Systèmes distribués

Filière INDIA (3ème année)

Pr. Abderrahim EL QADI

Département Mathématique appliquée et Génie Informatique

ENSAM-Université Mohammed V de Rabat

A.U. 2023/2024

Partie 3 : DevOps

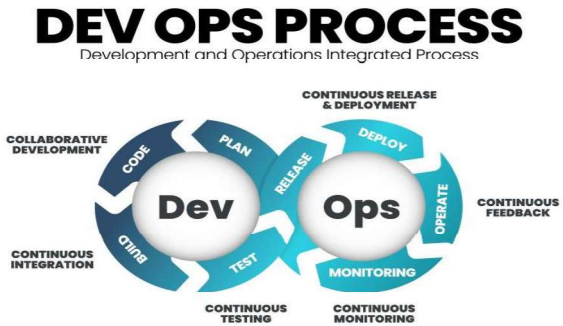
Références

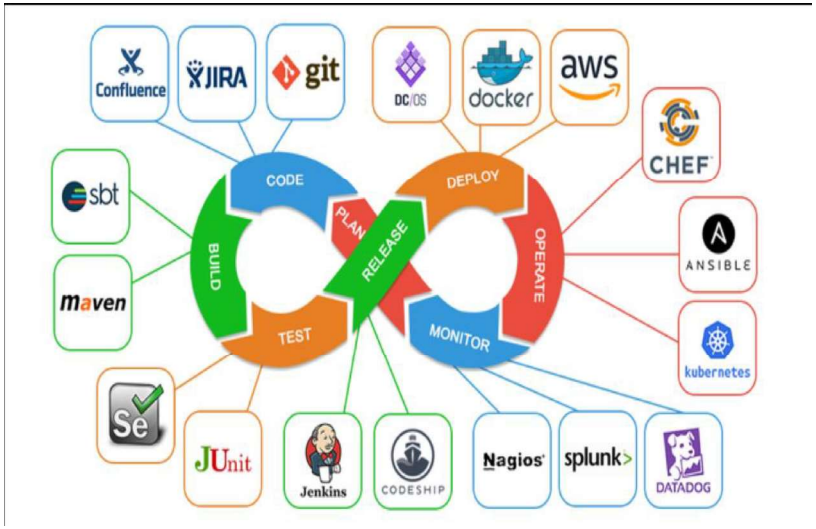
– DevOps, Intégrez et déployez en continu, Edition Eni
– DevOps 4 Null. Une introduction à devops
– DevOps Introduction. Thomas Ropars (https://tropars.github.io/)
– https://azure.microsoft.com/fr-fr/overview/what-is-devops/#culture
– https://aws.amazon.com/fr/docker/
– https://www.jenkins.io

1. DevOps (Développement et Opérations)

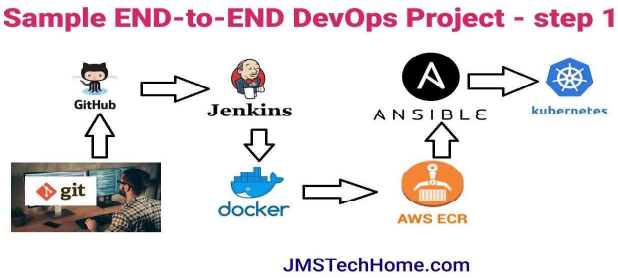
Dev: Equipes de développeurs logiciels
Ops: Equipes en charge de la mise en production des produits

Le terme "**DevOps**" est composé des termes "développement" et "opérations". Il s'agit d'une pratique visant à fusionner le développement, l'assurance qualité, et les opérations à savoir le déploiement et l'intégration en un unique ensemble de processus continus.



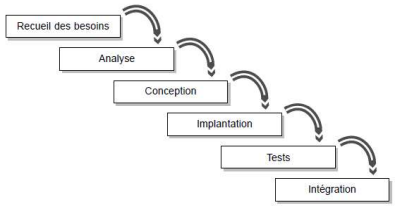


- DevOps s'appuie sur l'intégration et le déploiement en continu **(CI-CD)**
  - **Intégration continue :** Une méthode de développement logiciel dans laquelle le logiciel est reconstruit et testé à chaque modification apportée par un programmeur.
  - **Livraison continue :** est une approche dans laquelle l'intégration continue associée à des techniques de déploiement automatiques assurent une mise en production rapide et fiable du logiciel.
  - **Déploiement continu :** est une approche dans laquelle chaque modification apportée par un programmeur passe automatiquement toute la chaîne allant des tests à la mise en production. Il n'y a plus d'intervention humaine.



2. DevOps et le cycle de vie des applications

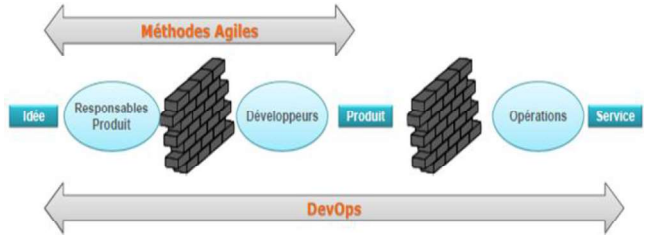
– Cycle de vie classique des applications



- Avec l'approche classique en cascade du cycle de développement d'un logiciel, le coût lié à la correction d'un défaut augmente à chaque étape.
- En détectant les problèmes plus tôt, on crée des logiciels plus efficaces, plus rapidement et avec moins d'efforts.

Vision classique	DevOps
60% de programmation	25% de programmation
20% de tests	10% de tests
10% d'intégration	10% d'intégration
10% de documentation	15% de documentation
	10% automatisation des tests
	10% automatisation du déploiement
	20% qualité/validation

DevOps est une démarche de collaboration agile entre développeurs, équipes opérationnelles et métiers (Business) sur l'ensemble du cycle de vie du produit, du design au support en production, destinés à fournir continuellement de la valeur aux clients.



– Méthode Agile :

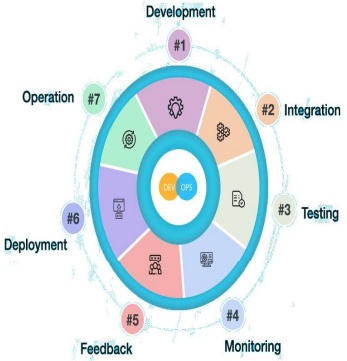
- Agile est conçu autour d'un processus itératif qui répond rapidement aux besoins du client.
- Peu importe la rapidité et l'agilité avec lesquelles effectuée le processus de développement, du fait du manque de collaboration entre les équipes de développement et de production, il est fréquent que les applications ne soient pas prêtes lorsqu'elles arrivent en production.
- L'idée consiste à déployer des changements incrémentiels fréquents au lieu de les stocker en vue d'un grand lancement.

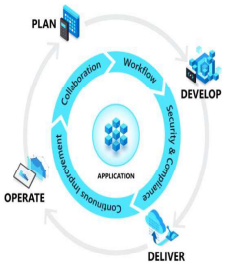
– DevOps et le cycle de vie des applications

- Devops est un mouvement qui étend Agile en intégrant les opérations dans les itérations
- Elle influence le cycle de vie des applications tout au long de leurs phases de planification, de développement, de livraison et d'exploitation.

Ces phases reposent les unes sur les autres et ne sont pas spécifiques à un rôle.

- En adoptant les pratiques DevOps, les équipes veillent à assurer la fiabilité, la haute disponibilité du système, et visent à éliminer les temps d'arrêt tout en renforçant la sécurité et la gouvernance.





Planifier (Plan): les équipes DevOps imaginent, définissent et décrivent les fonctionnalités des applications et des systèmes qu'elles créent:
Création de backlogs, suivi des bogues, gestion du développement logiciel agile avec Scrum, l'utilisation de tableaux Kanban et la visualisation des progrès réalisés grâce aux tableaux de bord.

Développement (Develop) : comprend tous les aspects du codage (écriture, test, révision et intégration du code par les membres de l'équipe), ainsi que la génération de ce code dans des artefacts de build pouvant être déployés dans divers environnements.

Fournir (Deliver): la livraison consiste à déployer des applications dans des environnements de production de manière cohérente et fiable.
La phase de livraison englobe également le déploiement et la configuration de l'infrastructure de base entièrement régie qui constitue ces environnements.

Opérer (Operate): la phase d'exploitation implique la maintenance, la supervision et le dépannage des applications dans les environnements de production.

3. Culture DevOps

DevOps repose essentiellement sur la culture de l'organisation, et les personnes qui en font partie :

– **Culture de collaboration** : Différentes équipes telles que le développement et les opérations informatiques doivent partager leurs processus, priorités et préoccupations DevOps.
  - Ces équipes doivent également planifier leur collaboration et s'aligner sur les objectifs et les indicateurs de réussite liés à l'entreprise.

– **Culture d'Évolutions en termes de portée et de responsabilité** :
  - Au fur et à mesure que les équipes s'alignent, elles s'approprient et s'impliquent dans d'autres phases du cycle de vie, sans se limiter aux seules phases inhérentes à leurs rôles.

– **Cycles de mise en production plus courts** :
  - Les équipes DevOps restent agiles en publiant des logiciels dans des cycles courts.
  - Les cycles de mise en production plus courts facilitent la planification et la gestion des risques.

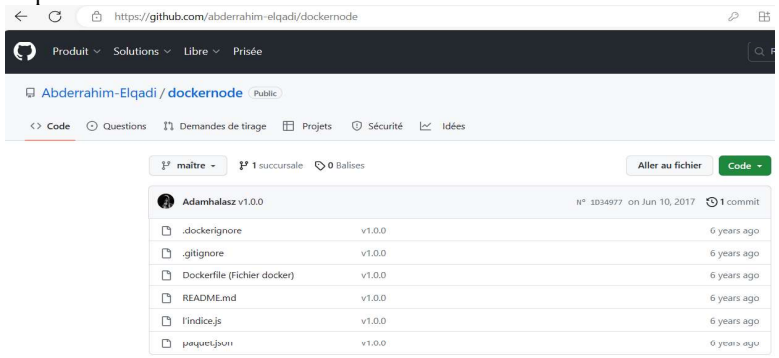
– **Apprentissage continu** :
  - Les équipes DevOps hautement performantes adoptent un état d'esprit de croissance.
  - Elles effectuent un Fail-fast et intègrent ce qu'elles apprennent dans leurs processus, s'améliorant continuellement, renforçant la satisfaction des clients, accélérant leur capacité à innover et à s'adapter au marché.

4. Outils de DevOps

Plein de langages de scripts utilisables (Python, ...), des technologies de développement (Maven, Gradle, ...) avec des librairies (libcloud, vmware, etc).

4.1 Git / GitHub

– **GitHub** est une plateforme de gestion et d'organisation de projets basée sur le cloud qui intègre les fonctions de contrôle de version de Git.
  - Tous les utilisateurs de GitHub peuvent suivre et gérer les modifications apportées au code source en temps réel tout en ayant accès à toutes les autres fonctions de Git disponibles au même endroit.



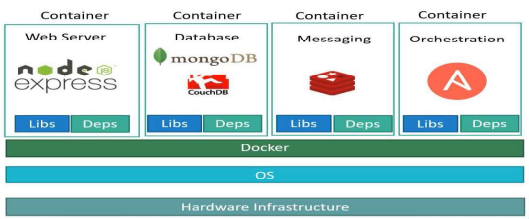
– **Git** est un projet open-source qui a été lancé en 2005.
  - Il s'agit d'un système de contrôle de version distribué.
  - Tout développeur de l'équipe ayant un accès autorisé peut gérer le code source et l'historique des modifications à l'aide des outils de ligne de commande Git.


```
ADmiN@DESKTOP-FPVVLBO MINGW64 ~ (master)
$ git log
commit 6558605b218b5237d573fc5291d7d86942b282f8 (HEAD -> master)
Author: demo <demo@gmail.com>
Date:   Sat Oct 14 09:25:55 2023 +0100

    Initial commit of design bd1

commit 212fa294a3f2de96b1f9e098493714a0f4a273cf (origin/master)
Author: demo <demo@gmail.com>
Date:   Wed Oct 11 11:09:25 2023 +0100

    premiere maj

ADmiN@DESKTOP-FPVVLBO MINGW64 ~ (master)
$
```

git init permet d'initialiser un dépôt local vide dans votre répertoire courant ;
git add permet d'indexer un fichier ou un répertoire
git commit permet de valider les modifications sur votre dépôt local ;
git remote add <nom> <url> : permet de lier votre dépôt local (initie par init) `a un dépôt distant (url sur GitHub) ;
git push permet d'envoyer les modifications indexées au dépôt distant.

4.2 Docker

– Docker est la société qui a justement lancé **la technologie des conteneurs** en 2013, et qui domine le marché aujourd'hui.
– Il s'agit d'une plateforme logicielle open source permettant **d'automatiser le déploiement** d'applications et **de leurs dépendances** dans différents **conteneurs** isolés sur un n'importe quel serveur Linux.

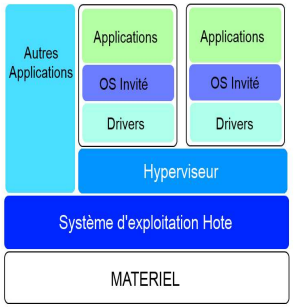


– Initialement créé pour fonctionner avec la plateforme Linux,
– Docker fonctionne désormais avec d'autres OS tels que Windows ou Apple macOS.
– Les services ou fonctions de l'application et ses différentes bibliothèques, fichiers de configuration, dépendances et autres composants sont regroupés au sein du container.
– Chaque container exécuté partage les services du système d'exploitation.

a. Les machines virtuelles (MV) et les containers

La virtualisation hardware consiste à faire fonctionner sur une même machine physique, plusieurs systèmes comme s'ils fonctionnaient sur des machines physiques distinctes.
Si on prend l'exemple d'un serveur, un serveur est composé de plusieurs "couches" :

- **Applications** (Ce sont les apps tel que gmail, word... qui vont "tourner" sur la machine.)
- **OS** (Operating System ou système d'exploitation : c'est le premier programme à s'exécuter lors du démarrage de la machine et il assure le lien entre les ressources matérielles (hardware) de la machine et les applications qui "tournent" dessus. Par exemple, on parle de MacOs, Windows, iOs...)
- **Hardware** (machine physique peut être un ordinateur...)



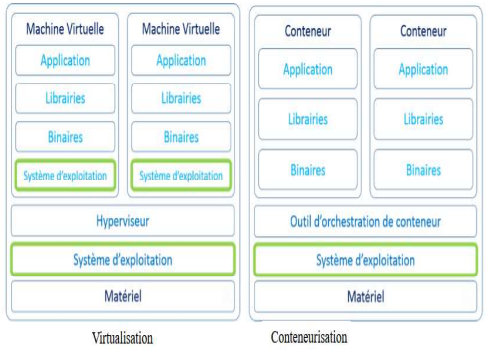
- **La virtualisation hardware** va donc permettre d'utiliser les ressources de la machine, soit le hardware, pour faire "tourner" plusieurs systèmes comme si chaque système tournait sur une machine séparée avec ses propres ressources matérielles, donc plusieurs OS par hardware sur lesquels tournent plusieurs applications.

- **Une machine virtuelle** c'est donc un ordinateur créé à l'intérieur d'un ordinateur, soit un OS installé à l'intérieur d'un OS.
- Une machine virtuelle va donc exécuter son propre OS qui va pouvoir accéder aux ressources de la machine physique qui l'héberge (CPU, mémoire RAM, etc).
- Il est possible d'exécuter plusieurs machines virtuelles simultanément sur un même ordinateur physique.

- Cependant, les hyperviseurs de machines virtuelles reposent sur une émulation du hardware, et **requièrent donc beaucoup de puissance de calcul**.
  Pour remédier à ce problème, de nombreuses architectures se tournent vers les containers, et par extension vers Docker.

– **Les containers** sont proches des machines virtuelles, mais présentent un avantage important.

- Alors que la virtualisation consiste à exécuter de nombreux systèmes d'exploitation sur un seul et même système, **les containers se partagent le même noyau de système d'exploitation** et isolent les processus de l'application du reste du système.



- Le conteneur fournit une virtualisation au niveau du système d'exploitation (virtualisation software) tandis que la machine virtuelle, quant à elle, fournit une virtualisation au niveau du matériel (virtualisation hardware).

– Il est donc nettement **plus efficient qu'un hyperviseur en termes de consommation des ressources système**. Concrètement, il est possible d'exécuter près de 4 à 6 fois plus d'instances d'applications avec un container qu'avec des machines virtuelles sur le même hardware.

– L'une des pratiques clés du développement de logiciel moderne **est d'isoler les applications** déployées sur un même hôte ou sur un même cluster. Ceci permet d'éviter qu'elles interfèrent.

– Pour exécuter les applications, il est toutefois nécessaire d'exploiter des packages, des bibliothèques et divers composants logiciels.
  - Pour exploiter ces ressources tout en isolant une application, on utilise depuis longtemps les machines virtuelles.
  - Celles-ci permettent de séparer les applications entre elles sur un même système, et de réduire les conflits entre les composants logiciels et la compétition pour les ressources. Cependant, une alternative a vu le jour : les conteneurs.

– Une machine virtuelle s'apparente à un système d'exploitation complet, d'une taille de plusieurs gigaoctets, permettant le partitionnement des ressources d'une infrastructure.
– Un conteneur délivre uniquement les ressources nécessaires à une application.
– Le conteneur **partage le kernel de son OS** avec d'autres conteneurs. C'est une différence avec une machine virtuelle, utilisant un hyperviseur pour distribuer les ressources hardware.
– Cette méthode permet de réduire l'empreinte des applications sur l'infrastructure. Le conteneur regroupe tous les composants système nécessaires à l'exécution du code, sans pour autant peser aussi lourd d'un OS complet.

– Un conteneur, c'est finalement un système d'exploitation minimaliste et un package logiciel qui contient :
  - Les services (applications, BDD, serveur web)
  - Les dépendances de ces services (librairie, code source, fichiers de configuration - ensemble des ressources externes nécessaires à l'exécution du code.)

  Tout en les isolant les uns des autres sur un même serveur physique.
  On nomme cet ensemble une image.

b. Docker : quelles sont les composants ?

– Docker est une technologie de **conteneurisation** qui facilite la gestion de dépendances au sein d'un projet et ce, à tous les niveaux (développement et déploiement).
  - Disponible sur Linux, Windows et Mac OS,
  - Le mécanisme de Docker se centre autour des **conteneurs** et de leur orchestration, et c'est en cela que la **conteneurisation** se différencie de la **virtualisation**.

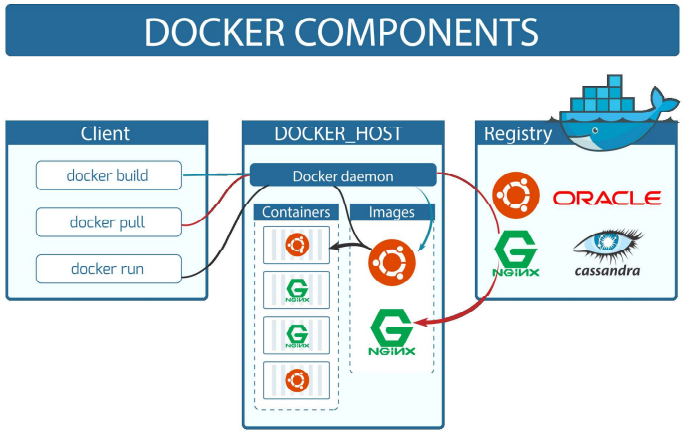
– Docker est un système d'exploitation pour conteneurs.

– Docker est installé sur chaque serveur et offre des commandes simples pour concevoir, démarrer ou arrêter des conteneurs.

Quelques exemples de commandes docker :

- Connaître la version de Docker installé : **docker -- version**
- Lancer un conteneur: **docker run -d -p 80:80 docker/getting-started**
- Lister tous les conteneurs : **docker ps -a**
- Obtenir toutes les images présentes sur le système : **docker images**
- Construire une image à partir d'un Dockerfile : **docker build https://github.com/docker/rootfs.git#container:docker**
- Supprimer un conteneur : **docker rm ID_du_conteneur**
- Supprimer une image : **docker rmi ID_de_l_image**
- Afficher les logs d'un conteneur avec leurs détails : **docker logs --details**

– La plateforme Docker repose sur **plusieurs technologies et composants** :



– **Docker Engine** est l'application à installer sur la machine hôte pour créer, exécuter et gérer des conteneurs Docker.
  - Il s'agit du moteur du système Docker qui regroupe et relie les différents composants entre eux.
  - C'est **la technologie client-serveur** permettant de créer et d'exécuter les conteneurs, et le terme Docker est souvent employé pour désigner Docker Engine.

– **Docker Daemon** traite les requêtes API afin de gérer les différents aspects de l'installation tels que les images, les conteneurs ou les volumes de stockage.

– **Docker Client** est la principale interface permettant de communiquer avec le système Docker.
  - Il reçoit les commandes par le biais de l'interface de ligne de commande et les transmet au Docker Daemon.

– **Les conteneurs Docker**
  - Un conteneur Docker ou Docker Container **est une instance d'image Docker** exécutée sur un microservice individuel ou un stack d'application complet.
  - En lançant un conteneur, on ajoute une couche inscriptible sur l'image. Ceci permet de stocker tous les changements apportés au conteneur durant le runtime.

– **Le registre Docker** est un système de catalogage permettant l'hébergement et le " push and pull " des images Docker.

– **Le Docker Hub** est le registre officiel de Docker.
  - Il s'agit d'un répertoire SaaS permettant de gérer et de partager les conteneurs.

– **Dockerfile**
  - Chaque conteneur Docker débute avec un " Dockerfile ". Il s'agit d'un **fichier texte** rédigé dans une syntaxe compréhensible, comportant les instructions de création d'une image Docker.
  - Un Dockerfile **précise le système d'exploitation** sur lequel sera basé le conteneur, et les langages, variables environnementales, emplacements de fichiers, ports réseaux et autres composants requis.

Les commandes de Dockerfile :

- **FROM** : c'est la commande qui sert à définir l'image de base de l'application ;
- **LABEL** : qui permet d'ajouter des métadonnées à l'image ;
- **RUN** : qui sert à exécuter des instructions dans le conteneur ;
- **ADD** : pour ajouter des fichiers dans l'image ;
- **WORKDIR** : afin de spécifier le répertoire dans lequel sera exécuté tout l'ensemble des instructions
- **EXPOSE** (facultatif) : qui sert à spécifier le port que l'on souhaite utiliser ;
- **VOLUME** (facultatif) : pour spécifier le répertoire à partager avec l'hôte ;
- **CMD** : qui sert à indiquer au conteneur la commande à exécuter lors de son lancement.

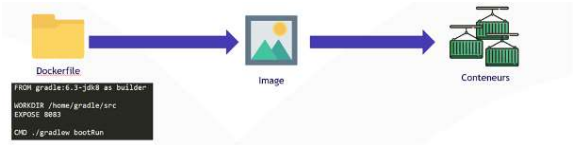
Exemple :

```
FROM node:6.10.3
# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install
# Bundle app source
COPY . /usr/src/app
EXPOSE 9000
CMD [ "npm", "start" ]
```

– **Image Docker** est un modèle en lecture seule, utiliser pour créer des conteneurs Docker.
  - Elle est composée de plusieurs couches empaquetant toutes les installations, dépendances, bibliothèques, processus et codes d'application nécessaires pour un environnement de conteneur pleinement opérationnel.

- Après avoir écrit le Dockerfile, on invoque **l'utilitaire " build "** pour créer une image basée sur ce fichier. Cette image se présente comme un fichier portable indiquant quels composants logiciels le conteneur exécutera et de quelle façon.

Une image vaut mille mots, voici donc une illustration qui montre les différentes étapes de création d'un conteneur :



Le **Dockerfile** permet de créer une image. Cette **image** contient la liste des instructions qu'un **conteneur** devra exécuter lorsqu'il sera créé à partir de cette même image.