructure.

Après tout, bien souvent, nos tests consistent à déployer une topologie complète d'infrastructure. Chez nous, tout n'est pas automatisable. Comment faire du test unitaire là-dessus ? De même, *mocker*[5] une API de *cloud* privé ? Cela n'aurait tout simplement aucun sens…

④ *Le "software craftsmanship"*, en français artisanat logiciel, est une approche du développement logiciel qui met en avant les pratiques de développement pour réaliser des logiciels de qualité, arguant qu'il ne suffit pas qu'un logiciel fonctionne, il faut également qu'il soit bien conçu. Voir notre ouvrage *Culture Code* : https://www.octo.com/fr/publications/20-culture-code.

⑤ En programmation, on utilise des *mocks* afin de reproduire de manière simulée le comportement d'une dépendance.

Automatisation

La première étape vers la qualité pour votre code d'infrastructure consiste à automatiser. Le succès de l'informatique vient de ces deux constats : les ordinateurs sont rapides et ne font des erreurs que très rarement. Un humain, pour une tâche algorithmique, a de grandes chances d'être plus lent qu'un ordinateur et d'avoir un taux d'erreur supérieur. En "scriptant" des tâches récurrentes, l'opérateur d'infrastructure se prémunit contre les erreurs liées à des actions manuelles et, au prix d'un investissement initial, se dégage du temps pour d'autres tâches.

Du fond de la mémoire de nos lecteurs aux cheveux grisonnants ressurgissent les angoisses des scripts d'exploitation en bash ou en Perl. Ces scripts prennent en compte tous les cas spécifiques à l'infrastructure qu'ils doivent gérer, dans un enchevêtrement d'appels et de contre-appels, se perdant dans une complexité cyclomatique[6] tendant vers l'infini. Mais ne soyons pas ingrats. Ces scripts, bien souvent, fonctionnent et répondent de façon pragmatique aux problématiques de l'équipe qui les a développés. Même s'ils sont le cauchemar de tout "archéologue de l'infra", ils rendent bien souvent service à ceux qui savent les utiliser.

Mais voilà, ceux qui les ont développés ont bien souvent quitté l'entreprise depuis longtemps et comprendre ce que font ces scripts devient aussi compliqué que de trouver un développeur Perl...

⑥ Le nombre cyclomatique, la complexité cyclomatique ou la mesure de McCabe est un outil de métrologie logicielle développé par Thomas McCabe en 1976 pour mesurer la complexité d'un programme informatique. Cette mesure reflète le nombre de décisions d'un algorithme en comptabilisant le nombre de "chemins" linéairement indépendants au travers d'un programme représenté sous la forme d'un graphe - source : Wikipedia.

Les geeks et les tâches répétitives

TEMPS

Perd

NON-GEEK

Lance les scripts

GEEK

Automatise
à l'aide de scripts

Gagne

Trouve ça inefficace

Fait manuellement

Se moque en voyant
la méthode compliquée
du geek

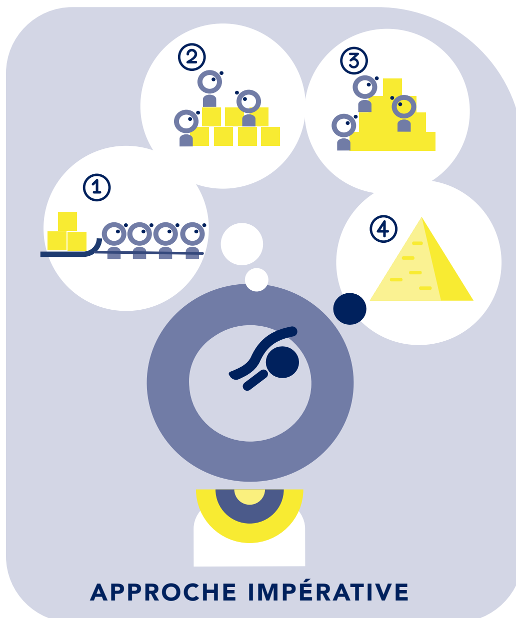
Fait manuellement

COMPLÉXITÉ DE LA TÂCHE

Infra as Code

Un pas vers un code d'infra plus compréhensible peut venir de "l'*Infrastructure as Code*". Pour l'expliquer simplement, il s'agit de passer d'une **approche impérative** (ou algorithmique) à une **approche déclarative**[7]. Cela permet de comprendre facilement l'état dans lequel on souhaite arriver et ainsi ne plus se soucier des détails d'implémentation mais plutôt du résultat final, le résultat voulu. Le code est intelligible et peut s'appréhender par briques logiques. Grâce à ces abstractions[8], le code d'infrastructure gagne ses lettres de noblesse.

Ce n'est plus des "*scripts*", faits à la va-vite sur le coin d'une table, mais du code choyé et imbibé de *software craftsmanship*. Les modifications à apporter à ce code sont plus facilement atomiques et nos procédures de qualité de code s'en accommodent donc plus facilement : *clean code, refactoring*, revue de code… On peut ainsi appliquer nos méthodes d'intégration continue, de travail en équipe et de suivi des changements. Le code, on le sait, doit porter la qualité de notre travail. Et la qualité, ça se teste !



Exemple de code utilisant Terraform

Load-balancer

:80

Web server

Approche impérative

```
$ gcloud compute \
    firewall-rules create allow-http-from-lb \
    --allow tcp:80 \
    --source-tags ln_network \
    --description "allow HTTP from lB_network"
```

Approche Déclarative

```
ressource "google_compute_firewall" "http" {
  name    = "allow-http-from-lb"
  network = "${google_compute_network.lb_network.name}"

  allow {
     protocol = "tcp"
     ports    = ["80"]
  }

  source_tags = "lb_network"
}
```


⑦ *Voir la partie Approche déclarative vs. impérative de l'ouvrage Culture Devops, Vol.02*

⑧ Voir l'article *Bonne nouvelle : L'Infra As Code c'est du code !* sur notre blog OCTO Talks : https://blog.octo.com/bonne-nouvelle-linfra-as-code-cest-du-code/

TDD : de nouveaux outils de tests d'infrastructure

À chaque fois que l'on évoque un nouveau langage de programmation ou un nouveau logiciel, une petite musique résonne inéluctablement dans nos cerveaux "piquousés" aux pratiques *crafts* : comment le tester ? L'*Infrastructure as Code* ne déroge pas à la règle.

Du code testé est aujourd'hui un gage de qualité (encore faut-il que les tests aient une valeur). Posons les bases : tester pour avoir un *feedback* sur le code écrit. Lancer une interface et simuler un parcours utilisateur, est une forme de test : un *feedback* est donné sur la manière dont notre code fonctionne.

En tentant d'appliquer cette stratégie à notre code d'infrastructure, deux tendances se dégagent ; les tests unitaires et les tests d'intégration en isolation.

Tests unitaires

Un **test unitaire** (ou TU) est une **procédure** (manuelle ou automatisée) permettant de **vérifier le bon fonctionnement** d'une partie précise d'un programme appelée "unité" ou "module". Du fait de la difficulté de réduire au maximum cette "unité", les tests unitaires sont à ce jour le parent pauvre de l'outillage de tests pour l'*Infrastructure as Code* (ou IaC).

Le seul client sérieux est *rspec-puppet*[9] qui, comme son nom l'indique, offre un cadre de TU pour Puppet[10]. Il permet de *bouchonner*[11] très largement le moteur de Puppet, d'y injecter des situations, d'invoquer la logique que nous avons codée dans nos modules et d'appliquer un ensemble d'assertions pour valider leur comportement.

Cette mécanique est rendue possible par l'architecture de Puppet, du fait de son fonctionnement de type purement déclaratif. Le principe d'une exécution se décompose suivant 4 étapes :

1. Collecte de faits (ou *facts*) : la machine sur laquelle est installé l'agent Puppet effectue un travail d'inventaire local à l'aide de l'outil *facter*. Celui-ci capture des caractéristiques

---

⑨ http://rspec-puppet.com/

⑩ Puppet est un outil permettant d'automatiser la configuration de serveurs : https://puppet.com/fr

⑪ Un bouchon est un bout de code qui permet de remplacer une dépendance, dans le cadre d'un test. Si les paramètres entrants sont identiques, il enverra toujours le même résultat.

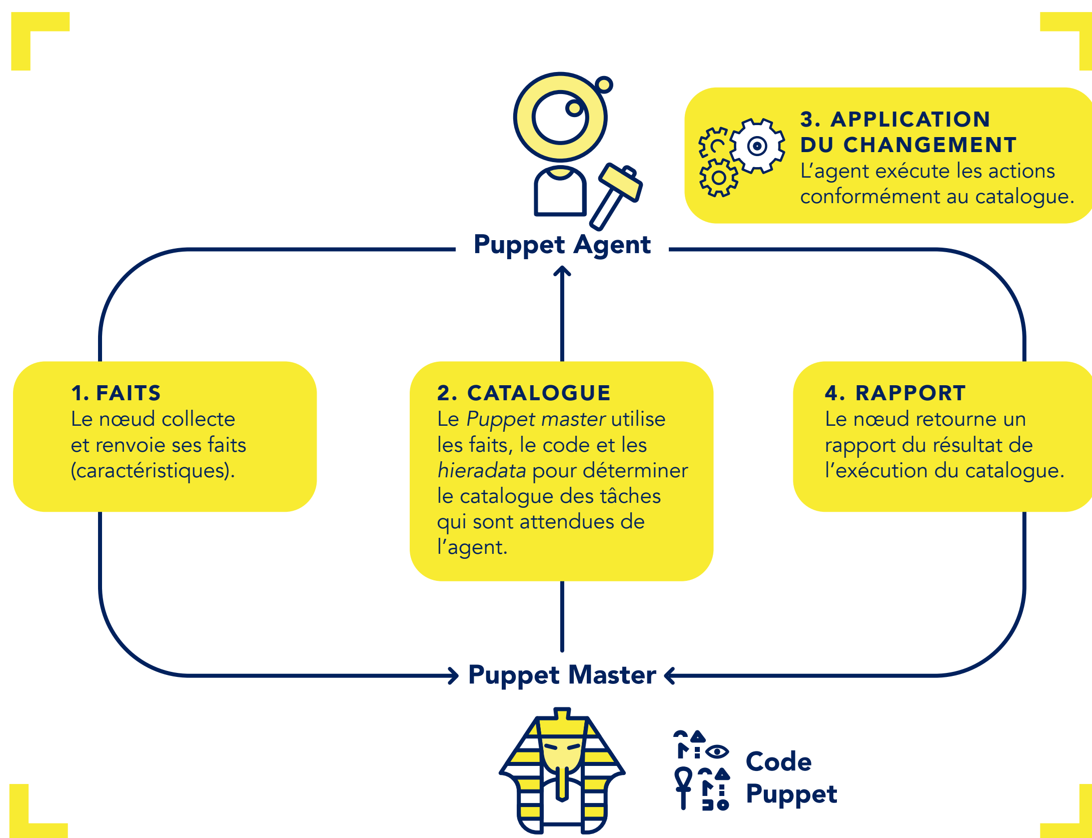
de la machine : CPU, mémoire, interfaces réseau, adresses IP, disques, volumes, types de système d'exploitation… De plus, il est possible d'ajouter ses propres *facts* en étendant le comportement par défaut de *facter*. Les faits sont ensuite envoyés au serveur Puppet : le *Puppet master*.

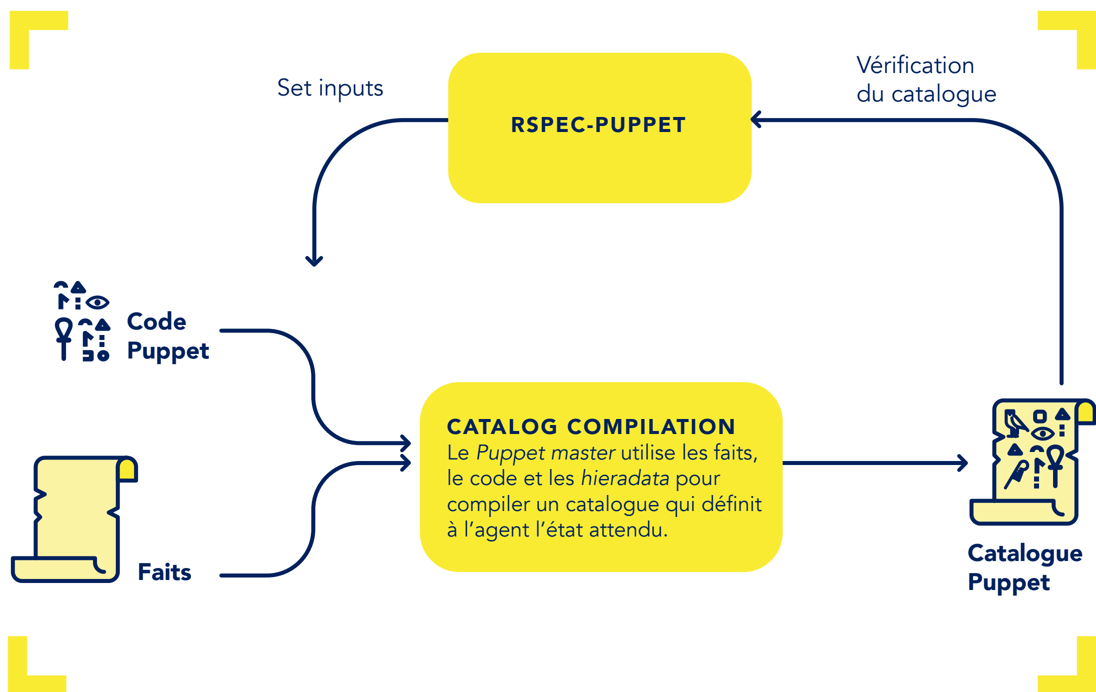
2. Élaboration et envoi du catalogue : en appliquant l'IaC (modules, classes) définie par le développeur Puppet, le *Puppet master* envoie à l'agent l'ensemble des états désirés à atteindre (type de ressources, états attendus) dans un catalogue complet.

3. Application locale du catalogue : l'agent applique le catalogue.

4. Remontée du rapport : l'agent remonte au serveur le résultat de l'exécution de l'application du catalogue.

En mode rspec-puppet, seule l'étape 2 est réalisée localement, c'est à dire sans même solliciter un *Puppet master*.





En entrée, des *facts* virtuels sont injectés à côté du code Puppet que l'on souhaite tester. En sortie, des assertions s'appliquent sur le catalogue généré.

La capacité à réaliser ces opérations en local combinée à un temps d'exécution très court (quelques secondes maximum) rendent ce processus tout à fait utilisable dans une approche de type TDD (*Test Driven Development*). Si vous n'avez jamais entendu parler de cette approche, pourquoi ne pas aller faire un tour sur notre blog[12] ?

En quelques lignes, cela consiste à coder d'abord le test exprimant le comportement que l'on cherche à atteindre, avant de coder la fonction qui implémente ce comportement.

Voici un exemple simple pour montrer concrètement ce principe. L'objectif : coder une classe Puppet qui installe deux paquets.

- **Étape 1**

Écrire le test en rspec-puppet et vérifier qu'il est bien en erreur. Puisque nous n'avons pas implémenté le code correspondant, c'est bien le comportement que nous recherchons.

---

⑫ Voir l'article *TDD contre les montagnes russes* sur notre blog OCTO Talks.

```
# Code RSpec-Puppet
describe 'base_packages' do
  context 'default params' do
    let(:params) { {} }
    it { is_expected.to compile }
    it { is_expected.to contain_package('vim') }
    it { is_expected.to contain_package('curl') }
  end
end
```

Le test propose un scénario dans lequel nous invoquons la classe Puppet `base_packages` sans paramètre. On s'attend à ce que le catalogue compile correctement (cela va sans dire, mais mérite toujours d'être dit) et à ce que les paquets `vim` et `curl` soient installés.

Faisons une première implémentation vide de la classe `base_packages` :

```
# Code Puppet
class base_packages () {
}
```

Lançons une première fois rspec-puppet pour constater le résultat des choses :

```
base_packages
  default params
    should contain compile into a catalogue without dependency
cycles
    should contain Package[vim] (FAILED - 1)
    should contain Package[curl] (FAILED - 2)
Finished in 0.62943 seconds (files took 0.42828 seconds to load)
3 examples, 2 failures
```

Patatra ! Les lignes jaunes sont sans appel : le test ne passe pas. Plus exactement, la compilation du catalogue a réussi (première ligne blanche), mais les deux assertions suivantes sont en échec. Logique, le code n'a pas encore été implémenté dans la classe Puppet.

• **Étape 2**

Implémenter le code pour faire passer les tests.

C'est à vous de jouer, vous pouvez implémenter la classe. Cet exemple, parfaitement trivial ne demande que très peu d'énergie. Mais vous pouvez imaginer mettre beaucoup plus de logique dans votre code.

```
# Code Puppet
class base_packages () {
  package { 'vim':
    ensure => present
  }
  package { 'curl' :
    ensure => present
 }
}
```


Vérifions que les tests passent alors en vert (soit blanc dans ce livre) :

```
base_packages
  default params
    should contain compile into a catalogue without dependency
cycles
    should contain Package[vim]
    should contain Package[curl]
Finished in 0.82657 seconds (files took 0.41641 seconds to load)
3 examples, 0 failures
```

Bravo, votre code est couvert d'un test, fantastique. Notez au passage que le *feedback* a mis un peu moins d'une seconde pour revenir.

Prise de décision

Conception du changement

Déroulement du test

Observation du résultat

La boucle de *feedback* représente le quadriptyque “prise de décision” / “conception du changement” / “déroulement du test” / “observation du résultat”. Coder consistant essentiellement à prendre une succession de décisions, il est impératif de travailler à avoir une boucle de *feedback* la plus rapide possible.

Travailler à rendre la boucle de *feedback* plus rapide rendra vos tests utiles, et ne fera qu’amplifier la confiance que vous aurez en votre code.

Faire de petits changements atomiques vous permettra de raccourcir chaque étape de votre boucle de *feedback*.

La phase de “déroulement des tests” est celle sur laquelle la manière de concevoir vos tests aura le plus d’impact.

• **Étape 3**

Refactoring, action de **modifier la structure** d'un code **sans en modifier le comportement**. Pas de *refactoring* sans tests "au vert" (blanc dans ce livre).

Le code fonctionne, il a un premier test. Nous pouvons à présent le refactorer pour le rendre plus clair. Nous allons simplement factoriser l'appel à la ressource *package* en lui passant un tableau en paramètre :

```
# Code Puppet
class base_packages () {
  package { ['vim', 'curl'] :
    ensure => present
  }
}
```


Nous avons ici appliqué le principe DRY (*Don't Repeat Yourself*). Vérifions à présent que les tests sont toujours verts (blancs dans ce livre) :

```
base_packages
  default params
    should contain compile into a catalogue without dependency
cycles
    should contain Package[vim]
    should contain Package[curl]
Finished in 0.76547 seconds (files took 0.54689 seconds to load)
3 examples, 0 failures
```

• **Étape 4**

Same player, shoot again. Nous souhaiterions pouvoir invoquer notre classe en lui précisant en paramètre (optionnel) une liste des paquets à installer :

```
# Code RSpec-Puppet
describe 'base_packages' do
  context 'default params' do
    let(:params) { {} }
    it { is_expected.to compile }
    it { is_expected.to contain_package('vim') }
    it { is_expected.to contain_package('curl') }
  end
  context 'override params' do
    let(:params) { { 'packages' => ['wget'], } }
    it { is_expected.to compile }
    it { is_expected.to contain_package('wget') }
    it { is_expected.to_not contain_package('vim') }
    it { is_expected.to_not contain_package('curl') }
  end
end
```


Deux tests sont désormais déclarés. Le premier garantit la non-régression du comportement par défaut : invocation de la classe sans paramètre conduisant à l'installation de `vim` et `curl`. Le second est introduit pour répondre à notre nouvelle exigence.

À nouveau, si l'on lance les tests à ce moment de l'histoire, la sanction est sans appel :

```
base_packages
  default params
    should contain compile into a catalogue without dependency
cycles
    should contain Package[vim]
    should contain Package[curl]
  override params
    should contain compile into a catalogue without dependency
cycles (FAILED - 1)
    should contain Package[wget] (FAILED - 2)
    should not contain Package[vim] (FAILED - 3)
    should not contain Package[curl] (FAILED - 4)
Finished in 0.91724 seconds (files took 0.39541 seconds to load)
7 examples, 4 failures
```

Reste alors à adapter le code Puppet pour faire passer tous les tests au vert. À nouveau, “grosse difficulté” puisque l’utilisation d’une variable par défaut dans la déclaration d’une classe Puppet fait partie des changements les plus triviaux :

```
# Code Puppet
class base_packages (
  $packages = ['vim', 'curl'],
  ) {
  package { $packages :
    ensure => present,
  }
}
```

Et voilà, les tests rspec-puppet nous confirment que la modification n’a pas déclenché un chaos interstellaire :


```
base_packages
  default params
    should contain compile into a catalogue without dependency
cycles
    should contain Package[vim]
    should contain Package[curl]
  override params
    should contain compile into a catalogue without dependency
cycles
    should contain Package[wget]
    should not contain Package[vim]
    should not contain Package[curl]
Finished in 0.93594 seconds (files took 0.46291 seconds to load)
7 examples, 0 failures
```

Même si l’exercice présenté peut paraître simpliste, la démarche n’en demeure pas moins primordiale. Refactorer permet de se poser des questions de lisibilité, de standard de code, de portabilité, de performance… Tout est permis désormais que l’on a un moyen simple de savoir si nos modifications ne vont pas engendrer un dysfonctionnement : notre test.

S'assurer que l'on implémente correctement des cas de plus en plus complexes sans casser les précédents est un des premiers bénéfices du TDD.

Tests d'intégration en isolation

Un test d'intégration est une procédure permettant de valider la bonne intégration de plusieures "unités" ou "modules". Les tests unitaires ne dispensent pas de réaliser des tests d'intégration.

Dans ce modèle de tests d'intégration en isolation, on ne parle pas de tests unitaires puisqu'on veut tester le bon fonctionnement entre plusieurs modules. Souvent il est nécessaire d'avoir une machine (ou un conteneur) où déployer une partie de notre code. Effectivement, là où un test unitaire sera "autosuffisant", une infrastructure est nécessaire à notre "code infra" pour y réaliser des tests d'intégration. En revanche, leur portée est limitée pour ne pas tirer toute la complexité d'un SI. Le principe de fonctionnement repose sur une fusée à 4 étages :

1. Approvisionnement d'un environnement minimaliste (serveurs, conteneurs, réseau, stockage…).

2. Application d'une portion d'IaC.

3. Outil de tests effectuant des assertions sur l'environnement déployé.

4. Nettoyage de l'environnement (destruction).

Plusieurs outils ont adopté ce principe de fonctionnement : Test Kitchen[13], Molecule[14], Beaker[15]. Leur force principale se trouve dans la versatilité de leur usage. Pour chacune des 4 étapes, des connecteurs permettent d'adapter le comportement souhaité :

Étapes 1 & 4 : Utilisation au choix d'AWS, Docker, VirtualBox…

Étape 2 : Application de code d'infrastructure (Ansible, Chef, Puppet, Saltstack…).

Étape 3 : Validation d'assertions (Bats, server-spec, TestInfra…).

⑬ Pour tester ses cookbook Chef : https://github.com/test-kitchen/test-kitchen. ⑭ Pour tester ses rôles Ansible : https://github.com/metacloud/molecule. ⑮ Pour tester son code Puppet : https://github.com/puppetlabs/beaker.

Ces tests n'étant par définition que sur une seule machine ("en isolation"), ils ne sont bien entendu pas idéaux pour tester l'état d'un *cluster* réparti sur plusieurs machines. Prenons l'exemple de Zookeeper, un outil de gestion de configuration pour systèmes distribués. En fonctionnement nominal, un des nœuds Zookeeper est *leader* :

```
# sur le leader
Mode: leader
```

Les autres sont *followers* :

```
# sur tous les autres nœuds
Mode: follower
```

Avoir avec certitude une idée de l'état de santé du *cluster* demande donc d'avoir une vision transverse sur toutes les machines du cluster car il n'est pas possible *a priori* de connaître l'identité du nœud *leader*.

Tests d'intégration et TDD

Dans le monde de l'infrastructure, nous utilisons principalement des outils d'abstraction (tels qu'Ansible, Puppet, Terraform…). Ces outils nous permettent souvent d'atteindre plus rapidement nos cibles, tout en nous reposant sur une communauté active dans la détection et résolution de bugs. Ces outils sont la plupart du temps, couverts par des tests unitaires (sinon : fuyez !). Dans une démarche TDD nous en viendrons à tester la manière dont nous les implémentons : **nous testons l'intention**. Par exemple, si votre outil de *configuration management* vous propose un module[16] pour créer un utilisateur, votre test portera sur la réalisation effective ou non de cette création d'utilisateur.

Ces tests que nous écrirons, et qui ressemblent à des tests unitaires, n'en sont plus vraiment : ils portent sur la manière dont nous utilisons le module, dont nous l'implémentons. Si la terminologie n'est qu'un détail, il est néanmoins important de bien comprendre l'intention avec laquelle nous écrivons un test :

- **Les tests écrits** lors de notre phase de TDD, nous permettent de documenter le résultat attendu, de nous assurer des non-régressions, de notre bonne utilisation de l'outil choisi…
- **Les tests d'intégration** à proprement parler,

---

16 "Modules" : terminologie que l'on retrouve dans Ansible (https://docs.ansible.com/ansible/2.7/modules/list_of_all_modules.html) ou encore dans TestInfra (https://testinfra.readthedocs.io/en/latest/modules.html).

portent sur un ensemble cohérent de fonctions successives.

Le développement d'infrastructure, aujourd'hui, ressemble de plus en plus à un travail d'intégrateur : nous prenons des outils existants, et nous étendons leurs capacités afin de répondre au besoin du client. Il est tout à fait normal – et rassurant – qu'une partie de nos efforts portent sur la qualité de ce travail.

L'objectif est de **réussir à découper le code de votre infrastructure** de manière à **rendre testable** des parties logiques de votre système d'information. Là encore, *no silver bullet*[17] : pas d'outil miracle qui s'impose à vous. Comme vu précédemment, à chaque outil son *framework* de tests (Test Kitchen, Molecule, Beaker...) auquel nous pourrons éventuellement en ajouter d'autres comme Cucumber[18], GauntIt[19], voire TestCafé[20] pour les plus énervés.

En intégrant des outils de tests pertinents dans notre routine de développement (qu'elle soit *test driven*, en intégration continue, ou autre), nous nous offrons de nombreux avantages :

- **Une boucle de *feedback* plus courte** : on n'attend pas de déployer sur un environnement pour connaître les actions que notre code va engendrer. Le développement se fait "par petits pas", ce qui permet de détecter plus tôt les bugs, et surtout, de diminuer la portion de code dans laquelle ils peuvent avoir été introduits.

- **Une qualité accrue** : en testant le code, non seulement vous diminuez le nombre de bugs, mais vous assurez également la production d'une documentation via votre code de test. Si l'on se demande quel résultat est censée produire telle ou telle partie, les tests, en utilisant des cas et des valeurs concrètes, nous apporterons une réponse plus fiable qu'un document Word ou PDF obsolète.

- **Un prérequis pour aller vers le déploiement continu** : le déploiement continu n'est pas le Graal d'une organisation DevOps, mais il y ressemble. Personne ne peut décemment penser déployer en continu du code qui ne serait pas suffisamment testé.

⑰ https://fr.wikipedia.org/wiki/Pas_de_balle_en_argent. ⑱ Tests d'acceptance : https://cucumber.io/. ⑲ Tests d'acceptance pour les principes de sécurité : http://gauntlt.org/. ⑳ Automatisation de navigation web : https://devexpress.github.io/testcafe/.

Idempotence comme indicateur de qualité

Dans le monde du code d'infrastructure, l'idempotence est très recherchée. Une opération est idempotente si elle donne toujours le même résultat quel que soit le nombre de fois où elle est exécutée. **C'est très important, car nous cherchons à décrire un état souhaité**. Certains frameworks de tests permettent de tester cette idempotence, comme par exemple, Molecule[21]. Nous allons regarder de quelle manière utiliser Molecule pour vérifier l'idempotence de nos rôles Ansible. Par défaut, son exécution se décompose comme ceci, en 11 phases :


```
--> Test matrix

└── default        : nom du scénario
    ├── lint           : linter de code
    ├── destroy        : supprimer les instances en cours s'il y en a
    ├── dependency     : installer les dépendances depuis les fichiers requirements
    ├── syntax         : vérifier la syntax du playbook
    ├── create         : créer l'environnement d'exécution (Docker, Vagrant…)
    ├── prepare        : jouer le playbook "prepare.yml" (vide par défaut)
    ├── converge       : lancer les tâches du rôle cible
    ├── idempotence    : tester l'idempotence
    ├── side_effect[22] : lancer des événements perturbateurs avant les tests
    ├── verify         : lancer les tests unitaires
    └── destroy        : supprimer les instances en cours
```

Sans entrer dans les détails du TDD avec lequel vous êtes désormais familier, nous souhaitons réaliser un rôle pour désactiver les algorithmes de chiffrement trop faibles[23]. Depuis le répertoire "roles" de notre projet, initions notre rôle Ansible (`molecule init role --role-name disable-weak-ssh-ciphers`). Dans notre configuration, l'environnement de tests sera composé d'une image Docker, et les tests réalisés via Testinfra[24] :

```
# roles/disable-weak-ssh-ciphers/molecule/default/molecule.yml
---
dependency:
  name: galaxy
driver:
  name: docker
lint:
  name: yamllint
platforms:
- name: disable-weak-ssh-ciphers
    image: centos_with_ssh:latest
    privileged: true
    command: /sbin/init
provisioner:
  name: ansible
  options:
    vv: "True"
  lint:
    name: ansible-lint
scenario:
  name: default
verifier:
  name: testinfra
  options:
    v: true
  lint:
    name: flake8
```


㉑ The Wizard: Ansible, Molecule and Test Driven Development : https://blog.octo.com/en/the-wizard-ansible-molecule-and-test-driven-development/.
㉒ The Wizard: Side effects : https://blog.octo.com/en/the-wizard-side-effects/. ㉓ Quelques sources : https://infosec.mozilla.org/guidelines/openssh ; https://stribika.github.io/2015/01/04/secure-secure-shell.html ; https://en.wikipedia.org/wiki/Cipher_security_summary. ㉔ Testinfra permet de faire des assertions sur l'état de vos serveurs : https://github.com/philpep/testinfra.

Éditons désormais notre test, qui ressemblera à ceci :

```
#roles/disable-weak-ssh-ciphers/molecule/default/tests/test_integra
tion.py

import os

import testinfra.utils.ansible_runner

testinfra_hosts = testinfra.utils.ansible_runner.AnsibleRunner(
    os.environ['MOLECULE_INVENTORY_FILE']).get_hosts('all')


def test_ssh_weak_ciphers(host):
    cmd = host.run("ssh -o
Ciphers=3des-cbc,aes128-cbc,aes192-cbc,aes256-cbc \
                        -o StrictHostKeyChecking=no \
                        -o UserKnownHostsFile=/dev/null \
                        localhost")
    assert cmd.rc == 255
    assert “no matching cipher found” in cmd.stderr
```


Ce code permet de demander à Testinfra d'exécuter une connexion SSH en utilisant des algorithmes de chiffrement (“*ciphers*”) considérés comme “faibles”. Si aucun de ces algorithmes de chiffrement n’est accepté par le serveur, alors nous estimons que le comportement est celui recherché.

Sans le code Ansible associé, l'exécution de notre test d’intégration va échouer lors de la phase de “`verify`” (la phase lors de laquelle seront exécutés les tests Testinfra) :

```
> molecule converge
--> Test matrix

└── default
    ├── dependency
    ├── create
    ├── prepare
    └── converge
[...]


> molecule verify
--> Test matrix

└── default
    └── verify
[...]

============================ FAILURES ========================
____test_ssh_ciphers[ansible://disable-weak-ssh-ciphers] ____

    host = <testinfra.host.Host object at 0x111c72690>

        def test_ssh_ciphers(host):
            cmd = host.run("ssh -o
Ciphers=3des-cbc,aes128-cbc,aes192-cbc,aes256-cbc \
                                -o StrictHostKeyChecking=no \
                                -o UserKnownHostsFile=/dev/null \
    >                           localhost")

    tests/test_default.py:13:

[...]
==================== 1 failed in 1.27 seconds ================
```

Le plus court chemin pour faire “passer ce test au vert” est d’ajouter les tâches suivantes :

```
# roles/disable-weak-ssh-ciphers/tasks/main.yml
---
- name: Enable strong ciphers only
  lineinfile:
    dest: /etc/ssh/sshd_config
    regexp: "^#?Ciphers"
    line: "Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh.
com,aes128-gcm@openssh.com,aes256-ctr,aes192-ctr,aes128-ctr"
    state: present

- name: Restart sshd
  service:
    name: sshd
    state: restarted
```


Ansible va éditer le fichier de configuration du service SSH, afin de n’autoriser qu’une liste restreinte d’algorithmes de chiffrement, et redémarrer ledit service, pour appliquer la nouvelle configuration.

```
$ molecule verify
--> Test matrix

└── default
    └── verify

--> Scenario: 'default'
--> Action: 'verify'

[...]
tests/test_default.py::test_ssh_ciphers[ansible://disable-weak-ssh
ciphers] PASSED [100%]

=================1 passed in 1.50 seconds =================
Verifier completed successfully.
```

Nos tests passent ! Pourtant… Nous n'avons pas fini. Comme nous pouvons le vérifier en lançant la commande "**`molecule idempotence`**", notre code n'est pas idempotent. Effectivement, notre seconde tâche, qui consiste à redémarrer le service **`sshd`**, se jouera à chaque fois, même si la configuration n'a pas été changée. Nous ne sommes plus dans une approche qui permet de définir un état, mais une approche impérative.

Ansible couvre ce cas grâce à la notion de "*handlers*[25]", qui sont des opérations qui ne s'exécutent qu'à la suite d'un changement. Voici notre code après *refactoring* :

```
# roles/disable-weak-ssh-ciphers/tasks/main.yml
---
- name: Enable strong ciphers only
  lineinfile:
    dest: /etc/ssh/sshd_config
    regexp: "^#?Ciphers"
    line: "Ciphers chacha20-poly1305@openssh.com,aes256-gcm@openssh
.com,aes128-gcm@openssh.com,aes256-ctr,aes192-ctr,aes128-ctr"
    state: present
  notify: Restart sshd


# roles/disable-weak-ssh-ciphers/handlers/main.yml
---
- name: Restart sshd
  service:
    name: sshd
    state: restarted
```

Et voilà !

Lançons désormais la cinématique complète via :

```
$ molecule test --all
```

It works!

㉕ https://docs.ansible.com/ansible/2.6/user_guide/playbooks_intro.html#handlers-running-operations-on-change

Limites et stratégie

Ces dernières années, les pratiques et les outils autour de la testabilité du code d'infrastructure ont commencé à faire leur apparition sur certains projets. Mais nous partons de très loin. Le sujet est complexe, et avance à petits pas : mais il avance. Malgré tout, peu nombreuses sont les infrastructures qui n'ont pas à rougir de leur couverture de tests. Et beaucoup mentiraient s'ils se prétendaient sereins à chaque déploiement. Nous avons réussi la plupart du temps à sécuriser les déploiements des applicatifs, mais nos socles d'infrastructure sont encore trop fragiles.

Aujourd'hui, encore de nombreux cas d'usage ne sont pas couverts par les outillages de tests, dont voici quelques exemples :

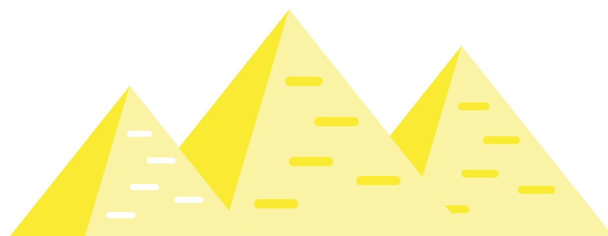
• **Les problèmes liés à un *provider cloud*.**
Des disques qui ne sont pas dans le bon ordre, une configuration d'OS exotique, une API trop souvent en erreur…

• **Les problèmes liés à l'intégration d'outils.**
Implémentation douteuse de standards, utilisation de dépendances dépréciées…

• **La lenteur de la boucle de *feedback*.**
Encore une fois : cela s'est amélioré, mais nous sommes encore loin de quelque chose d'indolore. En cause, entre autre, le téléchargement de paquets logiciels, ou encore la communication avec les API distantes. Une part non-négligeable de nos tests nécessitent de démarrer un conteneur, une machine virtuelle...

De nombreux sujets restent encore à "craquer".

Alors, en tant qu'OPS, comment choisir notre stratégie de tests ? En tant que dev, nous nous aidons de la pyramide des tests.



Attention à l'intention.

L'intention d'un test : la raison pour laquelle nous l'avons écrit.

Un test écrit lors d'un développement piloté par le test unitaire, pour assurer une boucle de *feedback* rapide et qui fournira un harnais de non-régression, n'a pas la même intention qu'un test d'intégration (cf. tableau ci-contre).

TEST UNITAIRE

INTENTION

- Fournir sa boucle de *feedback* sur un périmètre de travail restreint, pour permettre de piloter son développement par les tests.
- Mitiger les régressions :
  Changement de comportement de l'outil testé.
  Évolution dans notre code qui met le test en échec.
- Documentation du code lié.

LIMITATIONS

Peut faire doublon avec les tests mis en place par le fournisseur de l'outil d'IaC.

TEST D'INTÉGRATION

INTENTION

- Raccourcir la boucle de *feedback*.
- Documentation du résultat lié à une ou plusieurs étapes en dépendances.
- Peut tester une topologie complexe (*cluster*…).

LIMITATIONS

- Complexe à écrire et à maintenir.
- Plus long à exécuter qu'un test unitaire.

TEST END-TO-END

INTENTION

- Assurer qu'un changement n'ait pas un effet de bord quelque part dans la chaîne de bout en bout.
- Monitorer son service.

LIMITATIONS

- Très complexe à écrire et à maintenir.
- Long à exécuter.
- Vision trop bout en bout : si le test échoue, cela peut venir de très nombreux paramètres.
- L'investigation sera longue.
- Instabilité : en raison de leur complexité, certains tests *end-to-end* auront tendance à bagoter (ça marche 99 fois sur 100, mais de temps en temps, on a un *timeout*).

D'autres tests enfin, viendront en complément :

Tests de sécurité, Tests de charge, Tests de résilience, *Nightly builds*…

Pyramide

Comme décrite et portée dans de nombreuses publications OCTO (notamment *Culture Code*[26] et nos articles de blog[27]), la pyramide de tests est construite comme ci-dessous.

Comment l'adapter à notre travail sur le code d'infrastructure ? **L'approche déclarative du code d'infrastructure ne nous permet pas de retranscrire directement cette pyramide.** Les tests unitaires sont majoritairement gérés par l'outil qui fournit l'abstraction (Ansible teste son code Python, Terraform son code Go…). Ils sont donc nombreux, mais nous n'en avons pas la responsabilité. De plus, nous ne les lancerons que très rarement nous-même, voire jamais. En tant qu'utilisateur de ces outils, nous nous devons de nous assurer de leur bonne intégration.

PLUS CHER
PLUS LENT

Sécurité, charge, tests métier

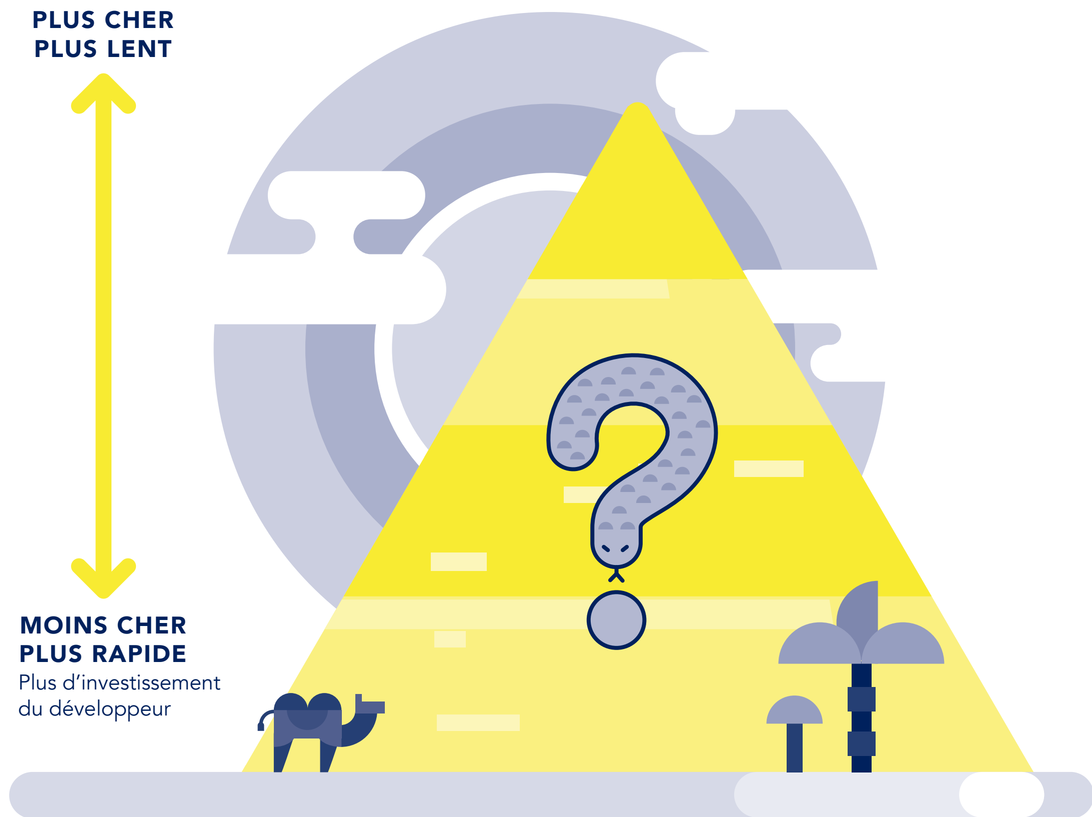
Interface graphique, de bout en bout, tests fonctionnels

Intégration, tests d'API

Tests unitaires

MOINS CHER
PLUS RAPIDE
Plus d'investissement du développeur

[26] https://www.octo.com/fr/publications/20-culture-code. [27] https://blog.octo.com/.



Mais au final, la dénomination des tests n'a que peu d'importance. Dans le cas de nos développements de code d'infrastructure, "l'espace-temps" est différent. Nos tests d'infrastructure sont plus lents que les tests applicatifs du fait que les assertions de nos tests doivent valider un état d'infrastructure plutôt qu'un état en mémoire. Mais les outils cités plus haut permettent de raccourcir la boucle de *feedback* en automatisant les assertions et en isolant une partie de l'infrastructure.

Écrire un test, c'est faire un investissement en temps. Cet investissement se rentabilise un peu plus à chaque fois que le test est lancé. Un test dont le retour est rapide, sera indolore pour le développeur et donc lancé souvent. Ainsi, il y a une corrélation très forte entre ces deux facteurs. **Plus un test est rapide (à développer et à exécuter), plus il sera exécuté souvent, plus il aura de la valeur.**

Accentuer l'effort sur les tests :

- ***qui sont rapides à exécuter,***
- ***qui ne coûtent pas cher à développer ET à maintenir,***
- ***qui expliquent/documentent pourquoi une ligne de code existe.***

Quels sont les tests qui doivent alors composer la base de notre pyramide ?

- Les *smoke-tests*, qui permettent de déceler si notre infrastructure est fonctionnelle ou s'il y a un problème critique, rendant tout test complémentaire inutile tant qu'il n'est pas résolu.
- Les *health-checks*, qui permettent de savoir si des briques d'infrastructure ou applicatives sont opérationnelles ou si elles ont besoin d'être remises en service.
- Les *readiness tests*, qui permettent de savoir si les services nécessaires au bon fonctionnement d'une brique applicative ou d'infrastructure sont opérationnels. Chacun des services est alors lui-même conscient que son environnement n'est pas opérationnel et va attendre la fin de leurs mise en place.

Pour continuer à construire votre pyramide, plus vous montez, plus la boucle de *feedback* est longue, plus la complexité à concevoir et maintenir est élevée, moins vous devez investir sur les tests.

Pour vous aider à classifier les tests et les placer dans la pyramide, posez-vous les questions suivantes[28] :

Qui exécute les tests ?
Où sont-ils exécutés ?
Comment les tests sont-ils exécutés ?
Quand sont-ils exécutés ?
Combien de temps la boucle de *feedback* prend-elle ?
Qui est le principal intéressé par les tests ?
Et pourquoi ?

Par exemple, pour un test unitaire :

Qui exécute les tests ? Le développeur.
Où sont-ils exécutés ? Sur sa machine.
Comment les tests sont-ils exécutés ? En lançant l'outil de test unitaire.
Quand sont-ils exécutés ? À chaque modification de code par le développeur.
Combien de temps la boucle de *feedback* prend-elle ? Quelques millisecondes.
Qui est le principal intéressé par les tests ? Le développeur.
Et pourquoi ? Pour vérifier que le code produit répond bien à l'intention du développeur.

Cet exercice est intéressant à refaire pour chaque typologie de tests que vous voulez mettre en place sur votre produit.

[28] Inspiré de l'article de Sébastien Douche, *Nomenclature des tests logiciels* : http://douche.name/blog/nomenclature-des-tests-logiciels/

Plus de capacités pour plus de tests

Une approche continue de la qualité implique d'avoir de la capacité supplémentaire disponible (calcul, stockage, mémoire) pour **construire et tester le plus possible**. Alors qu'avec des approches classiques, nous avons tendance à dimensionner au plus juste les capacités utilisées par les projets, l'approche continue impose de garder en permanence une marge suffisante pour gérer les aléas des projets et jouer en plus les tests d'infrastructure. L'idéal, bien sûr, est de s'appuyer sur le *cloud* : ses ressources sont virtuellement infinies !

Une fois cette contrainte libérée, le champ des possibles s'ouvre. Parmi les nouvelles familles de tests possibles, la plupart exécutables automatiquement toutes les nuits, citons :

Tests de déploiement d'infrastructure
(*neutral builds*[29], *nightly builds*[30]). Certaines stratégies de tests d'IaC imposent de disposer de plateformes pour des durées de vie très courtes (quelques minutes voire heures en moyenne).

Tests de résilience
(*design for failure*[31], *chaos engineering*[32]) Que se passe-t-il si telle ou telle pièce d'infrastructure tombe en erreur, d'un coup ? Nous devons admettre la panne comme faisant partie intégrante du cycle de vie d'une infrastructure, d'une application. En intégrant des éléments destinés à générer le chaos, et sous condition d'avoir des tests/sondes qui nous assurent que la plateforme fonctionne, nous sommes alors obligés de prévoir dans nos développements les éléments permettant une continuité de service.

Tests de performance
Deux approches complémentaires sont possibles.

Pour ceux qui peuvent accéder à des infrastructures éphémères (*cloud* privé ou public), la première approche consiste à construire une plateforme iso-production en termes de dimensionnement. Après une alimentation d'un jeu de données suffisant, un injecteur de charge (type *Gatling*[33]) passe sur la plateforme et affiche les résultats dans votre intégration continue.

L'autre approche complémentaire consiste à mesurer une **dérive de performance**. Ici, peu importe la valeur absolue du temps de réponse, mais le procédé permet de détecter si une nouvelle version est plus lente ou plus rapide que la précédente. Il ne s'agit pas d'une vision complète de la performance mais d'une première tendance, très rapidement disponible.

Tests de sécurité
Scan de vulnérabilités, logiciel de *pentesting* (tentative d'intrusion), tout est possible à partir

---

㉙ Construction de l'infrastructure pour refléter l'état courant de la base de code, réalisée dans un environnement neutre (c'est-à-dire qui ne servira qu'à cela) - source : Wikipedia. ㉚ Construction "de nuit" de notre infrastructure, de bout en bout, afin de vérifier que l'on en est capable, à n'importe quel moment, de zéro. ㉛ "Conçu pour supporter la défaillance" : une application informatique doit être capable de supporter la panne de n'importe quel composant logiciel ou matériel sous-jacent - source : https://blog.octo.com/design-for-failure/. ㉜ Simuler des pannes en environnement réel et vérifier que le système informatique continue à fonctionner - source : Wikipedia. ㉝ https://gatling.io/.

du moment où c'est automatisable[34]. Comme il est aisé de reconstruire les plateformes, lancer des tests même destructifs sur un environnement n'est plus un problème. Vous pouvez vous lancer en (presque) toute sérénité. Toutefois, cet outillage ne dispense de réaliser des campagnes de tests de pénétration (qui pour l'heure restent manuelles, non-automatisables).

Tests de montée de version

Comme il n'y a rien de plus fastidieux que de tester les montées de version, autant demander à l'intégration continue de faire le travail. Nous allons donc commencer par un re-déploiement d'une version iso-prod sur une nouvelle plateforme, enchaîner sur la récupération de données anonymisées, l'application de la procédure de montée de version en cours de mise au point, passer un non-régression générale sur la plateforme avant de la détruire si on le souhaite. Les opérations de provisionnement d'infrastructure étant particulièrement longues dans un contexte *bare-metal* ou serveur virtuel, on réalise généralement ce type de test en *nightly build*.

Parcours général d'une application Web

Si elle est tournée vers la publication de contenu éditorial, cette approche peut s'avérer très pertinente. Faire un *crawling* de tout le site permet de détecter les pages problématiques (qui déclenchent une erreur 500 par exemple) ou les liens cassés.

Tests d'opération des infrastructures

Pour les équipes très matures qui sont prêtes à aller très loin dans l'intégration continue, il est tout à fait possible de tester en continu des fonctions du SI "Opérations" : intégration avec la supervision (tests de collecte de métriques, tests de seuils et de remontées d'alertes…), les fonctions de sauvegarde et surtout de restauration. La restauration n'est en général testée que lors d'un désastre, il est pourtant assez simple de jouer tous les jours une restauration (éventuellement partielle, anonymisée) des sauvegardes de la veille.

[34] Liste des outils présents sur le site OWASP : https://www.owasp.org/index.php/Appendix_A:_Testing_Tools.

Les outils de nos rêves

Est-ce que l'outil d'*Infra as Code* idéal existe ?

N'y allons pas par quatre chemins, l'histoire reste encore à écrire et aucun outil d'*Infrastructure as Code* n'est aujourd'hui parfaitement satisfaisant. Probablement est-ce parce que l'écosystème des solutions d'*Infra as Code* est bien plus récent et pauvre que celui des langages de programmation "classiques". S'il fallait faire sa liste au Père Noël de ce que l'on attend d'une solution d'IaC, nous devrions répondre à plusieurs questions. Et pour chacune d'entre elles, les avis divergent, même chez OCTO.

YAML ? DSL ? Vrai langage ?

Dans l'histoire, différents choix ont été faits sur cette question :

Ansible peut paraître rassurant pour les non-initiés aux langages de programmation. Il ne se compose que d'une syntaxe en YAML, ce qui finalement colle assez bien à des besoins déclaratifs triviaux (90 % du temps en général). C'est lorsqu'il faut sortir de ces 90 % que les limites du YAML se font vite ressentir : il ne s'agit en aucun cas d'un vrai langage structuré, avec des vraies structures (conditions, blocs, mais aussi gestion d'erreur, fonctions...). Le paradoxe est pourtant bien là : si Ansible fait l'objet d'un tel succès, c'est notamment parce que sa syntaxe ne ressemble pas à un langage de développement et ne provoque pas de réactions épidermiques auprès des plus "OPS" d'entre nous. Elle permet de **décrire très simplement l'infrastructure et sa configuration**, à la limite de la documentation d'infrastructure. Le revers de la médaille est qu'il est très compliqué d'écrire des tests réellement unitaires en YAML pour Ansible.

Chef a choisi l'approche inverse en ce basant sur le langage Ruby qui est directement utilisable avec toute sa puissance. Il en résulte une courbe d'apprentissage plus laborieuse et du code qui peut avoir de très fortes abstractions, pour le pire et le meilleur.

À mi-chemin entre ces deux approches se trouve Puppet qui a pris le parti de définir un DSL (*Domain Specific Language*). Il reprend certains des principes de langages de programmation, notamment des mécanismes d'abstraction au travers de "classes".

Dans tous les cas, la puissance de ces solutions, quelles que soient les limites de leur syntaxe, réside dans leur capacité à écrire de façon *ad hoc* des nouveaux types de ressources dans des vrais langages de programmation. Ils sont, par construction, testables de façon unitaire car ils peuvent sans problème faire appel à toute la puissance de Python ou Ruby par exemple, qui disposent de nombreux *frameworks* de tests.

Déclaratif ? Impératif ? Hybride ?

La plupart du temps, une approche purement déclarative convient parfaitement pour les parties d'IaC relatives aux installations et configuration système et *middleware*. C'est au moment où il faut commencer à orchestrer des déploiements applicatifs (sans interruption de service qui plus est) que l'approche hybride

devient pertinente. Elle permet de séquentialiser précisément des étapes, ce qui est nécessaire lorsqu'il faut jongler avec plusieurs machines. L'exemple classique pour représenter ce besoin est la chorégraphie de *rolling-upgrade* entre un *load-balancer* et des serveurs applicatifs.

Push ou Pull ?

La question du ***push*** ou du ***pull*** est finalement moins impactante que le fait de devoir installer un agent sur les machines. Et c'est souvent ce dernier argument qui nous fait pencher pour une approche ***push***.

L'approche ***pull*** est séduisante car elle permet à un système de converger à partir d'un serveur de référence (Chef server, Puppet master), qui certes est un SPOF[35], mais également un point de passage et donc de journalisation de toutes les exécutions d'une solution d'IaC. On apprécie également sa capacité à lisser et répéter son exécution dans le temps en lançant régulièrement des travaux de convergence vers un état attendu.

L'approche *push* a pour vertu de permettre de décider précisément le début, le périmètre (en termes de machines) et la fin de l'exécution d'un travail de déploiement ou de configuration. Généralement peu intrusive sur les systèmes (Ansible ne requiert qu'un service sshd activé et Python installé), cette approche ne demande pas d'infrastructure centrale, complexe à mettre en haute disponibilité. Plusieurs machines avec peu de prérequis peuvent servir pour le déploiement : un poste de développement en phase de mise au point, puis une brique de CI/CD une fois le code versionné.

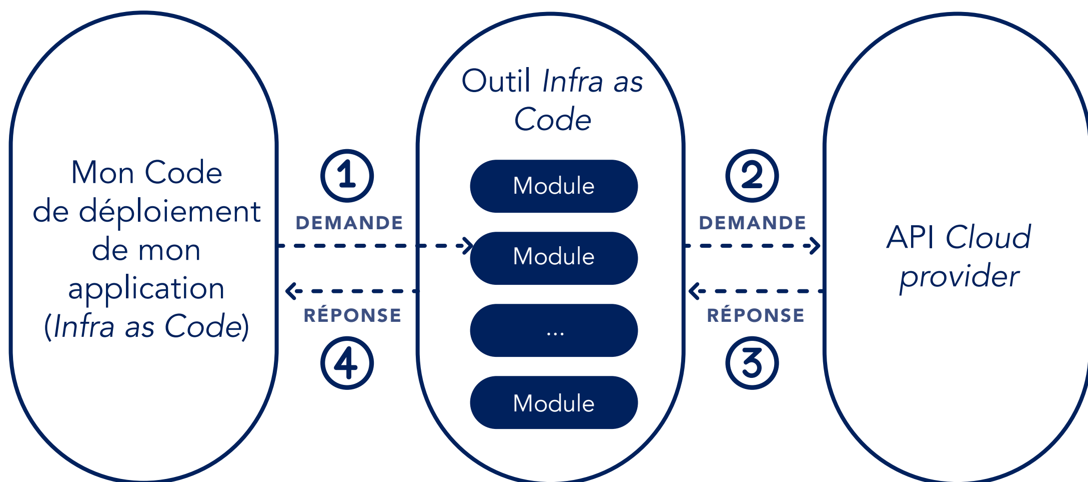
Quel niveau de testabilité ?

C'est surtout sur cette dernière question que l'essentiel du travail est à produire. Là où les langages de programmation classiques sont désormais très bien équipés dans leur capacité à rendre le code testable à tout niveau, l'IaC est bien mal lotie.

› Tester unitairement de la logique cachée dans de l'IaC

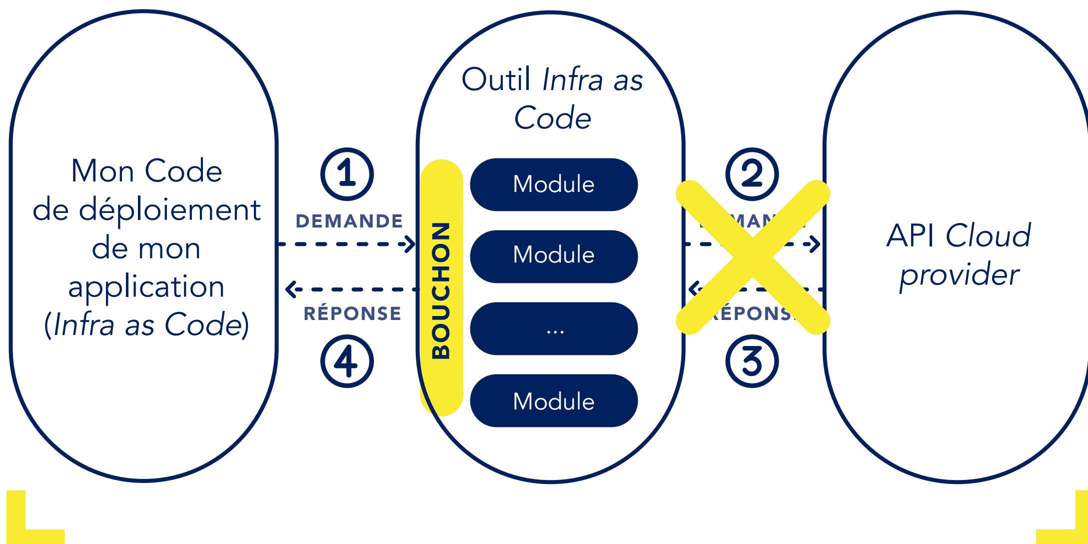
Le besoin se décrit simplement : en fonctionnement normal, notre code d'infrastructure invoque, via des modules, des ressources d'infrastructure. Prenons l'exemple d'une API permettant de créer des VM. Cette API est fournie par un *cloud provider* tiers et un module fourni par la solution d'IaC implémente la logique de comparaison / convergence vers un état attendu et les mécanismes d'idempotence. Problème : cette tâche de création de vraies ressources sur le *cloud* est coûteuse en argent et en temps.

(35) Un point unique de défaillance (*Single Point of Failure* ou SPOF en anglais) est un point d'un système informatique dont le reste du système est dépendant et dont une panne entraîne l'arrêt complet du système - source : Wikipedia.



Notre souhait est donc de gagner du temps et de tester le plus rapidement possible notre code d'infrastructure, à gauche du schéma.

Plusieurs stratégies sont possibles, mais dans l'idéal, on aimerait pouvoir tester en isolation notre code de la façon suivante :



Exit l'appel à la vraie API, et bienvenue au *feedback* le plus rapide possible. Nous restons ici volontairement flou sur l'implémentation de ce bouchon et son emplacement précis, car elle dépend en grande partie de la technologie retenue. Focalisons-nous simplement sur l'intention. La plupart du temps, l'IaC que l'on est amenée à écrire est parfaitement triviale, mais dans certains cas, la logique embarquée peut être beaucoup plus complexe :

- Invoquer un premier module pour récupérer une liste d'objets.
- Pour chacun des objets retournés par le premier appel, invoquer un autre module uniquement si l'objet listé répond à une règle métier spécifique.

Cet exemple décrit un cas très fréquent que nous voulons absolument tester unitairement car de la logique (cette fameuse règle métier) se cache au milieu d'IaC inoffensive. **S'il fallait donner un coup de baguette magique, nous aimerions qu'il soit donné dans cette direction : tester le code métier caché, très rapidement.**

› Écrire et exécuter simplement des tests d'intégration

Derrière cette seconde typologie de tests se cache un second besoin complémentaire : valider simplement que des ensembles de rôles / modules / recettes / classes ont été convenablement appliqués sur tout un système (qui peut être constitué de plusieurs machines).

Les tests d'intégration n'ont ni besoin d'être lancés localement, ni unitairement, mais il est primordial de pouvoir se baser sur le contexte utilisé lors du passage de l'IaC pour déterminer les tests à faire : autrement dit les inventaires, les faits (*facts*) et les variables de chaque machine (`host_vars`, `group_vars` pour Ansible, `hieradata` pour Puppet).

Pour illustrer ce besoin, prenons l'exemple d'une application Web qui est déployée sous une URL particulière et avec un certificat TLS spécifique. Le code d'IaC utilisé est toujours le même quelque soit l'environnement, mais les paramètres (l'URL et le certificat présenté par l'application dans le cas présent) sont probablement différents d'un environnement à l'autre. Nous aimerions que notre outillage de tests soit donc simplement capable, moyennant un inventaire fourni, de récupérer tout le contexte d'exécution pour éviter de devoir re-déclarer les méthodes d'accès aux machines (s'il faut passer par des rebonds sur un bastion par exemple), et surtout les variables spécifiques à chaque environnement.

La mauvaise testabilité de l'IaC est-elle une fatalité ?

On pourrait être tenté de penser que l'on ne parviendra jamais à correctement tester le code d'infrastructure. Pourtant, la plupart des outils qui voient le jour prennent de plus en plus en compte cette problématique de la testabilité. Notre attention est particulièrement portée dans cette direction sur des outils comme Pulumi[36], Testinfra, Test Kitchen, Molecule, pytest-ansible…

[36] Pulumi permet de créer, déployer et gérer une application cloud native et son infrastructure en utilisant un langage de programmation tel que Node.js, Go ou encore Python - source : https://blog.octo.com/decouvrir-les-cloud-native-languages-avec-pulumi/

L'Équipe

***Ptah** : dieu des artisans et des architectes, il est le patron de la construction, des chantiers navals et des charpentiers en général. Il joue également un rôle dans la préservation de l'univers.*

Non seulement je fais des tests, mais je pilote mon développement par les tests. Ma boucle de *feedback* est rapide, et surtout, elle est sûre : aucun environnement n'est maltraité pendant mes développements. Les problèmes de régression que je rencontrais auparavant sont plus limités, puisque mon harnai de tests me permet d'en détecter la plupart dans mon *pipeline*, sur ma plateforme d'intégration continue.

En appliquant à mon code d'infrastructure certaines pratiques issues du *software craftsmanship*, j'ai aujourd'hui quelque chose dont je peux être fier : un code testé. L'infrastructure qui supporte mon application est désormais automatisée, grâce à du code facilement compréhensible, qui décrit l'état dans lequel je souhaite me trouver plutôt qu'une suite incompréhensible d'instructions bash. Bon, il en reste, mais nos nouveaux développements ne se font pas sans test, au moins.

J'ai commencé à montrer un peu comment tout cela marchait à d'autres équipes qui sont curieuses. Les développeurs se sont un peu moqués de nous au début : "personne ne vous croit quand vous dites que vous faites du *craft* sur de l'infra". Mais ils ne peuvent pas nier que notre approche est plus qualitative qu'avant.

"Personne ne vous croit quand vous dites que vous faites du craft sur de l'infra."

Nous aimerions **leur donner plus d'indépendance pour le déploiement de leurs applications**, mais les vieux réflexes ont la vie dure. Déjà, au sein de l'équipe infra, tout le monde n'est pas convaincu par l'utilité de faire différemment et les périmètres de chacun demeurent cloisonnés. Mon code Chef n'est pas forcément compris par tout le monde et si je m'absente plusieurs jours, je peux être sûr que personne n'osera y toucher. Ceci est problématique, car s'il se met à dysfonctionner ou si une évolution est demandée, personne ne pourra avancer avant mon retour.

Certes, le problème n'est pas nouveau. La connaissance ne circule jamais à 100 % lorsque chacun est sur son écran. Même avant d'avoir introduit l'*Infrastructure As Code*, il fallait souvent passer par "les anciens" pour essayer de comprendre les différents scripts, avec leurs variables magiques, leurs paramètres ésotériques… "Heureusement", nous sommes souvent capables de déterminer qui est l'auteur d'un script en faisant un peu attention à la manière dont il est écrit.

Collective Code Ownership

On parle souvent de "s'approprier" le code du projet. En utilisant cette expression, on tient à accentuer le fait de comprendre le code : la manière dont celui-ci est rédigé, dont il va interagir avec les autres applications, ou encore son environnement d'exécution. Mais il faut voir encore plus loin. En tant que membre du projet, je partage ma base de code avec le reste de l'équipe, et je dois être à même de modifier n'importe quelle ligne sans demander avant à son auteur si cela est "bon pour lui". **Ce n'est pas "son code" mais celui du produit et des membres de l'équipe**.

"Le code n'appartient pas à son auteur, mais au produit et aux membres de l'équipe."

Bien sûr, le *collective code ownership* (propriété collective du code) s'articule autour de l'*egoless programming*, de la *Boy Scout Rule* (voir plus loin) et de bien d'autres pratiques décrites ici. Pouvoir modifier le code de n'importe qui, n'importe quand, est un bon indicateur de la **qualité d'un code**. Après tout, le projet n'a-t-il pas des tests de non-régression pour s'assurer que l'on ne déploiera pas un bug en production ? Le code est-il suffisamment clair pour qu'une modification n'entraîne pas un crash serveur à cause d'une obscure commande à lancer, documentée uniquement sur un post-it collé sur le bureau de l'OPS qui est en congé ? Évidemment.

Revue de code

La revue de code consiste à faire relire son code par une personne tierce, afin de relever au plus tôt "ce qui ne va pas". Si les tâches sont divisées en *user stories*, c'est l'unité de relecture que nous utiliserons. Les discussions pourront porter tant sur le fond que sur la forme. Aucune question n'est interdite, puisque le but est de **construire un code de qualité supérieure et compréhensible à la lecture**.

C'est pour nous l'une des pratiques les plus importantes dans un projet. Elle évite le silotage et permet d'assurer un **contrôle qualité[37] du code** qui va ensuite être déployé en production. Flexible, elle peut se faire en asynchrone, seul ou à plusieurs.

[37] Un article à lire sur la *code review* par Jeff Atwood : https://blog.codinghorror.com/code-reviews-just-do-it/.

Le mieux restant tout de même une revue de code à plusieurs, en synchrone.

Qu'est-ce qui pourrait différencier les éléments d'une revue de code d'infrastructure, au sein d'une équipe composée d'OPS, de celles menées sur du code applicatif ? Une attention particulière sera portée au partage des connaissances propres à une expertise : pourquoi avoir utilisé telle ou telle classe de disque dur pour supporter cette application ? Pourquoi laisser ces ports ouverts sur notre base de données ? Pourquoi utiliser ce port exotique ?
"Cette application est très gourmande en IO/s.", "Notre *provider cloud* utilise ce port pour son monitoring.","Le port standard est fermé sur le réseau.".

Une relecture de code d'infrastructure d'un OPS en *feature team* (c'est-à-dire, un OPS au sein d'une équipe de développeurs) aura plutôt vocation à diffuser une partie de la connaissance. Toutes les *stories* ne sont pas forcément éligibles à être relues par l'équipe de développement, il faut trouver le bon niveau de complexité. **Quelque chose d'actionnable, que le développeur pourra facilement s'approprier**. Par exemple, la relecture pourra porter sur le code de configuration des serveurs devant supporter son application. Il est classique que le code d'infrastructure génère des fichiers de configuration pour les applications qu'il déploie. Transférer le savoir permet de répondre à ces questions :

- Comment ce fichier est-il généré ?
- Pourquoi son contenu doit-il être dynamique ?
- Quelle est la syntaxe du moteur de *templating* utilisé ?
- Où se trouvent les variables qui seront utilisées lors de la génération dudit fichier ?

Une fois éclairé, chaque membre de l'équipe pourra, en autonomie, faire évoluer ces fichiers de configuration, au même rythme que l'application en aura besoin.

Egoless programming

Les métiers de l'infrastructure amènent à de nombreux travers. Une erreur et c'est la production qui tombe. On se retrouve en *war room*, un week-end, à restaurer les *backups*. L'impact d'une "coquille" est énorme. Pointer du doigt est un réflexe courant surtout dans le cas d'une nuit gâchée par un "`rm -rf`".

"Le baptême de la prod, c'est de la faire tomber."

Il y a beaucoup d'affect dans le travail et c'est normal, mais avec les enjeux que nous venons de citer, il serait mal avisé d'en mettre plus que nécessaire. Oui, des erreurs seront commises, et par tout le monde. Le "baptême de la prod", c'est de la faire tomber. Certains d'entre nous avons subi ce difficile baptême. Nous avons écrit du code d'infrastructure qui a planté la production. Nous savons combien cela peut être dramatique et savons que le code que nous avons produit ne reflète pas ce que nous sommes.

Pour limiter les erreurs, nous devons éviter l'effet tunnel, communiquer sur ce que l'on fait, et le faire passer par des étapes de validation. Les développeurs, encore une fois, ont eu les mêmes problématiques, et ont su trouver les solutions. Au-delà des pratiques que nous allons lister, il est nécessaire d'adopter une manière de penser le code : l'*egoless programming* ou "programmer en laissant son ego de côté".

En pensant de cette manière, nous essayons de nous détacher un peu plus de ce que l'on a produit. Le code est jugé, et non notre personne. Tout le monde sait qu'on ne voit pas forcément la meilleure solution du premier coup, qu'il y a des contraintes ou encore, que parfois, par manque de temps, le code n'est pas "parfait".

Le *refactoring* est l'un des garants les plus importants d'un code de qualité. Si le code est sanctuarisé, l'amélioration continue est impossible.

Voici les 10 commandements de *l'egoless programming*, tels que décrits sur le blog de Jeff Atwood[38].

1. Intégrer et accepter que vous allez faire des erreurs. *Understand and accept that you will make mistakes.*

2. Vous n'êtes pas votre code. *You are not your code.*

3. Qu'importe l'ampleur de vos connaissances, il y aura toujours quelqu'un qui en saura plus que vous. *No matter how much "karate" you know, someone else will always know more.*

4. Réécrire du code doit se décider à plusieurs. *Don't rewrite code without consultation.*

5. Soyez humble, patient et respectueux avec les autres. *Treat people who know less than you with respect, deference, and patience.*

6. La seule constante dans le monde est le changement. *The only constant in the world is change.*

7. L'autorité engendrée par la connaissance force le respect, plus que le grade. *The only true authority stems from knowledge, not from position.*

8. Défendez vos idées, mais acceptez la défaite avec humilité. *Fight for what you believe, but gracefully accept defeat.*

9. Soyez une personne avec qui il fait bon travailler. *Don't be "the guy in the room."*

10. Soyez dur avec le code, doux avec les gens. *Critique code instead of people – be kind to the coder, not to the code.*

Pair & Mob Programming

Le ***pair programming*** est l'étape suivante. Il est indispensable pour **faire monter en compétence** un nouvel arrivant sur le projet. Ce n'est pas du *shadowing* (qui consiste à se placer en observateur du travail de quelqu'un)

38 Source: https://blog.codinghorror.com/the-ten-commandments-of-egoless-programming/

mais réellement une coopération entre deux personnes face à une problématique. Il permet de partager des connaissances, des méthodes de travail, et tout simplement de créer un moment d'échange qui tisse des liens.

Surtout, cela permet la revue de code en même temps que son écriture. En effet, on ne fait pas travailler les mêmes zones cognitives lorsqu'on écrit ou lorsqu'on lit du code. L'équipier qui n'a pas le clavier pourra donner un *feedback* rapide sur la qualité du code produit. Il est d'ailleurs conseillé d'échanger le clavier régulièrement lors d'une séance de *pair programming* pour alterner les deux postures.

Le ***mob programming*** pousse la pratique encore plus loin. Toute l'équipe se retrouve devant un grand écran et travaille sur une même tâche. Celui qui a le clavier (*le driver*) ne réfléchit pas. Les autres (les navigateurs) indiquent ce qu'il faut faire. Cela a pour bénéfice de **faciliter les discussions et prises de décision par l'équipe**. La connaissance et les apprentissages sont aussi diffusés en **même temps à toute l'équipe**.

D'expérience, le *mob programming* est très bénéfique en début de projet lorsqu'on doit constamment prendre des décisions sur les conventions de l'équipe, les choix de design et les apprentissages fonctionnels et techniques. Il facilite l'amorce des sujets complexes et structurants, qui nécessitent un alignement des convictions et de l'expertise technique.

Clean Code

Au commencement, le code source est écrit. Entre ce moment, et jusqu'à sa suppression, ce code sera souvent lu ; lorsque nous voudrons résoudre un bug, lorsque nous voudrons le faire évoluer, ou juste pour le comprendre. Chacune de ces étapes sera facilitée par un code propre.

Passer du temps à écrire du code de qualité est un investissement, qui, s'il n'est pas réalisé, se paie tôt ou tard. **La non-qualité a un coût, technique, humain, et stratégique**. Notre but ici n'est pas de vous décrire l'ensemble des pratiques permettant d'améliorer la qualité de votre code, mais de présenter quelques pistes, afin de planter les graines d'une **culture durable de la qualité**.

Broken Window

Un tableau de bord dont les indicateurs restent constamment en alerte. Des tests qui ne passent jamais totalement. Un service externe qui tombe régulièrement. Un démon que l'on doit redémarrer occasionnellement, sans trop comprendre pourquoi. Les "erreurs normales" dans les logs... Ce ne sont là que quelques exemples pour illustrer ce qu'il se passe quand on laisse un problème s'installer. Cela témoigne d'une mauvaise qualité du code exécuté, et surtout, d'un mauvais état d'esprit.
S'habituer à passer tous les matins devant une fenêtre cassée ne nous fera pas remarquer la seconde fenêtre qui commence à se fissurer. Et si elle se fissure, après tout, ça ne doit pas être grave, puisque le bâtiment a l'air de tenir debout alors qu'il a déjà une fenêtre cassée. Et d'ailleurs, si personne n'y prête attention, c'est que ce n'est pas si important.

Par exemple, si les erreurs dans le *linter* sont "normales", à quoi sert-il finalement ? Plus aucune attention ne lui sera portée, jusqu'au jour où, une erreur de typographie sera envoyée jusqu'en production...

Un bon moyen de se prémunir contre ces problèmes est de les rendre bloquants pour le déploiement de nouvelles versions. Un déploiement ne doit pas se faire si l'on n'est pas capables d'être un minimum serein sur son bon fonctionnement, via des tests (unitaires, d'intégration, bout en bout…). Et si notre tableau de bord ne remonte que des erreurs, il ne servira à rien, et ne nous permettra pas de détecter un dysfonctionnement au plus tôt comme il le devrait.

"Un bon moyen de se prémunir de ces problèmes est de les rendre bloquants pour le déploiement de nouvelles versions."

Ce n'est pas nécessairement propre au *Broken Window*, mais le *turn-over* dans une équipe est une occasion de se poser les bonnes questions et d'éviter ainsi le syndrome de la grenouille dans la casserole d'eau chaude[39]. Les nouveaux arrivants sont-ils surpris par certaines pratiques ? Se retrouve-t-on à leur expliquer le contexte pour justifier que l'on ne regarde plus tel test ? Tel indicateur ? Il faut en profiter pour échanger sur les sujets, et trouver des solutions satisfaisantes dans l'idéal et appliquées à notre contexte.

BSR – Boy Scout Rule

La *Boy Scout Rule* (loi du scout) consiste à toujours laisser un endroit plus propre qu'on ne l'a trouvé. Appliqué au code, cela consiste à laisser le code plus propre qu'on ne l'a trouvé.

Cette règle vient d'une citation de Baden-Powell, père fondateur du scoutisme : "*Essayez de laisser le monde dans un meilleur état que celui dans lequel vous l'avez trouvé [...]*"[40].

Concrètement, cela peut aller de corriger une simple faute de typographie, jusqu'au *refactoring* d'un module de code entier. Le moment où l'on lit du code pour la première fois, où l'on s'interroge sur son fonctionnement et ses interactions avec d'autres services, est un moment privilégié. Prendre du temps pour retravailler un morceau de ce code aide aussi à sa compréhension.

La *Boy Scout Rule* n'a pas de fin : elle s'applique de façon opportuniste, à l'occasion d'une revue de code, de l'ajout d'une nouvelle fonctionnalité, de la résolution d'un bug...

KISS – Keep It Simple, Stupid!

Faites les choses de manière simple. **Un code non-optimisé, mais facilement compréhensible est souvent préférable à un code spaghetti**, qui fera gagner quelques temps de cycles processeur. L'équipe perdra beaucoup de temps à relire ce code, à le maintenir, à le réécrire…

[39] https://fr.wikipedia.org/wiki/Fable_de_la_grenouille. [40] "Try and leave this world a little better than you found it, and when your turn comes to die, you can die happy in feeling that at any rate, you have not wasted your time but have done your best." - Lord Robert Baden-Powell, 1941.

L'architecture d'une stack applicative doit suivre les mêmes règles, et réussir à faire la part entre simplicité et efficacité. Ajouter des briques, aussi efficaces soient-elles, ajoute de la charge en maintenance, en compréhension, et rend plus compliquée la reprise de la plateforme.

YAGNI – You Ain't Gonna Need It

Développez seulement ce qui est nécessaire aujourd'hui pour répondre aux besoins des utilisateurs et du métier. Gardez les "peut être que l'on en aura besoin" pour demain ! On a souvent tendance à vouloir faire des choses dans l'hypothèse où en aura peut être besoin un jour. En plus de générer du code inutile, ça nous fait perdre du temps et nous empêche de délivrer du code qui apporte réellement de la valeur au projet. **"Le mieux est l'ennemi du bien".**

DRY – Don't Repeat Yourself

Faites la chasse au code dupliqué en utilisant tous les mécanismes de factorisation à votre disposition : modules, classes, méthodes, fonctions. **Pour chaque traitement, ne conservez qu'une représentation unique**. Utilisez-la à chaque fois que cela est nécessaire. Le code s'en trouve d'autant plus limité en taille et donc maintenable. Des outils d'analyse de code produisent des indicateurs sur le taux de duplication du code, taux que l'on cherche à garder le plus bas possible. De plus, les IDE sont généralement équipés de fonctionnalités permettant de repérer facilement les doublons de code[41].

Single responsibility

Évitez de vous retrouver avec un module, une classe, une méthode ou une fonction de plusieurs centaines de lignes. C'est le symptôme d'une mauvaise modularisation du code. Le principe de "*single responsability*" est de dire qu'un module ne devrait avoir qu'une et une seule raison de changer. N'hésitez donc pas à extraire des méthodes, découper des fichiers trop gros. Cela peut paraître paradoxale mais mieux vaut une multitude de petits modules qu'un seul qui regroupe toutes les fonctionnalités : cela facilitera la lecture, la compréhension et l'évolutivité de votre code.

41 Sous la forme d'un plugin pour Visual Studio Code. (https://marketplace.visualstudio.com/items?itemName=paulhoughton.vscode-jscpd) ou dans la version ultimate d'IntelliJ IDEA (https://www.jetbrains.com/help/idea/duplicates-tool-window.html)

Continuous Integration : la nécessité d'une approche continue de la qualité

Il est délicat de mesurer les impacts de la modification d'une grande quantité de code. On pourrait penser que cette démarche ne s'applique qu'aux applications et/ou que les infrastructures et les systèmes, sont stables et ne courent pas ce risque ! Rien n'est plus faux. Pour tout ce qui est codé, configuré, paramétré, un seul changement peut casser quelque chose.

Quels que soient le soin et la qualité que l'on porte à nos travaux, une faute d'inattention finit toujours par arriver. Un espace en trop dans une commande que l'on joue tous les jours qui vide le serveur plutôt que de supprimer quelques fichiers. Un oubli dans l'installation du serveur de production que l'on ne détecte qu'à la mise en prod un mercredi soir à 20h. Un constat s'impose : les humains ne sont pas doués pour la répétition. Il est donc naturel de déléguer ces opérations fastidieuses à des automates qui ne feront pas de faute de frappe et ne peuvent pas "oublier" une étape de la procédure d'installation. **Nous devons laisser à l'humain ce que la machine ne peut pas remplacer : la compréhension des besoins et la conception.**

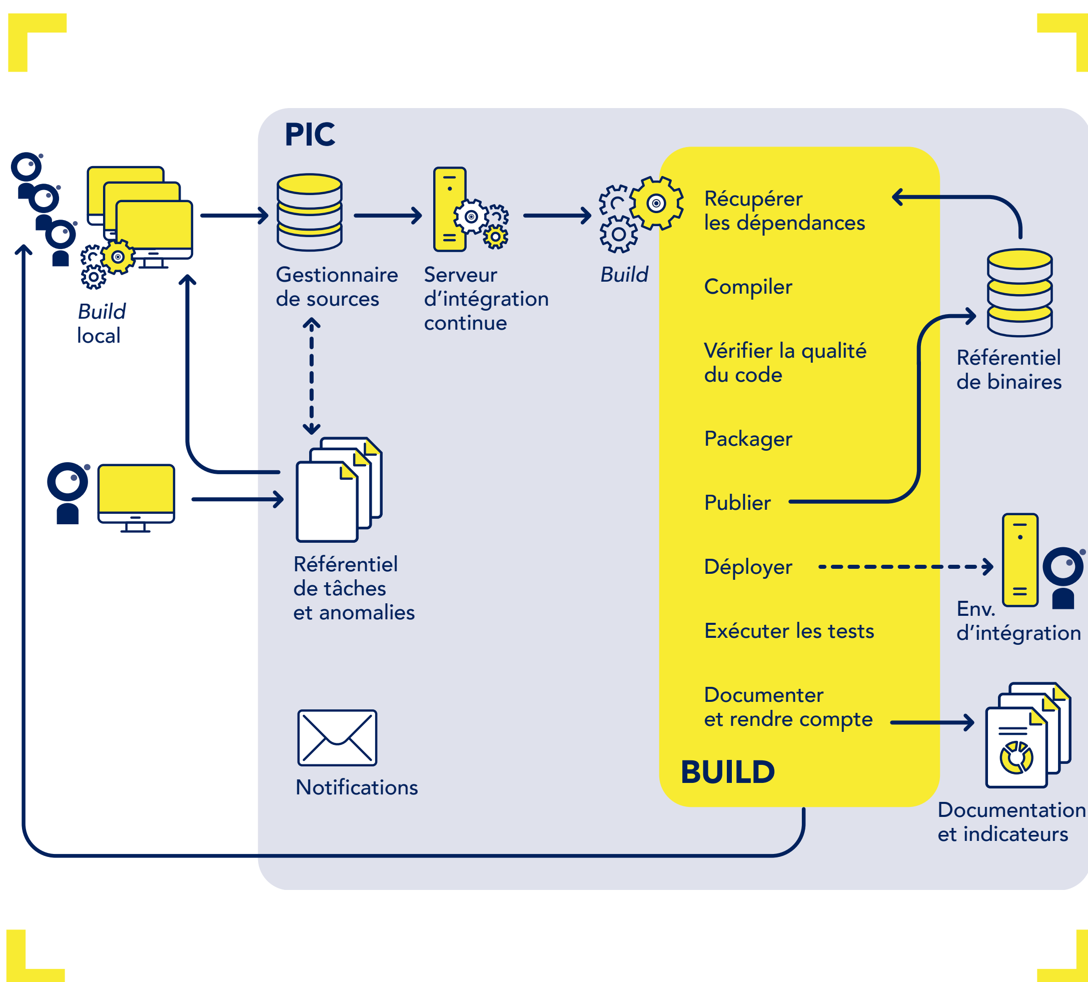
"Nous devons laisser à l'humain ce que la machine ne peut pas remplacer : la compréhension des besoins et la conception."

Les ingénieurs en mécanique en rêvent : imaginez ce monde merveilleux où la chaîne de construction de votre voiture est entièrement automatisée ; quelles que soient les évolutions que vous lui apportiez, vous avez un contrôle technique quasi instantané. Et si c'était possible pour notre infrastructure ?

La genèse : faire du neuf avec du vieux

Les équipes qui font du développement logiciel, en mode agile en particulier, ont la solution depuis longtemps. Elles utilisent quotidiennement des outils que l'on regroupe généralement sous le nom de Plateforme d'Intégration Continue (PIC). Vous pourrez trouver également le terme *Software Factory* ou Usine De Développement (UDD).

Les fonctions classiques d'une PIC peuvent se schématiser ainsi :



Si nous laissons de côté le rôle de **référentiel** d'une PIC (code source / tickets / artefacts), son rôle d'automate se résume ainsi :

- **Tester**
- Construire
- **Tester**
- Déployer
- **Tester**
- Produire des indicateurs de qualité de code
- **Tester**...

L'idée est toujours la même : à chaque fois que quelqu'un propose une modification dans le code d'une application, la PIC exécute et essaye de mettre en défaut ledit code. Dit autrement, la PIC essaye par tous les moyens possibles de faire planter l'application au travers des tests qui lui sont fournis. Si l'application résiste à toute cette somme de maltraitances, c'est qu'elle mérite de la considération et pourrait même partir en prod, qui sait.

Code applicatif

Tests applicatifs

PIC

Artéfact applicatif
déployable avec
de la valeur métier

Degré de confiance
dans l'artéfact
applicatif

En entrée de l'usine, on trouve du code applicatif et des tests, en sortie, un artefact construit et des indicateurs de qualité de ce dernier, voire un environnement d'intégration directement mis à jour. En cas d'erreur, on obtient un rapport d'erreur détaillé sur les causes de cette dernière.

Ce principe implique donc que l'activité d'écriture de code s'accompagne d'une activité miroir d'écriture de code de tests. Votre patrimoine est composé des deux familles de code : application et tests. Ces deux familles de code doivent être maintenues en cohérence. Si l'application évolue, les tests évoluent. Si les tests évoluent, c'est soit pour préparer un changement dans l'application, soit pour ajouter un cas de test qui n'était pas couvert et a donné lieu à un bug.

Importance de la vision pipeline

Pour modéliser l'activité d'une PIC, il est habituel d'utiliser une représentation sous forme d'un *pipeline* :

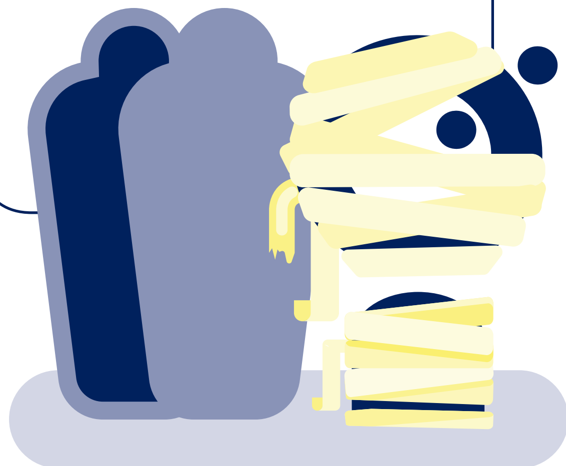
Commit
Analyse Syntaxe
Compilation
Tests unitaires
Packaging
Publication
...
Tests d'intégration
v390
v389
v388
v387
v386
v385
v384
TEMPS

7 versions candidates sont fournies au pipeline. La v384 est la plus ancienne à avoir été soumise, la v390 la plus récente. Parmi les 7 tentatives, 2 seulement (v387 et v390) ont traversé l'intégralité du *pipeline* et sont donc théoriquement éligibles à partir en production. Les différentes étapes de tests sont automatiquement parvenues à mettre les 5 autres versions en défaut. Elles sont donc rejetées et n'iront pas plus loin.

Grâce à la mise en œuvre de ce genre de *workflows*, toutes les tâches manuelles et répétitives sont automatisées. Les tâches exécutées et leur ordre sont ici donnés à titre d'exemple sachant que quelques principes régissent leur mise en place :

- Optimiser la boucle de *feedback* en exécutant les tests les plus rapides en premier. "Quitte à échouer, autant échouer le plus vite possible".
- Ne pas voir le *pipeline* comme un objet sacré, inaltérable. Il est normal, voire sain, de le réorganiser régulièrement : changement d'ordre de certaines étapes, ajout / retrait d'étapes, parallélisation…
- Adapter le *pipeline* au projet concerné. Rares sont les projets qui sont suffisamment similaires pour utiliser exactement le même.

Un pipeline qui n'exécute (presque) aucun test perd environ 99 % de son intérêt. Il permet juste d'envoyer plus vite des bugs en production. Pas de tests, pas de qualité. Pas de qualité, pas de confiance. Pas de confiance, pas de prod.



ANNEXE

Palmarès des plus beaux effets de bord liés à un changement de configuration

Pour cette année 2019, voici notre classement des 10 meilleurs effets de bord. Comme chaque année, ce sont des changements en apparence innocents qui déclenchent les plus beaux loupés.

Merci à tous les participants pour cette moisson très prolifique.
De l'engagement, de la détermination, du mental et du collectif.

10e place : une règle de *firewall* qui bloque un flux NTP. Classique parmi les classiques, cette perle est pernicieuse car elle peut ne montrer son visage sadique qu'après plusieurs heures, jours voire semaines le temps que les horloges internes des machines se désynchronisent au point de gêner les applications.

9e place : passage en mode verbose d'une application : saturation des I/O et du système de fichiers, toujours une valeur sûre.

8e place : changement de la *locale* d'un OS ou d'un SGBD. Un *must* qui peut avoir des effets dramatiques comme casser la fonction de recherche *full-text* en français d'un SGBD. Toujours sympathique de faire le rapprochement entre une recherche qui retourne 0 résultat et un tel changement de configuration.

7e place : changement de la *timezone* d'un OS ou d'une application. Rien de plus joli que de voir des *logs* avec la mauvaise heure, ou une application qui se trompe de jour.

6e place : renommage d'un fichier de *log*, qui n'est alors plus pris en compte par l'outil de rotation de *logs* : le système de fichier va alors se remplir jusqu'à saturer.

5e place : une replanification d'un *job cron*. Classique mais toujours appréciée par les connaisseurs. Mot compte double quand la sauvegarde mal planifiée ralentit tout le SAN aux heures de pointe.

4e place : montée de version d'une librairie système qui était utilisée dans une version spécifique d'un binaire installé à la main en partant du tar.gz de l'éditeur. Ne se voit qu'au redémarrage suivant de l'application et fait toujours son petit effet.

3e place : reconfiguration sauvage d'un *pool* de connexions SQL vers un serveur.

2de place : reparamétrage du nombre de *workers* d'une application qui écroule la mémoire de la machine. Note artistique majorée quand l'application n'est pas simplement tuée par l'OOM *killer* mais mange tellement de *swap* que les I/O mettent à genoux les disques.

1ère place : encore cette année, les durcissements de sécurité remportent la palme. Toujours motivés par une bonne intention, les impacts sont souvent difficiles à identifier au premier coup d'œil à tel point qu'on les considère potentiellement infinis. Cette année, citons simplement le changement de l'umask par défaut (022 ou 002) vers 077 plus restrictif. Entre l'application qui ne démarre plus ou le déploiement qui échoue, vous avez l'embarras du choix et toujours un mal de chien à investiguer. "strace" reste votre meilleur ami dans ce genre de situations.

Pilotage de dette : Principes d'archi

Dette technique ?

La dette technique fait référence aux raccourcis que l'on s'autorise parfois en diminuant la qualité pour pouvoir livrer certaines fonctionnalités rapidement. C'est un mal nécessaire, une dette que l'on contracte et qu'il faudra rembourser plus tard.

Par exemple, lorsqu'un projet commence, un OPS va rapidement mettre à disposition de son équipe une UDD sans l'avoir complètement automatisée, sans avoir géré les *backups* ni le *monitoring*. Il devra dans la suite du projet planifier des tâches avec le *Tech Lead* et/ou le *Product Owner* pour palier ces manques et rembourser la dette technique qu'il a contractée.

La dette technique, pour respecter la métaphore, est **un expédient temporaire pour lequel une solution de recouvrement est non seulement identifiée, mais aussi programmée dans le temps**. C'est un investissement, dans le sens où cela permet de s'acheter du temps.

Comment maîtriser sa dette technique ?

Une équipe qui s'endette (même consciemment) sans planifier son remboursement est vouée à une fuite en avant vers la non-qualité.

Une équipe qui cherche à être agile, créera tactiquement de la dette technique à un moment ou un autre. Et c'est parce qu'elle recherche l'excellence autant que l'agilité que cette équipe utilise ce *pattern*.

À quoi reconnaît-on qu'une équipe fait des "emprunts techniques" temporaires pour s'assurer un avantage (également temporaire) plutôt que s'endetter jusqu'au cou ?

Il suffit d'observer ses pratiques, et d'écouter comment elle répond aux questions suivantes :

- Avez-vous un standard et des principes d'architecture documentés que vous pouvez afficher, ou expliquer via une session en binôme ?

• Avez-vous des tâches de remboursement prévues pour les "emprunts techniques" en cours ?

• Les tâches de remboursement de la dette s'écoulent-elles au même rythme que les tâches standards ?

Si ces questions amènent une ou plusieurs réponses négatives, voici les actions à mettre en place :

• Provoquer un temps d'arrêt pour établir un dialogue autour de ce problème de non-qualité. Tout commence par une prise de conscience de l'équipe.

• Décider quels standards (*d'Infra as Code*) et quels principes d'architecture (exploitabilité, haute disponibilité…) l'équipe veut appliquer dans son travail.

• Confronter régulièrement les écarts à ces standards, via des revues de code ou du *pair programming*, et programmer leur remboursement.

• Mettre en place des formations visant à mieux connaître et mettre en œuvre les pratiques de qualité.

L’Entreprise

***Horus :** dieu protecteur, il incarne l'ordre et l'harmonie universelle*

Nous avons passé le cap de la transformation DevOps ; chaque équipe est indépendamment constituée selon les expertises nécessaires à sa tâche, et est capable de prendre des **décisions structurantes** sans en référer à 8 étages de management. La cohérence et le socle commun sont assurés par des entités transverses.

Nous avons encore un peu de mal à tout assembler, mais nous pensons tenir le "bon bout". Malgré nos pratiques communes, et nos rituels d'amélioration continue, les passages entre les différents processus peuvent créer de la friction, du stock, et donc, des ralentissements. Mais rien à voir avec ce que nous avions précédemment ! Bon, on n'est pas encore Facebook, mais l'équipe d'industrialisation a fait un plutôt bon boulot.

Nous subissons encore les reliquats de notre ancienne manière de faire. Nous manquons d'**harmonisation entre les équipes**, et même si elles ont fait des efforts, les équipes d'infrastructure sont encore trop dans leur monde : elles gèrent le *run* comme si c'était de leur unique responsabilité. Tant que l'on reste "dans les clous" de ce qu'ils proposent, tout va bien, mais si l'on souhaite qu'ils gèrent une nouvelle base de données par exemple, on se retrouve face à un mur. C'est dommage, nous sommes leurs clients, après tout.

Nous avons tout de même évité quelques catastrophes quand une équipe d'architectes d'entreprise a voulu nous imposer une solution de PaaS "sur étagère". Personne n'en avait vraiment besoin, et surtout, la typologie d'application supportée par leur solution ne correspondait à celle d'aucun produit ! Autant vous dire que la licence payée à prix d'or est restée dans son carton. Dans les différentes équipes, les OPS et les *Tech-Leads* ont organisé des ateliers pour démontrer par A+B que le produit n'était pas adapté. Et heureusement, ils ont été écoutés.

Gâchis d'argent, peut-être, mais beaucoup de positif en est ressorti : les OPS et les *Tech-Leads* des différentes équipes ont commencé à se parler, et à échanger autour d'intérêts communs...

Continuous Delivery

N'importe quel changement doit être testé, et ce en continu. Progressivement, ce que l'on appelait intégration continue peut être étendu vers de la livraison continue voire le Saint-Graal : le déploiement continu.

Les gains de cette approche peuvent se résumer ainsi :

• Visibilité permanente du niveau de qualité de chaque version de code (quel qu'il soit, infrastructure ou applicatif).

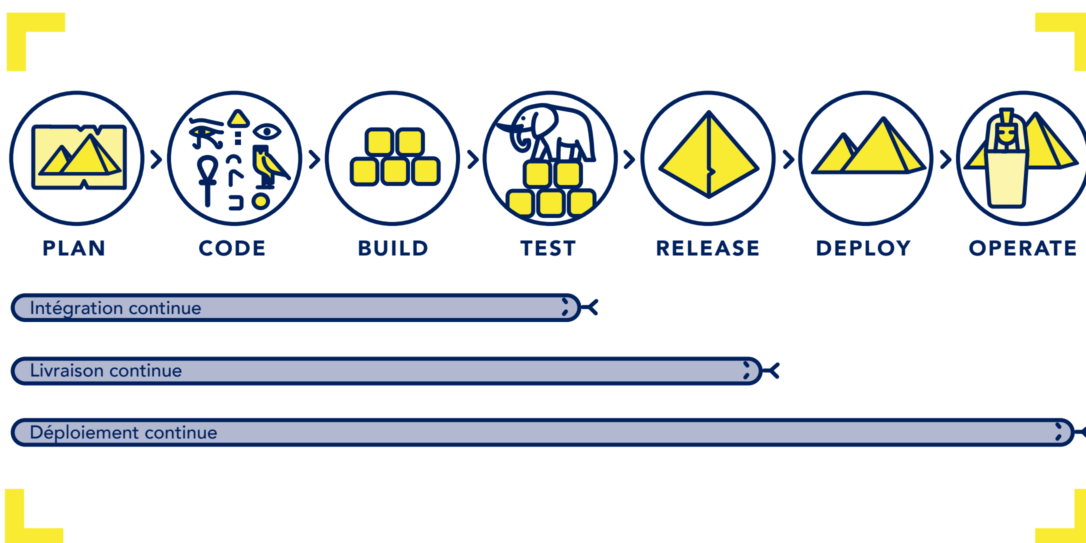
• Filtrage très tôt des versions buggées, qui ne pourront pas fonctionner.

• Confiance dans les processus, car ils sont joués souvent.

• Déploiements plus fréquents, donc avec des changements plus petits.

• MTTR (*Mean Time To Repair*) réduit car changements tracés, versionnés et de petite taille.

• Une preuve de la capacité à reconstruire toute une plateforme, un premier pas vers un modèle de DRP / PRA[42].



[42] *Disaster Recovery Plan* / Plan de Reprise d'Activité.

Un cercle vertueux peut alors apparaître :

Plus petits changements qui apportent de la valeur métier

Moins d'impact unitaire

Meilleure confiance dans les changements

Changements plus fréquents

Pour que ces principes s'appliquent, il faut changer sa façon de penser les interactions avec les machines. Si un changement doit avoir lieu sur une machine, c'est au travers d'une portion de code testé et déclenché automatiquement qu'il doit intervenir. C'est la fin des changements de configuration à la main. En poussant à l'extrême, deux tendances peuvent émerger :

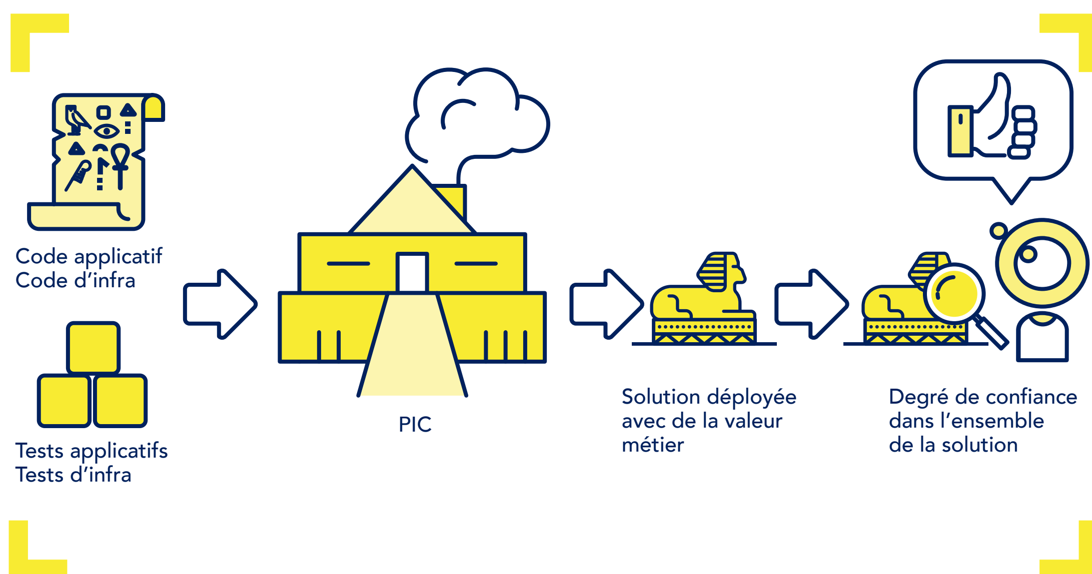
• Proscrire toute modification manuelle sur les machines. Seules les opérations en lecture seule sont autorisées et c'est au travers d'*Infra-as-Code* que les corrections seront déployées.

• Proscrire toute modification d'une machine, principe que l'on appelle également le ***rebuild vs. upgrade***. Si un changement doit être opéré, c'est en reconstruisant une nouvelle capacité (VM, conteneur...) à la place d'une ressource existante et non en modifiant une ressource existante.

Dans les deux cas, les corrections auront préalablement été testées via la PIC avant de donner le "Go".

Principe

L'idée assez simple consiste donc à faire grossir le travail de la PIC pour qu'elle prenne en charge de nouveaux types de travaux plutôt tournés vers l'infrastructure. *Provisioning*[43], déploiement, configuration et bien entendu nouveaux tests sont au programme. Nouveaux outils, nouvelles technologies, mais le principe reste identique.



[43] Instanciation des éléments d'infrastructure nécessaires à notre application.

En entrée, toujours du code, toujours des tests, simplement sur un périmètre plus large. En augmentant ce périmètre, le degré de confiance de l'objet produit augmente. L'objet produit augmente également, ce n'est plus une application, mais tout un système.

La mise en place de ce genre de stratégie gagne en efficacité si l'on est capable de détruire et (re)construire des environnements, dynamiquement. Le *cloud*, les API et toutes les technologies *as-code* sont alors d'un grand secours. Citons quelques services d'infrastructures que l'on aime pouvoir piloter programmatiquement :

- **IPAM[44] :** allocation de réseaux, d'adresses IP pour de nouvelles plateformes.
- **DNS[45] :** gestion des enregistrements des machines, des services.
- Règles de filtrage réseau entre réseaux et/ou entre machines.
- ***Load-balancers* :** matériels ou logiciels, la capacité à les configurer dynamiquement permet de s'assurer de leur configuration, en phase avec les applications gérées.
- **PKI :** j'ai un fournisseur de certificats qui est sollicitable à la demande pour gérer les terminaisons SSL des services exposés.
- **Stockage :** volumes, *buckets*.
- **Machines :** allocation de capacités CPU et RAM (virtuelles voire physiques avec un peu d'entraînement).
- ***Monitoring* :** intégration des machines, services avec les solutions de collectes de métriques et de supervision.
- ***Backup* :** intégration des services dans les fonctions de sauvegarde (et de restauration).
- **Remontée de logs :** capacité à mettre en œuvre une centralisation des journaux sur des machines nouvellement créées.
- etc.

L'accès aux outils n'implique absolument pas un accès open-bar à ces ressources. Il ne s'agit pas de remettre en questions les droits, permissions, quotas. Il s'agit plutôt de donner un accès sur un contexte limité (dev, recette par exemple) aux équipes produits. Elles deviennent alors autonomes pour manipuler toutes les ressources qu'elles souhaitent dans la limite de quotas clairement exprimés (nombre de VMs, de vCPU, de stockage, de sous-réseaux, d'adresses IP, de sous-domaines DNS…).

En fonction des capacités disponibles dans votre contexte, il est possible d'envisager différentes "profondeurs" de reconstruction :

- Les machines sont statiques, il faut les réutiliser. Niveau très moyennement satisfaisant, vous devez mettre en œuvre des purges efficaces, ce qui est toujours compliqué pour s'assurer que l'on n'a bien tout nettoyé. Niveau de confiance : 2/10.
- Il est possible de reconstruire des VMs, volumes... en revanche nous ne disposons pas de levier simple pour reconfigurer le réseau, les règles de pare-feu (pas d'outil simple, pas d'expertise, pas de droit). La purge des VMs

[44] IP Address Management. [45] Domain Name System, permet de traduire le nom d'un serveur (google.com) en son IP (216.58.201.206).

permet une réelle remise à blanc des systèmes d'exploitation. C'est déjà mieux, mais ne permet toujours pas de prouver la maîtrise de la configuration réseau, des *load-balancers* et du filtrage de flux. Dommage, c'est souvent lors de l'intégration de ces composants que les difficultés se présentent et que le temps se perd. Niveau de confiance dans le procédé : 5/10.

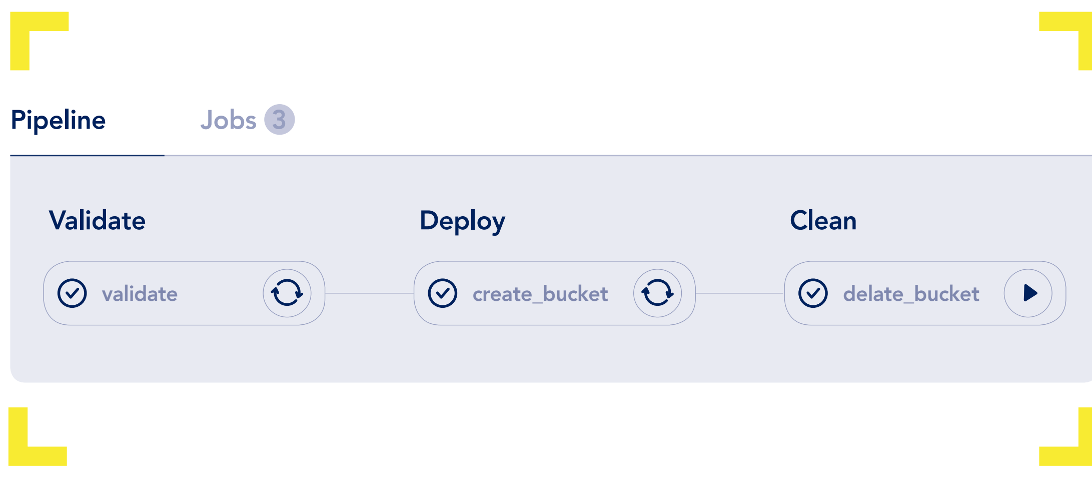
Il est possible de tout reconstruire, du sol au plafond. Que les machines soient à domicile (*on-premise*) ou sur le *cloud*, le procédé de construction est intégralement rejouable à loisir. Niveau de confiance : 10/10.

Laisser de la liberté aux projets

Attention néanmoins, à ne pas tomber dans le piège du "pipeline universel". La vision pipeline d'un produit doit être construite par l'équipe produit, et adaptée à leur manière de travailler. Si vous adaptez un outil, vous serez infiniment plus efficace que si vous adaptez votre organisation en fonction de l'outil. Prenons comme exemple les *workflows* d'intégration continue. Le pipeline d'un *Git Flow*[46] n'aura pas la même tête qu'un *Trunk Based*. De même, une équipe peut choisir de livrer par tag, alors qu'une autre choisira de livrer par branche.

C'est pour cela que nous privilégions les outils qui permettent à votre dépôt de code, de déclarer lui-même la structure de son pipeline. Par cela, nous entendons par exemple, les fichiers "`.gitlab-ci.yml`[47]" ou "`.circleci/config.yml`[48]".

Un exemple de *pipeline* sur Gitlab-ci consistant en la création d'une *bucket* S3 via du code Terraform, en utilisant le nom de la branche :



(46) Comparaison des *git workflows* : https://fr.atlassian.com/git/tutorials/comparing-workflows. (47) Configuration d'un pipeline via un fichier .gitlab-ci.yml https://docs.gitlab.com/ce/ci/yaml/. (48) Exemple de configuration Circle CI : https://circleci.com/docs/2.0/sample-config/.

```
image:
  name: hashicorp/terraform:0.11.11
  entrypoint:
        - '/usr/bin/env'
        - 'PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin'

variables:
  TF_VAR_fqdn: "example.com"
  TF_VAR_subdomain: "${CI_COMMIT_REF_SLUG}"

stages:
  - validate
  - deploy
  - clean

before_script:
  - terraform version
  - terraform init

validate:
  stage: validate
  script:
        - terraform validate
        - terraform fmt -check=true

create_bucket:
  stage: deploy
  script:
        - echo ${TF_VAR_subdomain}.${TF_VAR_fqdn}
        - terraform workspace select ${TF_VAR_subdomain} ||
terraform workspace new ${TF_VAR_subdomain}
        - terraform plan
        - terraform apply -auto-approve
  only:
        - branches
  except:
        - master

delete_bucket:
  stage: clean
  script:
        - echo ${TF_VAR_subdomain}.${TF_VAR_fqdn}
        - terraform workspace select ${TF_VAR_subdomain}
        - terraform destroy -auto-approve
        - terraform workspace select default
        - terraform workspace delete ${TF_VAR_subdomain}
  only:
        - branches
  except:
        - master
  when: manual
```

Prod-awareness : ça veut dire quoi d'avoir bien fait son travail ?

L'opérateur d'infrastructure est aujourd'hui un développeur à part entière. Oui, mais la production n'est pas un environnement local de développement. C'est la vraie vie. Le mode difficile. On ne peut pas dire "c'est bizarre que ça ne marche pas sur les serveurs, sur ma machine, ça marche". Non. Les infrastructures que notre code construit doivent répondre aux contraintes "de la prod".

Être *prod-aware*, c'est avoir conscience de ces contraintes et de ce que les gens qui s'y confrontent - les OPS - vivent au quotidien. Connaître cette sueur froide lorsque l'on se rend compte que l'action que l'on a réalisée a cassé la prod. Adopter une posture "chapeau noir" à la moindre demande d'évolution[49].

La stabilité, c'est aller contre le changement. La simplicité, c'est assurer la maintenabilité. C'est le métier de l'OPS, d'aller vers ce qui causera le moins d'incidents sur sa plateforme. Il est responsabilisé à ce niveau, et se fera réveiller la nuit en cas de dysfonctionnement. Au grand dam des développeurs, qui eux, doivent innover.

"La stabilité, c'est aller contre le changement. La simplicité, c'est assurer la maintenabilité."

Et développeur doit lui aussi évoluer vers le DevOps. Ce n'est pas un chemin à sens unique. Prenons un exemple : une application fonctionnant à l'aide d'un gestionnaire de processus, chargé de la redémarrer à chaque fois que celle-ci est en erreur.

Si l'application en question rencontre une erreur fatale non gérée, cette erreur la fait planter, et le *watchdog*[50] mis en place se charge de la redémarrer, à l'infini. La consommation de ressources s'envole, et l'application est indisponible le temps de son redémarrage.

---

[49] https://fr.wikipedia.org/wiki/M%C3%A9thode_des_six_chapeaux. [50] Un chien de garde, en anglais watchdog, est [...] un logiciel utilisé en électronique numérique pour s'assurer qu'un automate ou un ordinateur ne reste pas bloqué à une étape particulière du traitement qu'il effectue - source : Wikipedia..

Peu de temps certes, mais, manque de chance, cette application est critique.

L'*alerting* se met en marche, et la personne qui assure la permanence est prévenue, et priée de rétablir le service. Il se connecte à la machine, et voit que l'application se comporte normalement. Les logs ne sont d'aucune aide : une *stack-trace* de plusieurs dizaines de lignes. Dans le doute, il peut tenter un redémarrage du serveur. Rendant l'application encore plus indisponible. Mais il ne réussira à diagnostiquer le problème qu'au prix d'une longue investigation. Ce qu'il ne fera d'ailleurs sûrement pas, et préférera faire un *roll-back* pur et simple, restaurant l'application à une version antérieure. Le développeur, le lendemain, ne pourra que constater, et attendre la prochaine mise en production, six mois plus tard…

Nous ne voulons pas cela. Avoir conscience de la production, c'est se demander ce qui va casser en premier, et comment le limiter, ainsi que comment aller plus vite pour diagnostiquer et rétablir le service.

Alors, que voulons-nous ?

Nous voulons comprendre ce que l'on fait : les déploiements doivent être autant que possible standards, indépendants de l'environnement. Un déploiement ne doit pas contenir de valeur ajoutée. Il doit être **réalisable facilement et automatisable**. Cela va de pair avec des déploiements rapides. Un déploiement "lent" est vécu comme un événement, et donc, est réalisé avec parcimonie voire crainte.

Nous voulons des applications observables : qui sont conçues pour faciliter les diagnostiques. Nous voulons des logs, des métriques, nous voulons exposer des données sur leur état de santé. L'état d'incertitude est le pire. De même, nous souhaitons comprendre "pourquoi ça ne marche pas" et non pas "pourquoi ça marche".

L'infrastructure a sa part de responsabilité ; le "*delta surprise*" entre l'environnement de pré-production et de production, les plateformes branlantes sujettes aux micro-coupures, avec des pics de congestions… Bref, "la prod".

Il faut adopter ce nouveau paradigme dans le *Definition Of Done* d'une application "*prod-ready*". **L'entreprise doit offrir aux concepteurs des applications un *background* de pratiques communes afin de parvenir à ce résultat.**

Pêle-mêle, voici quelques principes, dont certains sont une redite des fameux "*12 factor app*[51]" et de notre livre blanc sur les applications *cloud-ready*[52].

Mon application est conçue pour être scalable

Notre application doit être conçue dès le départ pour absorber une charge grandissante. Nous ne parlons pas ici de prévoir dès le départ l'auto-scalabilité de l'application, mais de ne pas fermer définitivement cette porte.

[51] https://12factor.net/ : douze principes à respecter lorsque l'on souhaite concevoir une web-application moderne. [52] Voir notre publication sur le sujet : Cloud Ready Applications : Concevoir vos applications pour tirer parti du Cloud - https://www.octo.com/fr/publications/23-cloud-ready-applications.

Concevoir, par exemple, une application *stateful* nous rendra la tâche plus compliquée.

Nous avions pris pour habitude de mutualiser nos infrastructures, pour des raisons de simplicité, et de coût. Si provisionner un serveur prend trois mois, c'est être agile que de profiter d'un serveur existant pour y installer notre application. Et si en plus, vous devez payer des frais de licences…

Exit les serveurs "fourre-tout", sur lesquels votre application cohabite avec son *reverse-proxy*, sa base de données, et pourquoi pas, d'autres applications. Nous devons la diviser en petits services ayant chacun une responsabilité bien bornée. Grâce à cela, nous scalerons horizontalement (c'est-à-dire en augmentant le nombre de nœuds) le service en souffrance. La rapidité de notre *provisionning* et la mise à profit de l'écosystème gratuit nous permettent cette élasticité à moindre coût.

Mon application est prévue pour être hautement disponible

Un système indisponible coûte de l'argent, mais n'en rapporte pas. On ne peut plus se permettre, aujourd'hui, de traiter l'indisponibilité d'un service comme étant un état normal. Une application moderne n'est pas censée afficher un message de maintenance. Si vous livrez plusieurs fois par semaine (ou, soyons fous, par jour !), vos clients ne doivent pas s'en rendre compte : pour eux aussi, la livraison doit être un non-événement.

"Everything fails all the time[53] *"*

Et pourtant, "tout dysfonctionne, tout le temps". Nous ne prenons plus les incidents comme des événements ponctuels et aléatoires, mais comme un axiome. Notre infrastructure va subir des incidents. C'est comme cela, pas autrement. C'est un pré-requis, qui doit influencer notre façon de concevoir nos architectures : c'est le principe de *design for failure*[54], ou, "conçu pour supporter la défaillance".

> ***"Everything fails all the time.***[53]***"***

C'est la suite de nos fameux "plans de reprise d'activité[55]" et "plans de continuité d'activité[56]" (PRA/PCA). Avec eux, nous prévoyions que quelque chose se passera mal, et un plan à suivre dans ce cas. Malheureusement, cette approche "tout ou rien" demande trop d'effort, et est donc coûteuse à tester. Nombreux sont les PRA/PCA, qui ne le sont que rarement (voire jamais). Et le jour tant attendu où votre *datacenter* prend feu, ils se révèlent inefficaces.

Ce changement de paradigme transforme le principe même d'incident en *business as usual*. Plusieurs niveaux sont bien sûr possibles. Votre infrastructure peut être prévue pour résister à la panne d'un élément isolé, ou de plusieurs

---

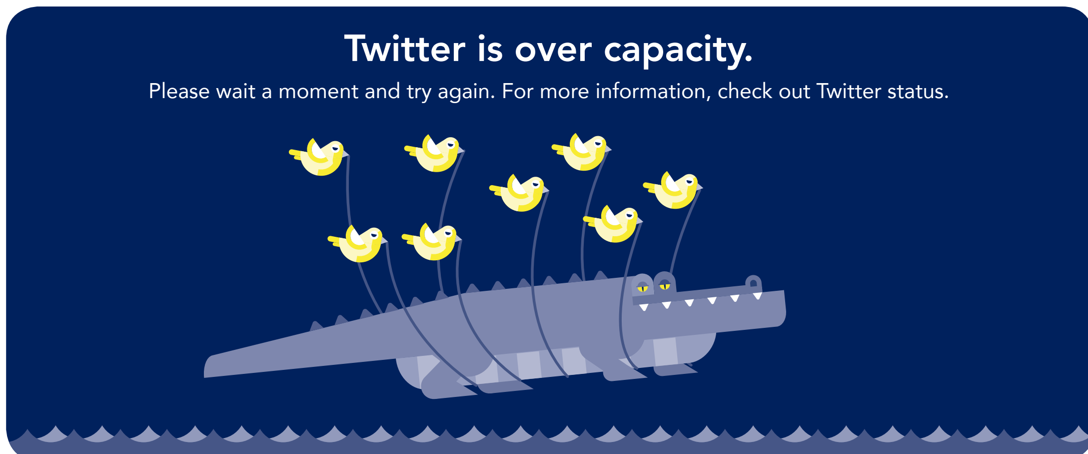
[53] Werner Vogels, CTO d'Amazon. [54] Voir le chapitre : "Plus de capacités pour plus de tests" de cet ouvrage.
[55] https://fr.wikipedia.org/wiki/Plan_de_reprise_d%27activit%C3%A9. [56] https://fr.wikipedia.org/wiki/Plan_de_continuit%C3%A9_d%27activit%C3%A9_(informatique).

en même temps, dans différentes couches. Quand un réel problème arrive, une fois celui-ci résolu, du temps doit être alloué à comprendre comment faire en sorte que cela n'arrive plus. Et si cela arrive quand même, comment faire en sorte que le service ne soit pas ou que peu perturbé.

Le *chaos engineering*[57] est le nom de cette discipline, qui vous permet **d'expérimenter la résilience de vos systèmes distribués**. L'un des exemples les plus connu est celui des *chaos monkeys* : introduit par Netflix, cet automate se charge de simuler (ou de déclencher) des pannes d'équipements au hasard, afin de s'assurer de la résistance du système global.

L'approche du *Testing in Production* est même possible, pour peu que vous ayez une confiance suffisamment grande en votre système : en sa capacité à mitiger les dysfonctionnements, à alerter en avance de phase et à se guérir tout seul. Se rapprocher des conditions d'une vraie production est très dur. C'est d'ailleurs une grande part du travail d'Ops : dérisquer les passages en production en diminuant au maximum le différentiel avec vos environnements "ante-prod". Mais quoi qu'il arrive, vos utilisateurs, eux, ne seront toujours qu'en prod. Il faut donc bien s'outiller : une stack d'observabilité efficace, pourquoi pas du *canary-release*. Pensez dès le départ qu'il faudra créer des comptes "de tests". Une approche DevOps de déploiement continu vous mènera forcément à cela.

Qui que soient vos utilisateurs, ceux-ci attendront de votre produit une qualité de service irréprochable.



La page de maintenance de Twitter, tristement célèbre il y a quelques années.

57 "Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production." - http://principlesofchaos.org/.

⊙ Mon application sait me parler[58]

"L'observabilité est la capacité du système à être appréhendé par l'homme afin qu'il puisse le comprendre, le modifier et le corriger.[59]"

Une application passe, en général, plus de temps en *run* qu'en *build*. Afin que celui qui en assure son bon fonctionnement puisse le faire dans les meilleures conditions possibles, celle-ci doit exposer son fonctionnement de manière compréhensible. Cela s'appelle l'observabilité. Elle se présentera sous plusieurs aspects :

- le *monitoring*,
- l'*alerting* et la visualisation,
- *tracing* des système distribués,
- l'agrégation de logs ou de métriques.

Un log (technique ou métier) est une mine d'or d'informations : c'est également la première étape de la plupart des corrections de bug. Attention toutefois, à ne pas mettre n'importe quoi dans vos logs, et surtout, à ne pas donner accès à n'importe qui à ceux-ci. Les récentes avancées en matière de respect de la vie privée des utilisateurs d'un système informatique (est-il nécessaire de présenter le fameux RGPD ?[60]) nous imposent d'être très vigilants quand à ce que nous conservons : il sera par exemple sage de **chiffrer les données personnelles d'un utilisateur** avant de les conserver sur disque.

En dehors des logs, un autre moyen d'expression indispensable est d'exposer de manière simple des données critiques, permettant de comprendre rapidement l'origine des dysfonctionnements les plus courants, via une URL de *health-check*[61].

```
[ops@dev-api ~]$ curl https://app.com/api/v2/status
{"version":"3.0.1","sha1":"6c0bc125da5d78bb756d 11d01e78aa6e822da
eb","databaseStatus":"up","redisStatus":"up","currentDate":"2018-
11-26T16:44:59+00:00","backendStatus":"up"}
```

Ou encore, pourquoi pas, témoigner de sa charge (ici un exemple avec NGINX) :

```
[ops@dev-rp ~]$ curl localhost:81/basic_status
Active connections: 1
server accepts handled requests
 316945 316945 317598
Reading: 0 Writing: 1 Waiting: 0
```

---

(58) Observabilité – compte-rendu du *talk* de Fabien Arcellier à La Duck Conf 2019 : (https://blog.octo.com/observabilite-compte-rendu-du-talk-de-fabien-arcellier-a-la-duck-conf-2019/).
(59) Décommissionner sans paniquer (https://blog.octo.com/decommissionner-sans-paniquer/). (60) Règlement général sur la protection des données, texte de loi européen visant à harmoniser les lois des différents pays de l'Union Européenne. (61) Un appel sur une URL de *health check* va exécuter un test pour s'assurer qu'un composant rend correctement le service qu'il est censé remplir - source : *Cloud Ready Applications* https://www.octo.com/fr/publications/23-cloud-ready-applications).

Ces indicateurs, *health-checks*, et autres routes d'observabilité rendent votre application exploitable, mais vous permettront également de rendre vos futurs développements plus rapides : l'application n'est pas une boîte noire, mais elle témoigne de son fonctionnement ou de ces dysfonctionnements. Grâce à cela, un Site *Reliability Engineer*[62] sera plus rapide à prononcer un diagnostic.

Plus les indicateurs seront fins, plus nous serons à même de monitorer notre application : via des deltas de performance par exemple, mais également via des données "métiers" : une baisse des commandes sur un site de e-commerce peut tout à fait être la conséquence d'une régression introduite au niveau infra (une nouvelle règle *firewall* qui empêche de contacter l'API de paiement ?)

L'observabilité n'est pas un luxe, ni quelque chose qui doit intervenir "juste avant une mise en prod". Vous devez traiter cela comme un *first-class citizen*[63], au fur et à mesure de vos développements.

Mon infrastructure est sécurisée by design

Lorsque que l'on parle de sécurité, souvent, les esprits se crispent. Cinéma, séries télévisées, actualités… La sécurité informatique semble nourrir plus de fictions que de projets informatiques. Étrangement, la réalité n'est pas très photogénique : des documents de plusieurs dizaines, voire centaines de pages d'une "politique de sécurité informatique". Laconiquement y sont décrits des principes abstraits et irréfragables que tout projet doit respecter.

Avant l'ouverture au public, les équipes de la "Sécurité" mandateront un audit indépendant. Cet audit arrive souvent "trop tard", et les préconisations, sont parfois trop structurantes pour être facilement applicables. Le projet prend du retard, ou se voit carrément remis en question.

62 Le SRE, concept apporté par Google qui consiste à une implémentation du DevOps en rendant les développeurs responsables des opérations et de l'infrastructure. 63 "Objet de première classe", un élément important qui a une place de premier choix - source : Wikipedia https://en.wikipedia.org/wiki/First-class citizen.

Un cycle DevOps classique

PLAN
CODE
REVIEW
TEST
RELEASE
DEV
OPS
PROVISION
SETUP
DEPLOY
OPERATE
MONITOR

Un cycle DevOps perturbé par un audit sur la sécurité des infrastructures

PLAN
CODE
REVIEW
TEST
RELEASE
DEV
STOCK DE FEATURES
AUDIT DE SÉCURITÉ
OPS
PROVISION
SETUP
DEPLOY
OPERATE
MONITOR

La mouvance DevSecOps propose d'intégrer la sécurité dans votre processus de livraison continue. Comme l'agile l'a fait avec les cahiers des charges rigides et à rallonge, comme le DevOps l'a fait avec les délais de livraison et les accidents en production, le DevSecOps veut le faire avec la sécurité.

Les pratiques DevSecOps reposent sur 4 piliers :

- placer les compétences sécurité au cœur des projets,
- faire de la sécurité plus tôt dans la chaîne de développement, en continu,
- avoir une sécurité orientée business,
- piloter par la donnée et des tests réels.

Concrètement, cela signifie que la politique de sécurité ne se passent plus via un document *Word*, mais via des communautés de pratique et des formations : les personnes qui réalisent un projet sont pleinement compétentes pour créer une architecture respectant les principes de sécurité généraux, et prendre les décisions spécifiques à leur projet.

Les outils permettant de réaliser des *scans* de vulnérabilités en mode "audit", type Nessus[64], OWASP ZAP[65], ou encore Wapiti[66] ont encore leur place. Mais leur exécution, trop lente, ne permet pas de les intégrer à notre chaîne d'intégration continue. Ils interviendront de manière régulière, en *nightly* par exemple.

Seront préférés des outils avec des temps d'exécution plus adaptés : scan des vulnérabilités dans les dépendances, avec `npm audit`[67] par exemple pour nodejs, indispensable de nos jours[68]. Nous pouvons aussi utiliser des outils comme Gauntlt[69], qui permet d'écrire des tests de sécurité en BDD *(Behaviour Driven Development)*, comme pour des tests Cucumber.

```
# nmap-simple.attack
Feature: simple nmap attack to check for open ports

  Background:
    Given "nmap" is installed
    And the following profile:
      | name     | value       |
      | hostname | example.com |
```

⑭ https://www.tenable.com/products/nessus/nessus-professional. ⑮ https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. ⑯ http://wapiti.sourceforge.net/. ⑰ https://docs.npmjs.com/cli/audit. ⑰ https://docs.npmjs.com/cli/audit. ⑱ Dernière histoire en date où nous écrivons ces lignes, une librairie nodejs qui, suite à un transfert de propriétaire, a commencé à inclure un *malware* visant à voler les portefeuilles de bitcoin https://www.nextinpact.com/brief/node-js---une-bibliotheque-populaire-verolee-vise-un-portefeuille-de-crypto-monnaies-6747.htm. ⑲ http://gauntlt.org/.

```
Scenario: Check standard web ports
    When I launch an "nmap" attack with:
      """
      nmap -F <hostname>
      """
    Then the output should match /80.tcp\s+open/
    Then the output should not match:
      """
      25\/tcp\s+open
      """
```

Exemple issu de la documentation de http://gauntlt.org.

de plus, ce genre de scripts peut très facilement être réutilisé et partagé au sein d'une même DSI, via des communautés de pratique. Ces outils ne sont que quelques exemples, expliquant comment injecter de la sécurité au plus tôt, dans vos pipelines de développement. Les principes d'automatisation du DevOps s'appliquent ici comme ailleurs : plutôt que de faire des tests manuels pour vérifier que votre application n'est pas sujette à des failles, automatisez ces tests.

⦿ La feature team partage la connaissance de la production

L'organisation de l'équipe doit être telle, que l'infrastructure est collective. Il ne doit pas y avoir un OPS responsable de l'infrastructure. Le pattern de "l'OPS dans la *feature team*[70]", qui consiste à avoir une personne dédiée à construire l'infrastructure n'est pas incompatible avec ce principe. Tant que votre *bus factor*[71] n'est pas égal à "1", c'est-à-dire, tant que la compétence pour gérer l'infrastructure n'est pas concentrée en une seule personne, votre risque est dilué.

⑦⓪ Voir le chapitre "DevOps : vers l'autonomie et la responsabilisation des équipes" de notre publication *Culture DevOps Vol.01* - https://www.octo.com/fr/publications/30-culture-devops-vol-01. ⑦① Le "bus factor" est la mesure du risque encouru en cas de non-partage des connaissances et des capacités des membres d'une équipe, venant de la phrase "si je me fais renverser par un bus".

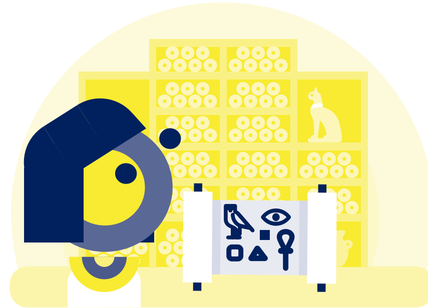
Les anti-patterns et les façons d'en mitiger les effets néfastes[72]

Le super héros
Résout vos problèmes mais… casse tout autour pour le faire.

Le moine codeur
Implémente tout ce que vous voulez mais… dans son coin.

Le grand architecte
Fait un dossier d'architecture pour chaque évolution mais … il se retrouve périmé après chaque implémentation.

Le critique
A des convictions sur tout mais … n'a jamais rien implémenté de concret.



L'archiviste
Refactore le code tout le temps mais … ne fait rien de nouveau.

Ces anti-patterns peuvent être limités grâce à certaines pratiques. Voici la liste des mieux adaptées :
Le super héros : Tests automatisés
Le moine codeur : Revue de code, *Pair programing*…
Le grand architecte : *Documentation As Code.*
Le critique : Retrospectives, atelier d'archi, *mob programming*.
L'archiviste : *Definition Of Done & tests* d'acceptance automatisés.

Il y a différentes manières d'éviter les *anti-patterns* de silotage des compétences. En plus de l'évidente co-localisation des équipes, dans le cadre d'une *feature team*, nous privilégions bien souvent les suivantes :

- **La présentation informelle - *Show me your stack***

Au détour d'un couloir, pendant un déjeuner, prendre le temps d'expliquer son architecture, et les principes avec lesquels on a conçu son application. C'est le niveau 1 du partage, qui s'il se ritualise, peut aboutir à des communautés de pratique.

- **Vis-ma-vie d'OPS**

Pendant une journée, ou plus, un membre de l'équipe de développement jouera le rôle d'un "OPS débutant", mentoré par l'OPS titulaire. Cette pratique lui permettra de se rendre compte des contraintes inhérentes à ce métier et du rythme qui est différent. En faisant travailler ensemble des personnes qui n'en ont pas l'habitude, vous amènerez de l'empathie en plus d'un mélange des savoirs.

- ***Peer review* et *pair/mob programming***

Plus compliqué à mettre en place, car il faut bien choisir les éléments sur lesquels on souhaite "pairer", ou "mober". Néanmoins, cela permet de donner efficacement des clefs pour mieux comprendre les problématiques d'infrastructure.

- **Exercices incendie infra - *Chaos Katas***

Ludiques et didactiques, les "exercices incendie infra" consistent à créer artificiellement un problème, et de simuler une réunion de crise pour résoudre ce problème. Bien évidemment, le mieux est de trouver un dysfonctionnement bloquant (ou déclenchant une alerte) sur un environnement à faible enjeux business (a.k.a., pas la prod), et de surtout, s'arranger pour ne pas avoir le temps d'aider. Vous pouvez tout au plus donner quelques indices.

Pêle-mêle, voici quelques pistes :
- Remplir un disque
- Arrêter une base de données
- `sudo iptables -A OUTPUT -o eth0 -m owner --uid-owner nodejs -j DROP`
  = interdire une interface réseau à un utilisateur spécifique

Ces *Chaos Katas*[73] permettent aux développeurs de prendre conscience de l'importance d'éléments que nous avons cités plus haut : lorsque l'on se retrouve à devoir debugger une application muette, on se rend très vite compte de l'importance de produire des logs !

⑫ Alexandre Raoul, Un peu de craft dans votre infra, 2018 - https://www.youtube.com/watch?v=CapWzdjYF5I. ⑬ Apparu en 2018 dans le Technology Radar Volume 19 de ThoughtWorks - https://www.thoughtworks.com/radar/techniques/chaos-katas.

Le rôle du management : accompagner plutôt qu'imposer

Être le terreau et non le jardinier

Une transformation DevOps est quelque chose de délicat à réaliser. Ce n'est pas simple (et souvent coûteux) pour les équipes d'infrastructure et de développement de changer de modèle. D'autant que pendant que tout cela se déroule, la prod elle, continue de tourner (ou parfois, de ne plus tourner !). Il n'est pas raisonnable de penser, après la lecture d'un article (ou d'une publication), que la qualité dans le DevOps est quelque chose de simple. Permettez-nous d'insister sur ce point : la **qualité dans le code d'infrastructure est quelque chose de récent et de complexe.**

Inventer le train, c'est inventer le déraillement, inventer l'avion c'est inventer le crash[74]

"Inventer le train, c'est inventer le déraillement, inventer l'avion c'est inventer le crash[74]"

Alors comment le "management" d'une entreprise peut-il encourager ses équipes à ses tourner vers les pratiques que nous décrivons dans cet ouvrage ? Notre premier réflexe sera de **former ces équipes**. Les former aux pratiques DevOps et à l'importance de la qualité. Automatiser sans qualité, n'est que ruine de la prod. L'automatisation, et le DevOps, nous permettent de manager efficacement des tailles d'infrastructure toujours plus impressionnantes. Mais en inventant l'avion, nous avons inventé le crash aérien. Là où l'action manuelle a cela de rassurant qu'elle a bien souvent une portée limitée, l'action automatisée, elle est de portée industrielle *by design*. Une erreur dans un déploiement manuel et votre serveur est cassé. Cette même erreur appliquée au code d'automatisation, et c'est toute votre infrastructure qui est indisponible.

Investir dans un "harnais de tests" efficace dans le sens où celui-ci permet de détecter les défauts de qualité le plus en amont possible de la chaîne de production est une nécessité

pour profiter sereinement (et donc pleinement) des gains d'une organisation DevOps.

L'un des freins classiques à la transformation en entreprise est le "trop occupé pour s'améliorer". Appliqué à des équipes de production, cela veut dire être constamment en train de gérer des urgences : c'est ce qu'on appelle le "mode pompier". Le nez dans le guidon, les équipes ne pourront pas d'elles-même demander un temps mort. Il faudra que quelqu'un, dans l'organisation, le fasse pour elles. Quelqu'un qui puisse aller jusqu'à leur imposer de prendre le temps.

Formez-les au développement, au *Software Craftsmanship*. Mettez-leur les mains dans le cambouis d'outils magiques. Une équipe qui revient d'une formation Ansible ou Docker, voudra **absolument** tester ces produits dans son environnement !

Encouragez l'émergence d'une organisation apprenante : faites **du partage** des bonnes pratiques une valeur centrale de votre organisation, en créant des espaces d'échanges. Une "Communauté de Pratiques DevOps" qui se réunit toutes les semaines, à entrée libre, permettra de créer des synergies dans vos équipes. Au sein de ces rassemblements, pas de sujets imposés.

"On n'inflige pas une transformation DevOps : on l'accompagne."

Quelques exemples de discussions :

- Retours d'expérience sur une solution logicielle ou un prestataire.
- Présentation et mise à disposition de "morceaux de code".
- *Mob-Programming* pour tester une nouvelle technologie prometteuse.
- Passage en revue des supports de l'OSSIR[75].
- Construction et généralisation de KPI de qualité d'infrastructure.

Ce cercle vertueux pourra décider d'action d'évangélisation, afin de diffuser largement une vision de la qualité taillée pour votre entreprise. Grâce à ces rituels, identifiez et promouvez des "champions" de la qualité, capables de faire référence sur certaines expertises pointues, comme la sécurité, le réseau...

Le tout est de co-construire. Une direction efficace ne doit pas infliger de l'aide.

Dans le même esprit, attention à ne pas succomber au chant des sirènes des "solutions" DevOps. Certains éditeurs savent se montrer très convaincants quand il s'agit de présenter leur nouveau produit, capable de rendre votre entreprise "*DevOps ready*", "DevOps compatible"... Allant même jusqu'à présenter des KPI : gain en rapidité, baisse d'effectifs des opérateurs...

---

74 Paul Virilio (1932-2018), urbaniste et essayiste français.
75 Observatoire de la Sécurité des Systèmes d'Information et des Réseaux https://www.ossir.org/paris/supports/liste.shtml

DevOps est d'abord une philosophie. Une remise à plat permettant de donner plus de responsabilités aux équipes. Si la direction cherche à leur imposer une solution qu'ils n'ont pas choisi, il y a de très forte chance que cette entreprise ne soit absolument pas "*DevOps ready*".

Former les bon profils

Les profils DevOps sont rares sur le marché et suffisamment chers pour que la solution de les former soi-même soit viable. Au-delà du sujet largement traité du "comment attirer les bons profils", via le cadre de travail et en utilisant des technologies en vogue, nous souhaitons présenter les différentes stratégies possibles.

• Recruter un développeur, et le former à la *prod-awareness* (approche SRE) :

Un développeur fera un bon candidat au poste de DevOps. Il connaît déjà l'algorithmique et les besoins des développeurs d'applications. Il ne manquera plus qu'à le former à la production, à lui faire ressentir ces petits tracas du quotidien des OPS. Attention néanmoins, s'il lui faut rattraper un retard sur l'*adminsys* à proprement parler. C'est un champ de compétences complexes mêlant réseau, sécurité, stockage, connaissances Linux et Windows… Et la mauvaise nouvelle, c'est qu'à l'heure actuelle, acquérir ces connaissances est compliqué.

• Recruter un OPS et le former au *Software Craftsmanship* :

Un opérateur d'infrastructure que l'on forme à la qualité est une autre possibilité. Car pour se former aux bonnes pratiques de code, il existe une multitude de sites internets que l'on peut utiliser : Kata-log[76], codewars[77] ou encore exercism[78] pour ne citer qu'eux. Le point d'attention ici sera à porter sur l'état d'esprit de votre candidat au DevOps : est-il capable de travailler en *pair-programming* ? Trop d'années passées à travailler en mode "moine" (dans son coin) peut rendre complexe la transition à un mode plus collaboratif.

Vous l'aurez compris, le DevOps demande un panel de compétences compliqué à trouver. Il est donc plus facile de ne pas "partir de zéro". Mais quel que soit d'où l'on part, le chemin ne sera pas de tout repos et nécessitera un candidat prêt à remettre en question ses connaissances autant que son savoir-être.

Traiter l'infrastructure comme un produit à part entière

Tout comme l'Agile pousse à migrer d'un mode projet à un mode produit[79], l'infrastructure doit s'extirper de son *run*. Votre infrastructure doit être traitée comme un produit : elle a ses clients, qui sont, au final, les équipes produits : des POs, des développeurs, d'autres OPS… L'infra en tant que produit se doit d'adopter, entre-autres, une vision, une *roadmap dynamique* et un *Product Owner*.

Maintenant produit, l'infrastructure va devoir conquérir ses utilisateurs :

- La qualité sera un critère de choix.
- La satisfaction devra être mesurée.
- *Happy Employees, Happy Customers.*

Paré de ces atours, le socle d'infrastructure sera plus à même de séduire les équipes de développement, et de répondre à leurs attentes. Elles pourront désormais compter sur une interface unique, le Product Owner, capable de répondre positivement ou négativement aux demandes. Grâce à ce changement de paradigme, nous pourrons enfin **mettre l'infrastructure au service des projets et des métiers.**

[76] http://kata-log.rocks/. [77] https://www.codewars.com/. [78] https://exercism.io/.
[79] Passer du mode projet au mode produit : https://blog.octo.com/passer-du-mode-projet-au-mode-produit/.

Annexes

–

Exemple : plateforme de conteneurs

–

Contexte initial

Dans cet exemple, l'entreprise *bidibule* met en œuvre une plateforme d'exécution pour conteneurs ou CaaS[80] pour ses projets à l'aide de Kubernetes. Une approche *on-premise* sur 3 datacenters bien interconnectés. Après un POC[81] plutôt convaincant, l'entreprise a déployé manuellement un *cluster*, l'a intégré à une solution de stockage (*Ceph*) également distribuée sur les 3 DC et montée à la main. Elle a également mis en œuvre un dépôt privé d'images Docker (Registry). Les deux composants (Kubernetes et la Registry) ont été intégrés à la gestion d'identité de l'entreprise. La mise en œuvre de cette architecture, même si elle donne désormais de très bons résultats a été longue à stabiliser : jeunesse des solutions retenues, méconnaissance des technologies tant côté de la production de la plateforme que côté applicatif où l'on mesure difficilement les impacts sur les patterns logiciels à mettre en œuvre. Une pile E/F/K (*ElasticSearch*, *Fluentd*, *Kibana*) a été mise en œuvre pour centraliser les journaux.

Les applications qui sont déployées dans le *cluster* ont des degrés de maturité très hétérogènes. Les projets construisent des images sans réels guides de bonnes pratiques. Certains construisent une image par plateforme, d'autres créent des images qui tirent du code source au démarrage.

Les limites

Plusieurs types de limites sont actuellement atteintes.

Supervision laborieuse : les métriques remontées par le *cluster* et ses applications sont rares ou difficilement exploitables car elles se transposent mal dans l'outil de supervision historique (*Centreon*). Ce dernier est en effet peu adapté à des systèmes distribués, dynamiques, utilisant des conteneurs à durée de vie très limitée.

Journalisation délicate. Il est toujours compliqué de suivre les événements sur Kibana. Difficile de donner du sens à un *imbroglio* de traces qui s'enchaînent sans réelle cohérence.

Des **changements** de version des clusters Ceph et Kubernetes **à prévoir**. Ces montées de version, même si théoriquement possibles sans interruption de service font craindre des ondes de choc douloureuses. Il suffit de regarder la première mise en place de la solution pour s'attendre au pire.

Peu de suivi de l'**obsolescence des images** construites et déployées. Même si mécaniquement, les images exposent une surface réduite face aux risques d'attaque, le RSSI continue à suivre les CVE sur les souches systèmes qui servent de socle aux images avec une certaine anxiété.

---

[80] CaaS : *Container as a Service*. [81] POC, *Proof Of Concept* : preuve de concept.

ANNEXE > Exemple : plateforme de conteneurs

Les promesses de l'approche continue

À l'identique de l'exemple 1, la mise en place d'une approche continue de la qualité doit permettre de contrôler les risques et limites évoqués. Au travers d'un *feedback* permanent, il est possible d'avoir une vision claire sur la situation, les tests qui passent, ceux qui échouent, les scénarios qui sont validés et ceux qui ne le sont pas.

Les changements à opérer

› Double intégration continue

La première chose à concevoir est une intégration à deux niveaux.

Une première intégration continue valide la **plateforme**. L'objectif est de prouver la capacité à reconstruire la plateforme (plusieurs instances), dans sa version actuelle et de tester rapidement les scénarios de montée de version des différents composants. Avec une telle visibilité, la procédure (ou les procédures si en définitive plusieurs étapes successives sont nécessaires) est jouée en automatique de nombreuses fois avant le jour J.

Les utilisateurs de la plateforme sont les opérateurs de service en charge de fournir la fonction de CaaS aux équipes produits (centrées sur les applications).

La plateforme est constituée des composants principaux : *Ceph, Etcd, Kubernetes*. En fonction des instances, sa géométrie est variable : instance simple, représentative de toutes les exigences en termes de haute-disponibilité : simulation de 3 DC. Elle s'accompagne en outre d'au moins une application blanche pour prouver le déploiement correct dans le cluster. Il est possible d'utiliser une application Web par exemple utilisation un volume RBD[82], CephFS, et/ou un stockage objet si ces trois têtes sont exposées sur *Ceph*. L'opérateur du service CaaS est en charge d'écrire des tests d'infrastructure (avec pytest-ansible) pour valider que la plateforme répond aux attentes :

- Configuration de base des composants techniques : présents en nombre suffisant, configurés pour écouter uniquement sur des ports en HTTPS, sans accès anonyme.
- Création des espaces de travail pour les projets (namespaces *Kubernetes*) suivant les principes d'isolation et de nomenclature attendus (pose de labels, ségrégation des permissions entre projets…).
- Tests des procédures de *backup* et de restauration fournies par la plateforme (avec Ark).
- Validation de versions des composants dans le cas où des CVE ont été identifiées.

---

82 RBD : *Rados Block Device*. Mode d'exposition de Ceph au travers d'un périphérique de type block (comme un disque) distant. Une fois mappé sur une machine, il peut être formaté puis monté comme un système de fichiers classique (en ext4 par exemple).

Pour chaque évolution, un test accompagne le code d'infrastructure fourni : soit pour identifier que des mauvaises configurations sont bien corrigées, soit pour tester une nouvelle fonctionnalité qui n'existait pas.

Une seconde intégration continue valide les **applications**. Suivant le modèle classique d'intégration continue d'une application Docker, un pipeline standard (même s'il peut et doit être adapté par projet) est utilisé sur les applications :

1. Compilation et tests unitaires du code.

2. Construction des images Docker à partir d'un dépôt de code et d'un *Dockerfile* en mode "*build once, run everywhere*[83]".

3. Tests "*live*" des images en mode unitaire pour vérifier au minimum leur bon démarrage.

4. Tests d'intégration.

5. Publication des images sur un registre privé après analyse des vulnérabilités (voir ci-dessous).

6. Déploiement sur les environnements suivant une stratégie de tests spécifique au projet. Les environnements sont modélisés au travers de *namespaces* Kubernetes. Ils permettent d'assurer le cas échéant l'isolation des projets, des instances de plateformes et d'une éventuelle ségrégation des permissions entre les intervenants sur le pipeline.

› Ajout d'un outil d'analyse de vulnérabilités

Un outil d'analyse de vulnérabilités est inséré dans les pipelines de construction des images des applications. Son principe : grâce aux bases de vulnérabilités connues dans les paquets des distributions standards d'images Docker (*Centos, Ubuntu, Debian, Alpine Linux*), cet outil permet de vérifier que les images Docker n'ont a priori pas de faille connue. *Clair* de *CoreOS* est une implémentation open source qui convient pour réaliser ce travail. Notez toutefois qu'il s'agit là d'un **outil d'analyse statique** qui n'est en mesure d'analyser que les paquets standards des distributions supportées. Les logiciels déployés autrement (recompilés ou via une archive décompressée manuellement) ne seront pas analysés. Il ne s'agit donc que d'un élément de réponse partiel à tous les enjeux de sécurité de l'entreprise.

Cet outil a donc vocation à être utilisé à deux occasions : **systématiquement** lors de la construction des images, et **périodiquement**. Dans ce second mode d'utilisation, cet outil est également lancé régulièrement sur toutes les images en cours d'utilisation sur Kubernetes. Une menace peut en effet être référencée sur les sites publics (CVE[84]) à tout moment donc (par définition) après la construction des images. Il est nécessaire de **détecter les failles de sécurité** réellement présentes sur le SI.

---

(83) "*Build once, run everywhere*" : principe de construction des artefacts qui les veut indépendants de la plateforme sur laquelle il seront déployés. Les bienfaits de ce principe : les images construites une seule fois dans la P.I.C. sont transportées d'un environnement à un autre sans subir d'altération ou de reconstruction.
(84) *Common Vulnerabilities and Exposures* ou CVE est un dictionnaire des informations publiques relatives aux vulnérabilités de sécurité. Le dictionnaire est maintenu par l'organisme MITRE, soutenu par le département de la Sécurité intérieure des États-Unis - source : Wikipedia.

ANNEXE > Exemple : plateforme de conteneurs

› Repenser le modèles de supervision

L'utilisation de cluster de gestion de conteneurs comme Kubernetes impose de repenser le modèle de supervision historique. Les spécificités du modèle d'exécution de conteneurs (très grand nombre d'objets, durée de vie courte, déplacement d'une machine à l'autre, nouveaux concepts) rendent les outils historiques peu adaptés en standard.

Une nouvelle génération d'outils (citons à titre d'exemple *Prometheus, InfluxDB, Thanos, Grafana…*) sont donc apparus avec comme objectif de répondre à ces nouveaux enjeux (découverte, inventaire dynamique, scalabilité). Ils sont nativement capables de collecter, stocker et grapher des indicateurs liés aux conteneurs.

Plusieurs orientations peuvent être envisagées :

• Coder des connecteurs et agrégateurs pour remonter des indicateurs dans *Centreon*. Compatible avec le format de plugins *Nagios*, *Centreon* est tout à fait capable de s'interfacer avec des *scripts shell* qui réalisent la glue entre les modèles de *Kubernetes* et de *Centreon*. De nombreuses limites apparaissent dans cette approche : difficulté de modéliser les concepts de haut niveau de *Kubernetes* (*namespaces, replicasets, deployments, pods…*), difficulté à agréger des indicateurs distribués (et mobiles) sur plusieurs nœuds, mais offre l'avantage de ne pas trop changer les pratiques…

• Oublier l'outillage historique et basculer intégralement dans le nouvel écosystème. Les outils comme *Prometheus* sont capables de collecter des métriques très diverses, bien au-delà du monde des conteneurs : compteurs du système, applicatifs spécifiques. Si vous êtes prêts à faire le deuil d'une solution comme *Centreon*, vous pourrez vous y retrouver avec des fonctions de visualisation de graphes bien plus avancées, au détriment des fonctions de remontées d'alertes (*alerting*) plus pauvres.

• Avoir une approche **hybride** : garder *Centreon* pour les serveurs et leurs indicateurs traditionnels (CPU, RAM, Disque…) de nœuds et utiliser le nouvel écosystème pour les métriques spécifiques aux conteneurs. Le meilleur des deux mondes, au détriment d'une cohérence de l'ensemble si vous ne faites pas le fastidieux travail d'intégration.

› Adaptation des applications

Les applications ne sont pas en reste puisque plusieurs types de transformations sont proposées.

Architecture *Cloud-ready*. Les équipes des applications doivent être accompagnées pour les aider à percevoir les avantages et les contraintes des architectures dites *cloud*, naturellement disponibles et scalables. La présentation des *12-factor apps* et leur implémentation constitue un conducteur, en insistant sur certains éléments majeurs :

• Séparation des composants à état "*stateful*" et des composants sans état "*stateless*".

• Injection des données d'environnement dans les applications (variables d'environnement ou

fichiers de configuration montés dans les conteneurs).

- Contraintes sur le démarrage et l'arrêt des applications.
- Etc.

Gestion des **métriques applicatives**. En fonction des choix sur le modèle de supervision, l'exposition d'une URL de monitoring peut très simplement ajouter de la valeur en exposant des métriques applicatives. L'utilisation de *Prometheus* côté serveur combinée à la librairie client *Prometheus* côté application (disponible pour tous les langages standards) permet d'exposer et de collecter très rapidement des indicateurs. Un pas de plus vers une meilleure observabilité.

Corrélation des journaux. La normalisation des journaux doit être prise en compte pour permettre de suivre l'intégralité du parcours d'une demande cliente. Si un message entre dans le système et qu'il n'a pas encore d'identifiant de corrélation, un lui est aléatoirement attribué. Tous les appels à des sous-systèmes doivent par la suite propager cet identifiant (dans les entêtes HTTP par exemple). Les composants suivants, si l'identifiant est déjà présent, doivent le propager. Dans tous les cas, l'identifiant doit être mentionné dans les traces (journaux). Il pourra être utilisé par la suite comme un filtre dans Kibana.

Conclusion

Dans ce cas, une architecture ambitieuse est utilisée. Malgré les nombreuses promesses de ces technologies, elles ne peuvent être maîtrisées sans une approche continue de la qualité et un *feedback* permanent prouvant que le *cluster* et les applications qu'il héberge font bon ménage.

Conclusion

Apophis : *dieu des forces mauvaises et de la nuit, personnification du chaos, du mal, de l'obscurité, cherchant à anéantir la création divine.*

Loi de Murphy : *tout ce qui est susceptible d'aller mal, ira mal. Ou, comme l'a exprimé Werner Vogels (CTO d'amazon), « Everything fails all the time ».*
Règle de conception, on ne considère pas la loi de Murphy comme vraie, mais on conçoit tout système comme si elle l'était. Une application doit être à l'épreuve non seulement des accidents les plus improbables, mais aussi des manipulations les plus inappropriées de la part de l'utilisateur.

Les applications évoluent en territoire hostile : l'infrastructure. Ici, la seule loi est celle de Murphy. Longtemps, le plan a été simple pour éviter les vagues : ne pas y toucher. Ou alors le moins possible. Mais les **méthodologies agiles** sont arrivées, obligeant ce monde si stable à se réinventer.

DevOps a forcé les équipes d'infrastructure à se réorganiser : ils doivent désormais parler avec les développeurs d'application autrement que par tickets interposés. Ce contact prolongé a amené de la porosité dans les pratiques : chacun a pu apprendre de l'autre. Les OPS ont alors eu la douloureuse surprise de se rendre compte que leur manière de faire relevait de l'amateurisme, voire de l'inconscience.

Seule mesure de la qualité : l'uptime. Seuls tests : un environnement plus ou moins semblable à celui de production sans données utilisateurs, sur lequel sont testés les scripts de déploiement. **Comment l'informatique a-t-elle fait durant toutes ces années pour ne pas totalement s'effondrer ?** La révolution numérique est au coeur de la transformation de nos sociétés et pourtant soyons honnêtes : la plupart du temps ce n'est que bricolage. **Il devient de plus en plus critique de mettre en place des filets de sécurité.**

L'entreprise doit absolument ne plus succomber à la tentation du *run* au profit du *build*, et encourager l'émergence des bonnes manières de faire : **s'améliorer est un investissement qui rapporte**. Au sein d'une équipe, l'émulation est la meilleure des dynamiques. Les plus belles infrastructures se construisent à plusieurs et ce n'est pas en isolant la connaissance ou en semant la terreur qu'émergera un produit d'exception. Individuellement, la qualité ne doit faire aucun doute. On ne "croit pas en les tests" : ceux-ci sont indiscutables. Il est de la responsabilité collective de bâtir la qualité, chacun doit y participer.

Les nouvelles pratiques de qualité sont encore jeunes, mais progressent vite. Nous construisons encore nos manières de faire et il y a fort à parier que dans dix ans, tout ce que nous avons décrit ici aura été remplacé, obsolète : c'est ce qui fait la beauté de notre métier.

Nous appelons DevOps ces organisations agiles jusqu'en production. Nous appelons DevOps ces technologies en rupture qui accélèrent l'innovations. Nous appelons DevOps ces infrastructures scalables, hautement disponibles, observables et sécurisées.

Le mot de la fin :

⊙ Remerciements :

Un grand merci à tous les contributeurs et relecteurs : Adrien Boulay, Alexandre Raoul, Antoine Tanzilli, Arnaud Jacob-Mathon, Arnaud Mazin, Aurélien Gabet, Aurore Bonnin, Christian Faure, Edouard Perret, Frédéric Petit, Joy Boswell, Laurent Dutheil, Ludovic Cinquin, Meriem Berkane, Nelly Grellier, Nicolas Carron, Salim Boulkour, Tanguy Patte, Victor Mignot et Yohan Lascombe.

La réalisation de ce livre est le fruit d'une dynamique collective incroyable qui dure depuis plusieurs années et qui perdure pour donner vie à ces trois volumes de la trilogie. Ce projet n'aurait pas pu voir le jour sans le soutien actif d'Eric Fredj, ni sans l'inspiration et la solidarité dont font preuve chacun des membres de la Tribu OPS d'OCTO Technology : ADBO, ALR, AMZ, ANTA, ARJA, ARP, ASC, ASI, AUB, AUG, BBR, BGA, BOJE, BONI, CANI, CHL, CQU, CYPA, DABA, DUE, EDE, EFR, EPE, ETC, FRP, FXV, GLE, GLEP, HVLE, JGU, JOJE, KSZ, LBO, LCH, LDU, MAMI, MAT, MBO, MBU, MCY, MDI, MHE, NBO, PYN, RAL, RRE, SBO, SCA, SOUF, TPA, TWE, TWI, VMI, YATI et YOL.

⊙ À propos de l'auteur :

Ce dernier volume de la trilogie a été écrit par Arnaud Mazin, Laurent Dutheil et Victor Mignot, consultants DevOps chez OCTO Technology.

La direction artistique et les illustrations sont l'œuvre de Caroline Bretagne.

"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

Dépôt légal : Décembre 2019

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

© OCTO Technology 2019

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.