DevOps - Loïc Saint-Roch

Plan du cours :
Mercredi matin : TP 3h
TP collectif avec note individuelle

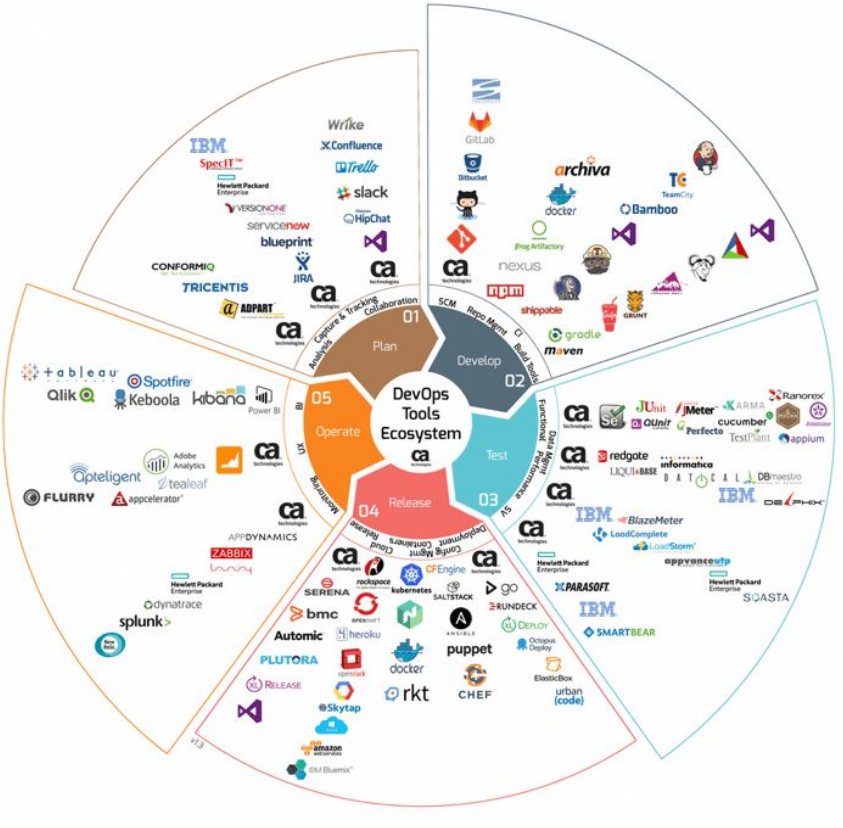
DevOps ?

Dans DevOps on a Dev et Ops (malinx le lynx) :

- optimisation d'infrastructure.
- Tout doit être pensé en User Stories
- Ne pas séparer Dev & admin sys
- Adapter le développement à la prod

Dev + opérateurs qui parlent entre eux ⇒ DevOps
Améliorer/simplifier le processus de travail entre développeur et opérateur.
Les développeurs ne sont pas des sys admin et le contraire n'est pas vrai non plus.

DevOps ⇒ Gestion de projet, workflow qui fonctionne au sein de l'équipe.
=> Plus on a d'outils, plus on a de dépendances, plus le workflow va être lourd.



Tous les outils DevOps

Les parties du devOps :

- *Planification* : gestion de projet
- *Développement* :
- *Outils de test* : tester son code de manière lourde (pas très utilisé)
- *Release*
- *Operation :* log du serveur, monitoring …

Il y a aussi une troisième partie du devOps (après le dev et l'admin Sys), la partie sécurité :

- Parfois géré par l'admin sys (petites équipes)
- Dans les grosses équipes il existe des équipes dédiées à la sécurité
- Elles doivent être intégrées dans le WorkFlow DevOps

JOB ou une CULTURE ?

pour certain, Oui : Ca demande quelqu'un à temps complet sur un projet.

D'après Loïc : En réalité c'est une culture de simplification de WorkFlow en harmonisant le travail entre dev et admins ⇒ Chef de projet technique (DevOps engineer).

Quelle est la différence avec un CTO ?

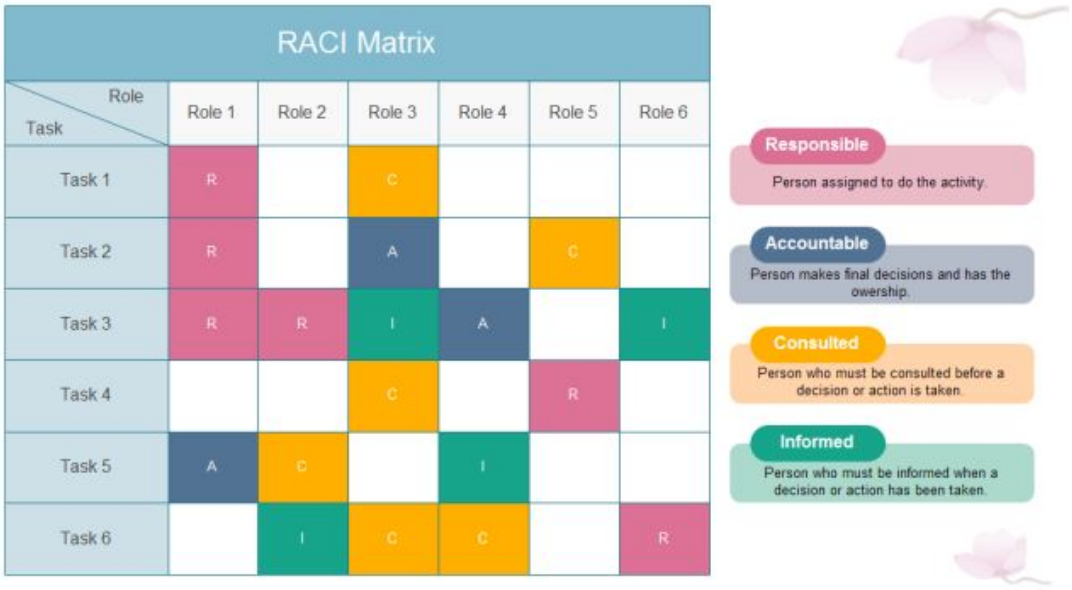
Ça va dépendre du **type d'organisation** que l'on a :

- Équipe de 10 - 15 personnes ⇒ CTO = DevOps Engineer
- Plus grosses structures ⇒ CTO = Il va gérer l'ensemble des projets et les choix techniques (vision plus globale) / DevOps Engineer = Chef de projet technique sur un projet

There is not one path. There is what works.
Pas une seule manière et pas un seul workflow, il n'y a aucune livre, aucune vidéo qui diront comment faire du bon develops. C'est quelque chose qui **s'adapte à chaque projets**, c'est l'**accélération des process**, **enlever les barrière entre les dev**. *(Loïc "DevGod" Saint-Roch, Paris, 13 novembre 2017)*

RACI vs. Tree

RACI : Matrice d'organisation et d'opération



Exemple de matrice RACI

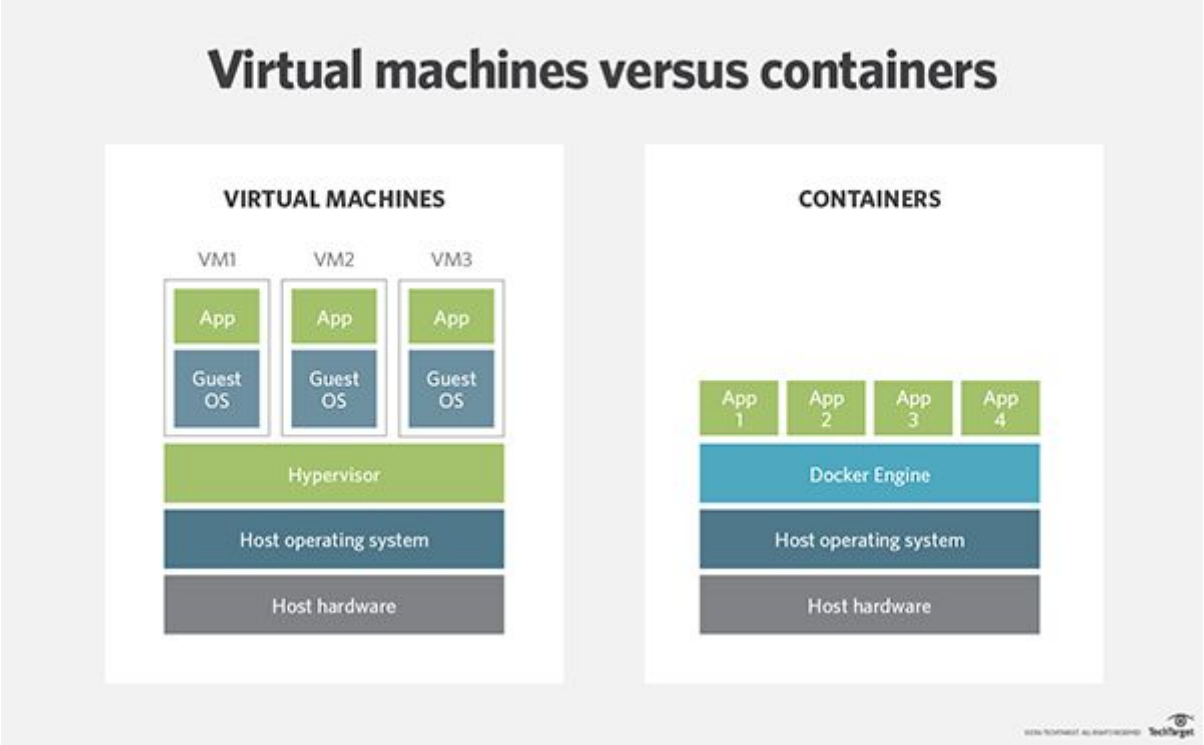
Fonctionnement en RACI efficace pour des tâches très précises.
Un RACI explique concrètement qui a le droit de faire quoi.

Best practice : Mettre un RACI common pour tous les projets et en cas de spécificités en créer en particuliers pour le projet.

Notion et concept à connaître :

Environnement parity : avoir le même environnement en dev, en staging et en prod. (“*si ça marche en dev, ça doit marcher en prod*”)

Comment on peut faire ?



- **VM vs Containers** :

    - Les container ne sont pas des VM, c'est beaucoup plus léger, dans une VM, chaque machine virtuelle est isolée des autres contrairement à des container, le run time docker va lancer un réseau local, les container peuvent se parler entre eux, ils ne sont pas isolé.
    - Ce sont deux méthode différentes mais vont être combiné. Quand on deploy sur le cloud, on a une machine virtuelle créé et dans cette machine virtuelle, on a les runs time docker.
    - Une machine virtuelle ne peut pas parler à une autre par contre les container de chaque VM peuvent se parler.
    - La gestion des VM est beaucoup plus lourde / lente que les container, quand on mets en prod, il y a toujours 3-5 min d'attente pour la lancer. lorsque que l'on va lancer de nouvelle instance on va pouvoir les isoler dans des zones privé. À l'intérieur de ces dernières, on des machines virtuelles qui contiennent des containers.
    - Docker Windows c'est de la merde.
    - Le fait d'avoir notre OPS + hypervisor + docker (ou RKT) va nous permettre d'avoir le même environnement en dev, en staging et en prod => on va pouvoir régler le firewall = plus secure.
    - Si on veut une infrastructure safe = prendre du debian ou du ubuntu.

Artifact :

- C'est le résultat d'un build. par exemple quand on lance gulp, on a un dossier build (exemple : bundle de webpack) ⇒ Résultat d'une compilation
- En opération, un artifact docker est, par exemple, le résultat d'une image docker qui a été compilé.

Services cloud :

- PAAS, SAAS, IAAS.
- Aujourd'hui il y a de plus en plus d'entreprise qui hébergent sur le cloud : le fait d'héberger sur des services à distance.
- API cloud (ex : AWS), il ne faut pas oublier que AWS peut être gérée par une api. simplicité et rapidité et peu onéreux.

Cloud natives vs. Lock-in :

- Une techno qui est **cloud-native** veut dire qu'elle peut fonctionner sur n'importe quelle infrastructure.
- Une techno **Lock-In** veut dire qu'elle utilise un service spécifique à un hébergeur (dynamoDB est une BDD NoSQL sur AWS, très compliqué de faire une migration ailleurs car on va passer sur un autre cloud provider).

Multi-cloud :

- Être multi-cloud, ça veut dire qu'on a héberger notre app sur different cloud provider.
- Les gros services sont sur plusieurs cloud.
- Être multi cloud conserve l'availability (#ovhgate), garde accessible notre application peut importe les problèmes des cloud providers.

Scheduler :

- Process qui va nous permettre d'exécuter des tache toutes les 10 minutes, toutes les heures etc. (exemple : toutes les 24 heures je fais une sauvegarde de ma base de donnée).

Serverless :

- Il y a un serveur mais ce n'est pas nous qui le gérons la scalabilité et la sécurité sont gérées.. (exemple : AWS lambda, azure cosmos, aws dynamo).
- c'est le cloud provider qui va s'occuper de la sécurité etc. de notre application.
- **functions as a service** :
    - pas de port à gérer, ce sont des config sur AWS que l'on va gérer.

- Par exemple quand on va faire une requête sur post tweet, ça va exécuter cette requête, juste besoin d'écrire la fonction :

- Il faut gérer l'infra le plus tôt possible. Il y a une énorme différence entre faire l'infrastructure cloud et le serveur less. On peut économiser beaucoup d'argent. c'est quelques chose de très sérieux en dev et en marketing. **Attention** : des boîtes coulent financièrement à cause d'une mauvaise infrastructure.
- **A ne pas confondre :** Tous les serverless ne sont pas forcément des functions as a service.

Que choisir ?

- Back pour les objets connecté (tout ce qui est IOT) => serverless plus adapté et moins chère.
- Apres ça depend du type d'application et du nombre de visiteur que l'on a.
- Ce sont des calcul qu'il faut faire en amont.

Calcul :

- pour le functions as a service : AWS va scaler les fonctions. Une fonction appelée 50 fois à la minute, elle sera appelée 50 fois par minute et on paye à la minute. On paye uniquement le temps d'exécution.

High-Availability :

- Faire en sorte que l'app soit tout le temps disponible.
- Plusieurs façons :
  - Multi cloud.
  - serverless.
  - instancier plusieurs nodes.
- On ne le fera jamais manuellement, c'est le cloud provider qui s'occupe de tout.

Tests unitaires et tests fonctionnels :

- **Test unitaire** : on teste fonction par fonction. une fonction est égal à minima à un test.
- **Test fonctionnelle** : rédaction de test lié à des users stories. (une story appelé new User = une dizaine de test)

Test-Driven Development :

- Faire du TDD, écrire les test avant d'écrire les fonctions.
- On va écrire les test, pour valider le résultat attendu puis on va écrire les fonctions.
- **Long** ⇒ 30 à 40% de temps en plus (dépend du langage ou framework).
- **Avantages :** une fois les tests, écrit, écrire les fonctions est très rapide.

Immutability :

- **immuabilité** : incapacité de modifier un serveur existant.
- imaginons que l'on a des setting sur notre serveur et qu'on modifie un réglage. l'immuabilité fait que l'on ne va pas modifier l'ancien serveur mais on va en créer un nouveau et une fois qu'il est fonctionnel, on va supprimer l'ancien. **Par contre**, pour des réglage beaucoup plus important, on va détruire le serveur et en recréer une autre.
- **Avantage** : on connaît exactement les réglages de notre serveur.

Backup, snapshots and replications :

- **Snapshot** : sauvegarde de toutes les données de notre base de donnée. on va pouvoir programmer des snapshots tous les jours à 17 heure avec un scheduler.
- **Backups** : on récupère les data mais derrière on va en faire quelque chose (les mettre dans de l'analytics, les envoyer à un CRM).
- **le point en commun backups / Snapshot** : ce n'est pas du temps réel (toutes les heures / demi-heures etc.)
- **Replica** : théorème des BDD ⇒ CAP theorem (consistency, availability, Partition tolerance). **Réplication** c'est répliquer une BDD de manière consistante
    - la consistance c'est le fait de toujours avoir la bonne version de nos donnée. Si on a plusieurs nodes, la **consistance** dis que l'on aura la même donnée sur les différents node.
    - une base de donnée n'a que deux caractéristiques sur les trois.

CI / CD :

- **Intégration continue :** Quand on push sur une branche en particuliers, quelque chose se passe au niveau des tests, on va tester l'environnement de staging, les tests unitaires.
- **Déploiement continue :** Dès que les tests passent en staging , on déploie en prod

Monolithics vs Microservices :

- Un seul serveur avec une seule app -> monolithic (ex: l'app node du tp)
- Avoir la même intégration continue sur tous les microservices.
- A quoi ça sert de gérer une application en microservice ?
    - Payer que ce qu'on utilise (functions as a service).
    - Faire un RACI par service ou par groupe de service.
    - on sait exactement sur quoi on doit travailler et sur ce qu'on ne doit pas travailler.

Mono-repo :

- Avoir tous les micro services et toutes les applications sur un seul repository.
- Notre code est découper en services, eux même fragmenté en fonction.
- **Avantage** : un seul repository à gérer.
- **Conseil** : Lancer toujours tous les tests sur chaque micro-services

Infrastructure as code :

- Aujourd'hui sur heroku, on gère notre application via le dashBoard.
- **IAC** : Ce sont des fichiers de configuration qui vont permettre de mettre en place un infrastructure sur un cloud provider
- Ce qu'on va utiliser à partir de demain, c'est un outil qui s'appelle Terraform (https://www.terraform.io/) .
- c'est quelque chose que l'on doit maîtriser car on va nous le demander lors de workflow de devOps.

Notre mission lors de devOps :

- Pas de sécu, pas de sécu.
- Un développeur qui ne sait pas mettre en place une pipeline sur heroku, c'est pas normal, pareil pour le calcul de l'infra la moins onéreuse.

Les produits AWS :

Plus de 75 services en tout. Pour des projets normaux, on va se servir de beaucoup de services présentés ci dessous.

- **Compute** :
    - **Amazon EC2** : elastic cloud computing ⇒ EC2 va nous permettre de déployer des apps sur amazon. AWS EC2 va nous donner le hardware, c'est à nous de déployer le software.
    - **Amazon EC2 container Registry** : liste des container docker dispo pour installer sur Amazon EC2 container service.
    - **Amazon EC2 container Service** : déployer des container docker.
    - **Amazon VPC** : Virtual private cloud ===> isoler des ressources du monde extérieur. ex : une instance amazon EC2 va vivre dans un amazon VPC pour la protéger.
    - **AWS lambda** : functions as a service d'AWS. (premier million de requête par mois gratuit)
    - **Auto Scaling** : fonctionnalité d'AWS que l'on peut plugger sur d'autres service. elle va scaler des ressource (BDD, instances EC2).

- **Storage** (ça nous sert pour uploader des fichier) :
    - **Amazon Simple Storage Service (S3)** : va nous permettre d'uploader de fichier, image, PDF, audio, CSS, html etc. On déploie sur S3 des assets statiques.

- **Database :**
    - **Amazon Aurora :** version customiser de MySQL qui auto scale.
    - **Amazon RDS** : MySQL mais pas d'auto scalling.
    - **dynamoDB** : base de donné serverLess NoSQL de AWS. (ne coupe rien en dessous de 20 lectures par seconde).
    - **Amazon RedShift** : base de donné avec moteur PostGresSQL mais on ne sert pas de redShift comme base de donné transactionnel. On peut avoir une BDD qui nous sert a sauvegarder toutes nos datas de notre base de donnée transactionnel (par exemple amazon aurora). Toutes les requêtes SQL analytics, on les fait sur redShift.

- **Networking :**
    - **Amazon route 53** : service de gestion de domaine et de DNS d'amazon.

- **Management tools :**
    - **Amazon cloud Watch** : monitorer nos instance EC2, nos fonctions, nos BDD etc.
    - **AWS Cloud Formation.**

- **Security, identity and compliance:**

    - **AWS IAM :** gérer les users et les ressources d'AWS.
    - **AWS HSM** : gérer les normes bancaire.

- **AWS Cost Management :**
    - **AWS cost explorer** : suivre le tarif de chaque ressource par mois.
    - **AWS budgets** : permet de fixer des budgets limités.

- **Application services :**
    - **Amazon API gateway** : permet de gérer la sécurité pour les instances EC2 et les fonctions sur AWS lambda.

- **Messaging :**
    - **Amazon SNS** : notification push en fonction event sur AWS
    - **Amazon SES** : notification email en fonction event sur AWS

Pour le calculateur de prix d'AWS : https://calculator.s3.amazonaws.com/index.html

Google cloud platform :

- **compute** :
    - **compute engine** : équivalent AWS EC2 (IAAS)
    - **App engine** : équivalent d'heroku (platform de PAAS)
    - **Kubernetes Engine** (container engine en français) : pour les docker (tous les dockers sont orchestrés par kubernets)
    - **Cloud Functions** : equivalent d'AWS lambda (functions as a service).

- **Management tools**

- **Outils de bg DATA :**
    - Cloud Pub/Sub : Meme fonctionnement que les web Socket.

- **Networking :**
    - Gestion des DNS
    - gestion des VPC

- **Identity et security :**
    - IAM (identity access management)

- **Storage et BDD :**
    - **cloud storage** : AWS S3
    - **Cloud SQL** : Equivalent de AWS RDS, on va pouvoir mettre en place une BDD de type mySQL ou Postgres.
    - **Cloud Bigtable** : BDD NoSQL pour analitycs

- **Cloud Spanner** : BDD SQL distribué (serverless) qui ne peut pas tomber.
- **Cloud Datastore** : equivalent de dynamo DB.

Chez google, toutes leurs apps sont fait en go et fonctionne sur kubernetes.

Microsoft Azure :

- **MAchine virtuelle** : VM (IAAS)
- **App service** : équivalent de App engine sur google cloud
- **SQL DB** : équivalent de SQL AWS
- **functions** : serverless.
- **Azure cosmos DB** : service NoSQL qui va nous permettre de déployer une base de donnée NoSQL serverless.
- **container registry** : gestion des container docker.
- **Key**
- **Vault** : gestiond e nos clé d'encryption. Nos app ne sont pas consciente des clé d'encryption que nous avons.

Comment choisir ?

Tout dépend de ce dont on a besoin.

Supposons que l'on a une base de donné mongoDB, plusieurs solutions :

- Cosmos pour la DB / VM & container : Azure, AWS ou GCP
- Azure : 2 VM (une avec Mongo et une avec notre app)
- Calculer combien va nous couter une BDD Mongo avec notre trafic actuel, dans 1 an, dans 3 ans et dans 5 ans. Sinon, on calcul une autre solution que Mongo DB. Est ce qu'il est plus avantageux de rester sur Mongo ou alors de switcher de techno ?

hashiCorp :

- **Terraform** : écrire de l'infrastructure (ainsi que de la versionner par environnement).
- **Vagrant** : faire des VM très facilement.
- **packer** : image unique pour tous les cloud providers.
- **Vault**
- **Consul** : faire du service discovery. gérer des config partager pour tous nos microservice.
- **Nomade** : lancer n'importe quelle tâche sur n'importe quelle plateforme.

Terraform :

- C'est une ligne de commande qui va nous permettre de gérer des ressources Cloud (gestion de pipeline, gestion de VPC etc.).
- On a des fichiers de config et en fonction de nos fichiers de config, ça va appliquer une infrastructure serveur.

- Apres les lignes de commande, on a les providers que l'on va chosir en focntionde chaque projet (1 pour gitLab, un pour AWs etc.)
    - ces provider ont des ressources