1 Théorie

Le modèle Waterfall est plutôt prédictif et séquentiel, chaque phase doit être complétée avant que la suivante ne commence, par contre l'Agile se base sur la collaboration, l'adaptabilité, et les livraisons itératives (sprints), alors que le modèle DevOps se concentre sur la collaboration continue entre les équipes de développement (DEV) et d'exploitation (OPS) pour améliorer l'efficacité du cycle de développement logiciel.

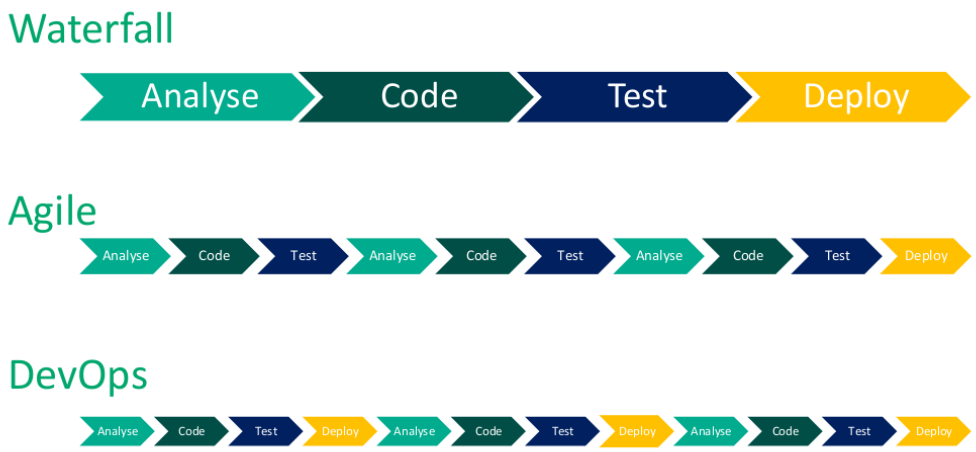

FIGURE 1 – Les modèles de chaîne de livraison

1.1 L'usine logicielle

L'usine logicielle c'est l'ensemble des technologies et outils misent à disposition des développeurs, des IOps, des architectes et qui sont nécessaires à la réussite du projet. Durant cette partie, il y aura une démonstration d'un enchaînement logique de technologies justifiant ainsi l'utilisation de chacune d'entre elles par rapport au contexte général de cette transformation numérique.

1.1.1 Les applications microservices

Tout a commencé par l'adoption de ce nouveau modèle d'application microservices. **Martin Fowler** [**?**], l'une des figures des architectures microservices en donne la définition suivante :

« C'est une approche de développement d'une seule application en tant que suite de **petits services**, chacun s'exécutant dans son propre processus et communiquant avec des mécanismes légers, souvent une API sur HTTP. Ces services sont construits autour de capacités métier et peuvent être **déployés indépendamment** par des chaînes de déploiement entièrement automatisées. Il existe un minimum de gestion centralisée de ces services, qui peuvent être écrits dans différents langages de programmation et utiliser différentes technologies de stockage de données »

Résilience globale	En cas d'indisponibilité d'un service, ce n'est pas tout le système qui tombe, mais seulement quelques fonctionnalités qui sont indisponibles.
Favorisation de l'innovation	Chaque service pouvant utiliser sa propre technologie afin d'adopter l'outil le mieux adapté à un besoin métier en particulier. Que ce soit au niveau du langage utilisé ou même des types de base de données.
Scalabilité	L'autonomie des services permet d'adopter des technologies de scalabilité adaptées au profile technique du service afin d'optimiser les coûts en réduisant les ressources durant les heures creuses.
Fiabilité et accélération des déploiement	La fragmentation des déploiements permet de gérer plus facilement les évolutions technologiques du service. La gestion par de petites équipes améliore la productivité.
Réutilisabilité des services	La mise à disposition des services à travers des interfaces normalisées permet de factoriser certains services dont les informations sont utilisées par d'autres services.

1.1.2 Conteneurisation Vs Virtualisation

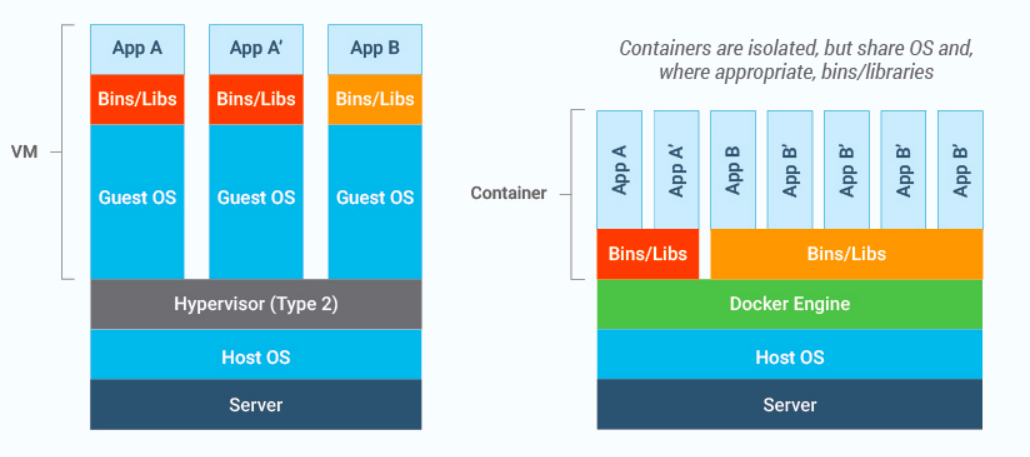

FIGURE 2 – Conteneurisation Vs Virtualisation

La virtualisation [**?**] consiste à créer des machines virtuelles à partir d'un seul ordinateur physique dont chacune exécute son propre système d'exploitation en total isolation les unes des autres. L'hyperviseur c'est la couche de virtualisation qui gère l'allocation des ressources physiques aux machines virtuelles.

La conteneurisation consiste à créer une unité standardisée et isolée appelée conteneur pour une application contenant tous ses dépendances. Les conteneurs en différence par rapport au machine virtuelle sont légers et partagent le noyau du système d'exploitation de l'hôte, ce qui les rend très portables et plus efficaces par contre très vulnérables si le l'hôte est attaqué.

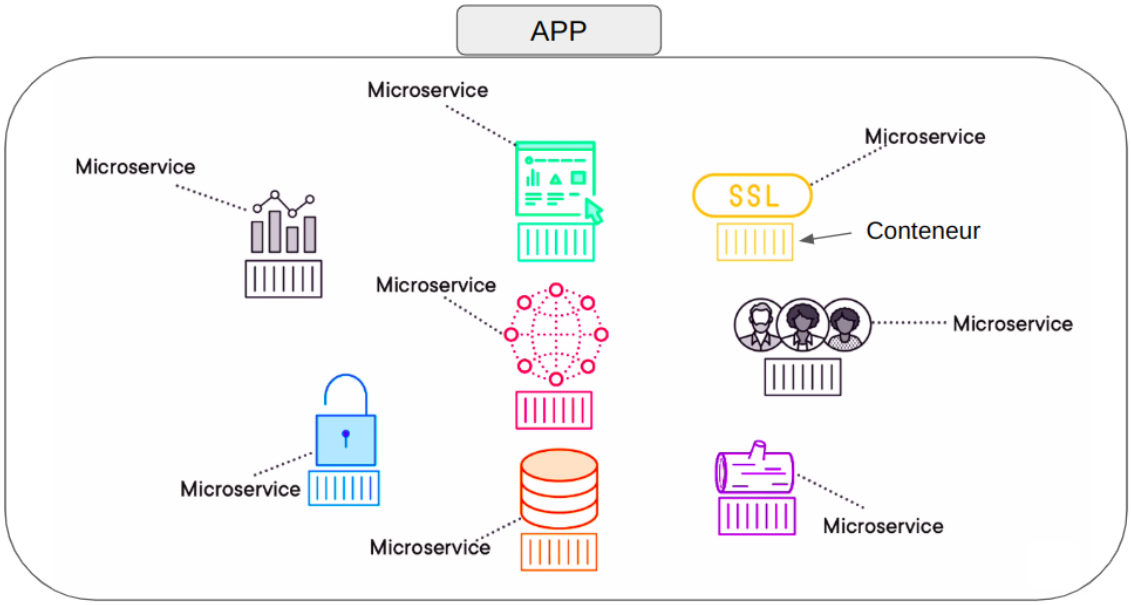

FIGURE 3 – Conteneurisation et microservices

Naturellement, et après l'essor des applications microservices, la conteneurisation des applications ou plutôt des microservices d'une application devient de plus en plus utilisée et accélère cette transition d'une architecture monolithique centralisée vers une architecture microservices décentralisée vu que les services qu'offre la conteneurisation répondent parfaitement à l'utilité de ce nouveau modèle que ce soit au niveau de la résilience, la scalabilité, la fiabilité ou l'interopérabilité.

1.1.3 Le Cloud computing

Le Cloud computing est basé sur le principe que la majorité des opérations réalisées s'effectuent dans des machines à distances. Les données collectées au cours de ce processus sont traitées et stockées dans des serveurs Cloud. Du coup la machine locale ayant accès au Cloud n'a pas besoin d'avoir un matériel avec une grande performance mais jouera le rôle d'administrateur et développeur de ses ressources sur le Cloud.

Dans un environnement de Cloud hybride, un conteneur est l'unité de base d'exécution des opérations des microservices. Le fournisseur est donc responsable de l'endroit où le conteneur tourne selon la configuration souhaitée et les régions disponibles des serveurs du Cloud. Les conteneurs facilitent alors le déploiement de ces microservices dans un environnement de Cloud hybride grâce à la platforme DevOps.

1.1.4 Les modèles de déploiements

D'après RedHat [**?**] : " L'expression « aaS » ou « as-a-Service » signifie généralement qu'un tiers se charge de vous fournir un service de cloud computing, afin que vous puissiez vous concentrer sur des aspects plus importants, tels que votre code et les relations avec vos clients. Chaque type de cloud computing allège la gestion de votre infrastructure sur site. "

Avant de parcourir les modèles de déploiements adoptés, les types de Cloud présentés avant exigent l'existence d'un partage de responsabilité entre le fournisseur du Cloud et l'entreprise

cliente

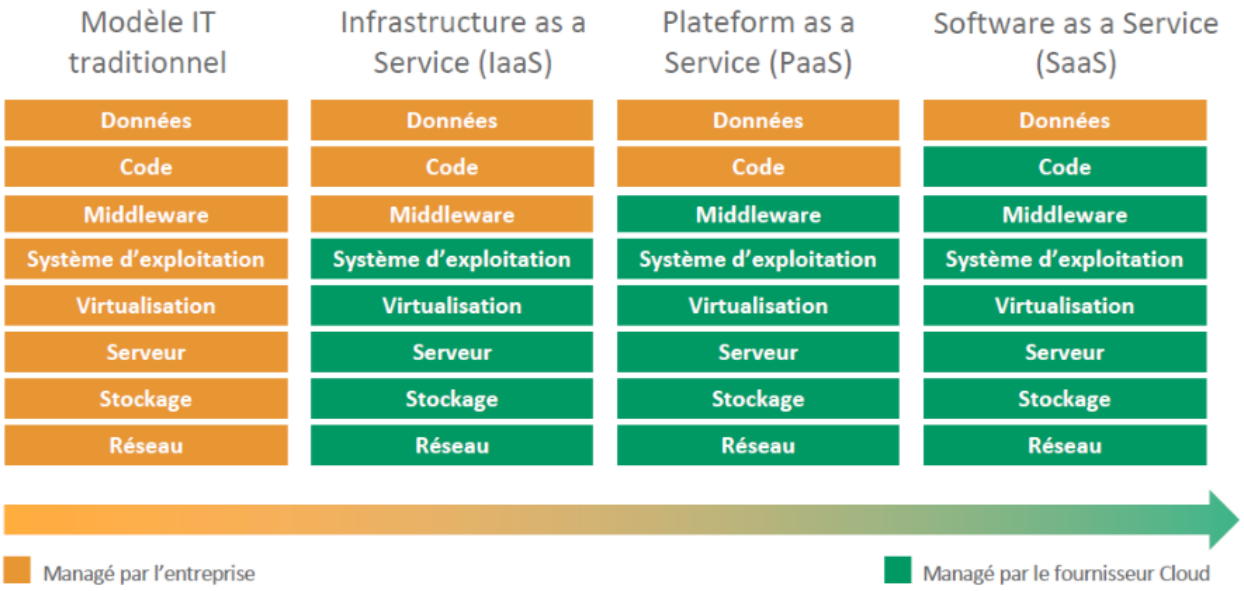

FIGURE 4 – Les types de services sur le Cloud

- **Déploiement en mode IaaS** : Ce type de déploiement se déroule des VMs sont des machines Linux RedHat et les applications sont lancées en mode services avec systemctl. Dans ce cas particulier le fournisseur du Cloud est lui même le client puisque c'est un Cloud privé.

- **Déploiement en mode PaaS** : les applications sont lancées sur des conteneurs Docker sur un cluster Kubernetes (KaaS). Dans ce cas le fournisseur Cloud c'est cloud provider et s'en occupe de toute les mises à jour logicielles et de la maintenance de l'infrastructure tandis que l'entreprise est responsable que du code et des données.

1.1.5 BUILD et RUN : les pratiques DevOps

Lorsqu'on parle du DevOps, on ne parle pas d'une technologie mais plutôt d'une culture adoptée. L'origine de ces pratiques DevOps viennent d'un constat d'échec de collaboration entre les développeurs et les ingénieurs de production.

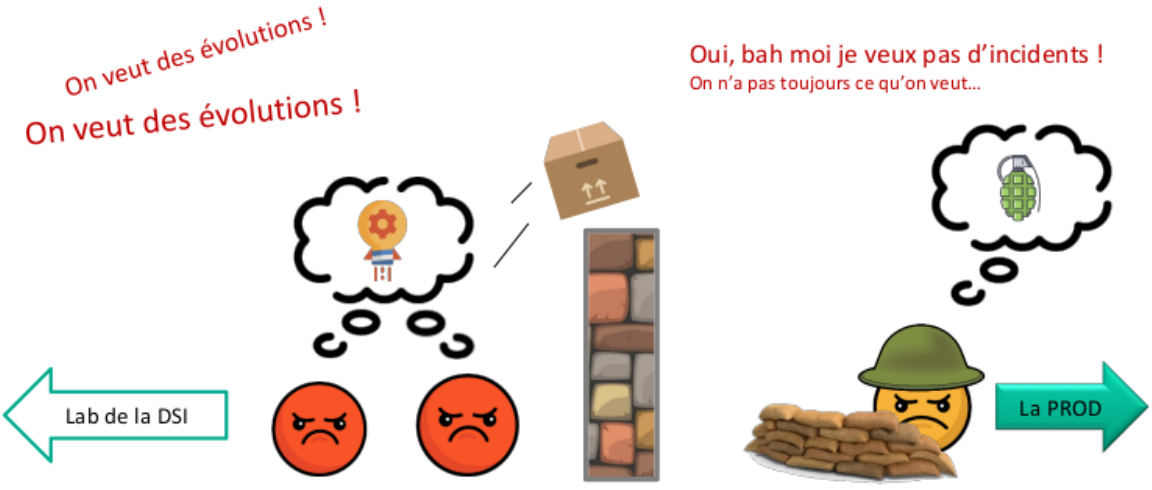

FIGURE 5 – DEV vs OPS

Du coup afin de garantir une saine transformation numérique une montée en fréquence est nécessaire sur la chaîne de livraison :

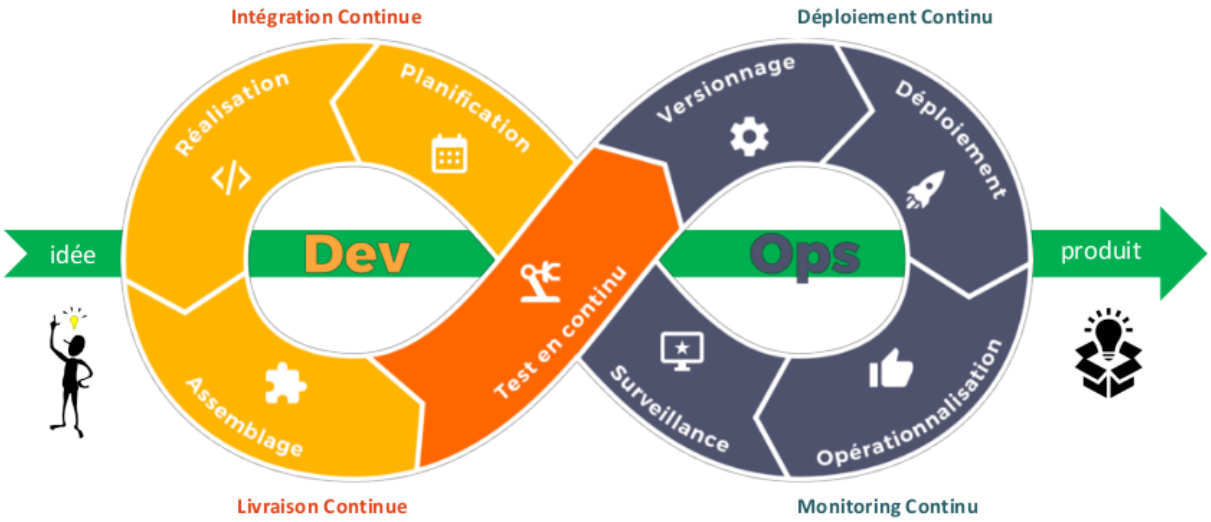

FIGURE 6 – Chaîne de livraison DevOps

la culture DevOps [**?**] nécessite de la collaboration, la transparence, la confiance et de l'empathie afin que cette chaîne continue de livraison peut fonctionner correctement et transformer une idée abstraite à un produit concret. Cette chaîne se compose de 4 principaux étapes :

- **L'intégration continue** (CI) "**BUILD**" : L'objectif de cette pratique c'est de détecter, à travers une série d'actions automatisées, les erreurs du nouveau code intégré par le développeur ainsi que l'analyse de sa qualité et les vulnérabilités afin d'assurer d'avoir un code fonctionnel.

- **La livraison continue** (CD) "**BUILD**" : Pendant la livraison continue le développeur va réaliser un build automatisé pour tester sur chaque environnement cible (DEV, INT, QUAL, PROD) [1] le bon fonctionnement de son application en générant deux types de packages adaptés à la Toolchain CI/CD utilisé : un binaire généralement un dossier compressé de type zip dans le cas du mode IaaS ou bien une image Docker dans le mode PaaS qui sera lancée pour créer le conteneur.

- **Déploiement continu** (CD) "**BUILD**" : Cette étape géré par les IOps vise à automatiser entièrement le déploiement des packages livrés par les développeurs sur l'environnement souhaité avec les paramètres associés selon le mode désiré (IaaS ou PaaS) pour objective de vérifier avant la mise en production que l'application est fonctionnelle sur les autres environnements de tests.

- **Surveillance continue** (CM) "**RUN**" : L'IOps s'occupe de cette pratique qui consiste à analyser et collecter les données de performance de l'application ou de l'infrastructure grâce aux outils de monitorings sur les différents environnements de déploiement. Cela permet de surveiller la disponibilité continues des applications.

1. Plus d'informations sur la partie implémentation : l'approche d'ingénierie

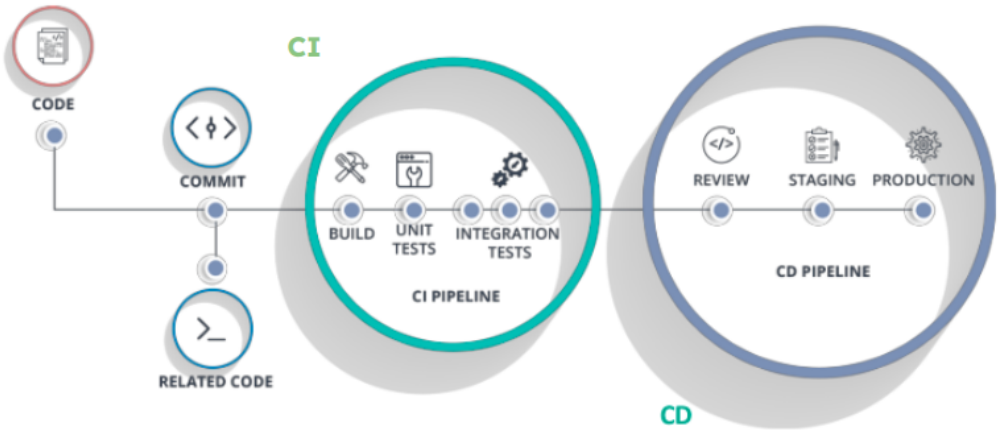

FIGURE 7 – CI/CD pipeline

1.1.6 La Stack technique

Dans cette partie les différents outils, par fonctionnalité, utilisés dans le cadre de notre périmètre seront présentés, par contre leur utilisation technique dans le contexte sera plus détaillée dans la section implémentation :

- **Gitlab "Gestion du code source"** :

Une plateforme de développement collaborative , basée sur les fonctionnalités du logiciel GIT. Elle permet de centraliser le code à travers des dépôts , et de gérer les versions du code source, avec un système de branches ainsi la gestion des conflits lors du fusionnement des différentes branches.

Afin d'éviter les conflits dans le code, il est préférable de suivre les bonnes pratiques suivantes de modification du code :

- Le process de modification du code commence par la création d'une nouvelle branche nommé Develop qui est l'image d'une branche mère (généralement la branche master qui est crée par défaut), son rôle c'est valider les tests après les modifications afin de synchroniser avec le Master que les nouvelles versions du code fonctionnel.

- Si le besoin est d'ajouter une nouvelle fonctionnalité une sous-branche est créée à partir de Develop et son nom peut commencer par "Feature", si c'est un bug à corriger le nom commence par "fix", le nommage est définit selon les conventions entre l'équipe.

- Lorsque les modifications sont faites et les tests sont validés, un "merge request" est crée par l'auteur de la branche pour synchroniser sa branche avec la branche principale et selon les droits il peut valider lui même ou bien quelqu'un d'autres le merge.

- Si plusieurs personnes travaillent sur une seule branche avant chaque modification, lancer un **"git pull"** pour avoir les dernières modifications sur la branche en remote sur le code local et après chaque modification lancer respectivement **"git add fichier modifié"** - **"git commit"** - **"git push"**

pour synchroniser le code local avec le code en remote sur la même branche.

- **Jfrog Artifactory "Gestion des binaires"** :

Selon Jfrog [**?**] :"il s'agit d'un gestionnaire Universel qui gèrent les artefacts, les packages et les builds d'un emplacement centralisé et fait office de source unique de vérité pour tous les packages, les images de conteneurs et les charts Helm lorsqu'ils traversent l'ensemble du pipeline DevOps. Cela réduit le temps passé et le risque lié au téléchargement de dépendances à partir de dépôts publics. La gestion universelle d'artefacts évite les incohérences en permettant aux équipes de développement de trouver facilement la bonne version d'un artefact."

- **Jenkins "Orchestration de la CI/CD"** :

C'est un outil initialement développé pour faire de l'intégration continue, il permet de gérer le cycle de vie des releases d'une application orchestrer leur déploiement et de scanner chaque branche disponible sur le dépôt Git associé et instancier une pipeline par branche. La présence d'un Jenkinsfile écrit en Groovy[2] est indispensable pour qu'une branche Git soit reconnu par Jenkins et pour aussi définir les différentes étapes du déploiement nommé "Stages".

2. le nom d'un langage de programmation orienté objet destiné à la plate-forme Java

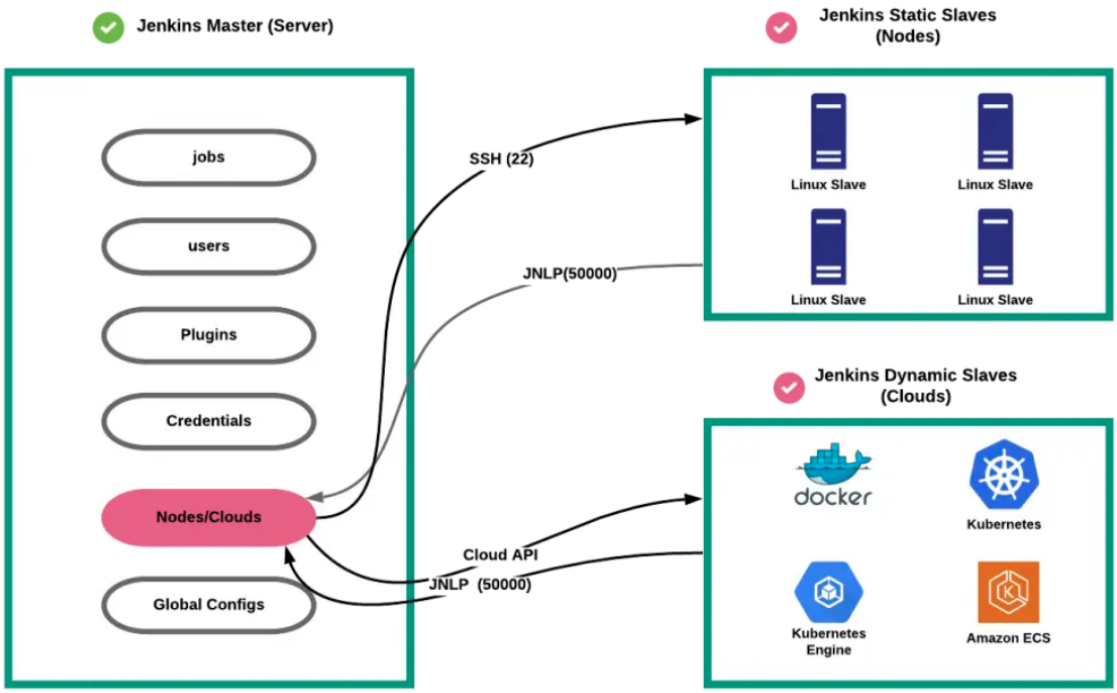

FIGURE 8 – Architecture de Jenkins

Le fonctionnement de Jenkins se base sur une architecture de type Master-Slaves [**?**] :

- Le nœud Master de Jenkins contient toutes les configurations clés (mots de passe, paramètres, authentification, etc), c'est le serveur de contrôle qui orchestre tout le workflow défini dans les pipelines. Par exemple, planifier un Job, surveiller Jobs, etc.

- Les Jobs de Jenkins c'est simplement les différentes tâches qui s'exécutent séquentiellement tel que défini par l'utilisateur sur les agents Jenkins.

- Il existe des cas d'usage dans lesquels l'agent doit se connecter à un compte cloud, à l'artifactory, ou à une API à l'aide des secrets. Dans Jenkins, différents types de secrets peuvent être enregistrer en tant qu'une information d'identification chiffrés avec AES [3] de type : Texte secret, Identifiant Mot de passe, etc.

- Les noeuds agents de Jenkins lancent les jobs ordonnés par le Master. il existe deux type d'agents :

> L'agent statique : il s'agit des serveurs (Windows/Linux) qui seront opérationnels tout le temps et resteront connectés au serveur Jenkins.

> L'agent dynamique : cet agent Jenkins est lancé sur le Cloud. Cela signifie que chaque fois le Master déclenche un job, un agent est déployé en tant que VM/conteneur à la demande et sera supprimé une fois le job terminé. Cette méthode permet d'économiser de l'argent en termes de coût d'infrastructure lorsque plusieurs builds continus sont nécessaires.

- **Ansible Tower "Automatisation du déploiement en IaaS"** :

3. Algorithme de chiffrement

C'est une plateforme Ansible permettant d'automatiser les tâches techniques (Déploiements, sauvegardes, reboot ...) sur des serveurs (VMs) via SSH, éliminant ainsi le besoin d'un agent dédié.

Les éléments principaux du fonctionnement sont :

> Projet : le projet associé au dépôt Gitlab qui porte les sources de déploiement

> Inventaire : La liste des machines où lancer les scripts.

> Playbook : Le script de déploiement.

> Rôles : les différents tâches Ansibles à exécuter dans le cadre du playbook.

> Job Template : le job qui associe le playbook à l'inventaire.

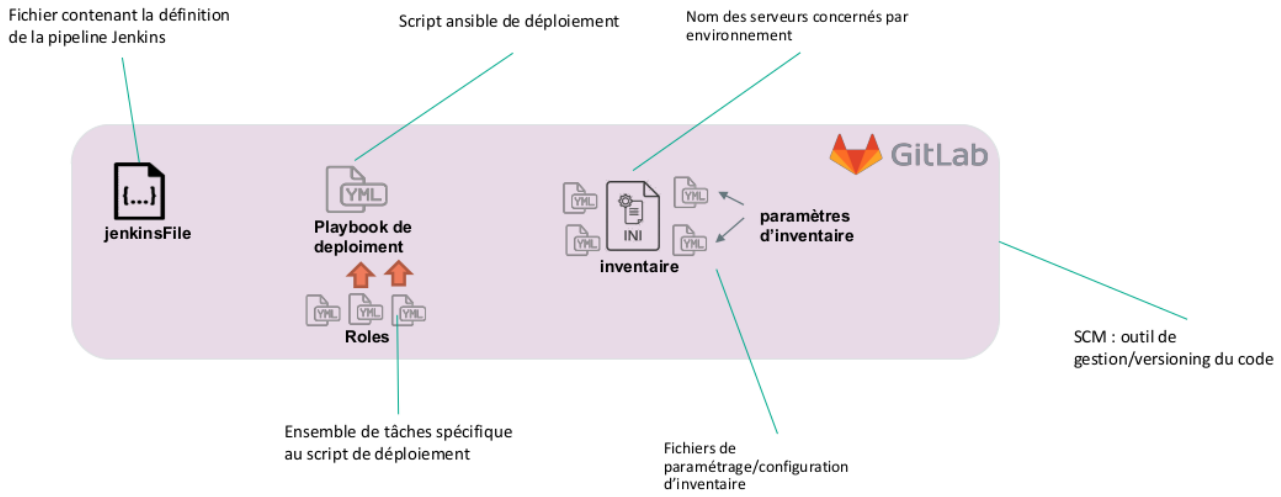

FIGURE 9 – Fonctionnement d'Ansible Tower avec Jenkins et Gitlab

- **Docker "Automatisation du Build"** :

Selon la définition d'IBM [**?**] : "Docker est une plate-forme open source qui permet aux développeurs de créer, déployer, exécuter, mettre à jour et gérer des conteneurs, des composants standardisés et exécutables qui combinent le code source de l'application avec les bibliothèques du système d'exploitation (OS) et les dépendances nécessaires pour exécuter ce code dans n'importe quel environnement."

Deux notions importantes à comprendre sur Docker :

- Image Docker : c'est un modèle immuable en lecture seule qui définit sur un Dockerfile les caractéristiques du conteneurs souhaités (OS,librairies, dépendances,script de lancement...) l'action de création d'une image c'est le Build d'où l'automatisation du Build.

- Conteneur Docker : c'est une instance d'exécution d'une image Docker.

- **Kubernetes "Automatisation du déploiement en PaaS"** :

Kubernetes ou K8s [4] est le principal outil de gestion d'applications conteneurisées à grande échelle, appelées aussi les applications microservices Cloud natives en adoptant une architecture de type Master-Slaves sur un cluster Kubernetes. Pour mieux comprendre son fonctionnement global une analogie avec une équipe de football est le meilleur exemple :

Imaginons une équipe de football sans coach, il y aura un désordre fatal et l'équipe ne peut pas se gérer, c'est là où le rôle d'un coach est primordial pour organiser l'équipe, placer chaque joueurs dans sa position, prendre les décisions de changer un joueur blessé pour garantir un bon état de l'équipe. Dans notre cas le coach c'est le noeud Master de Kubernetes et les joueurs c'est les noeuds Workers Kubernetes où les conteneurs Docker sont lancés. Le Master alors assure le bon déploiement des ces conteneurs ainsi que la gestion des flux entre eux, et garantie aussi la disponibilité de ces conteneurs selon le besoin désiré. En gros, la force de cette technologie c'est qu'il suffit de lui fournir une image Docker et Kubernetes s'occupe du reste.

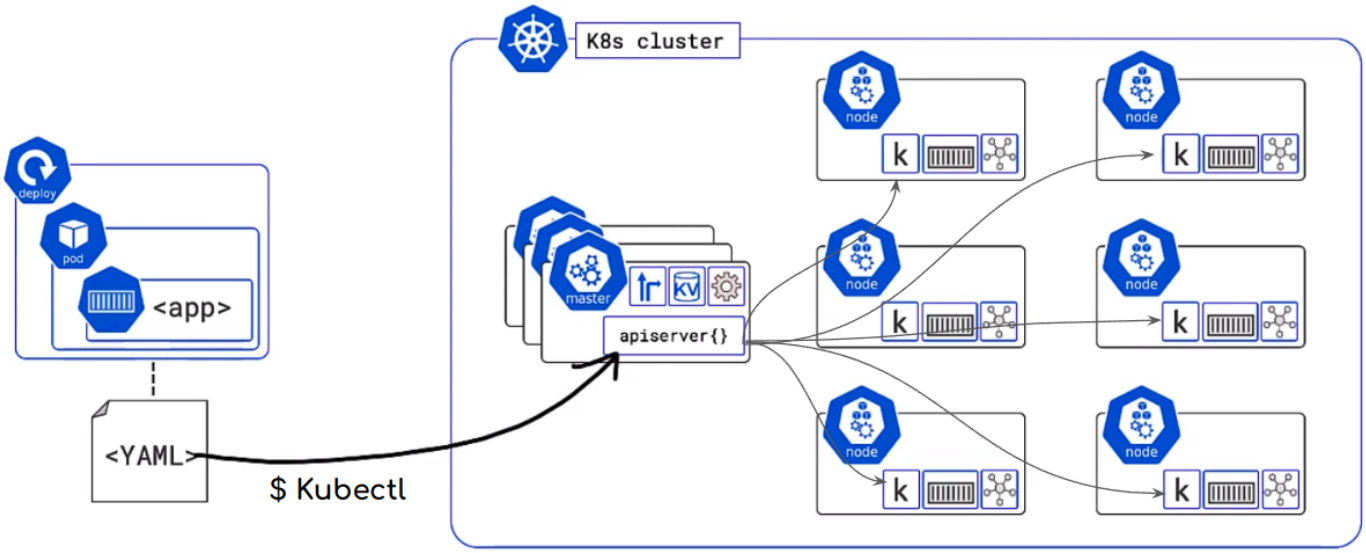

FIGURE 10 – L'architecture globale de Kubernetes

Pour mieux comprendre cette magie derrière le fonctionnement de Kubernetes, il est nécessaire de détailler la fonctionnalité de chaque composant de cette architecture [**?**].

> **K8s cluster** :

Un cluster en général dans le contexte de l'IT c'est un groupement logique de ressources flexibles qui agissent comme un seul système afin d'assurer la disponibilité et la redondance, le K8s cluster c'est simplement l'ensemble des noeuds Kubernetes (Master et Worker).

Au sein d'un cluster existent des clusters virtuels nommé **namespaces** qui joue le rôle de répartitions des ressources d'un cluster entre plusieurs environnements.

4. 8 c'est le nombre des lettres entre K et s dans Kubernetes

> **Noeud Master** :

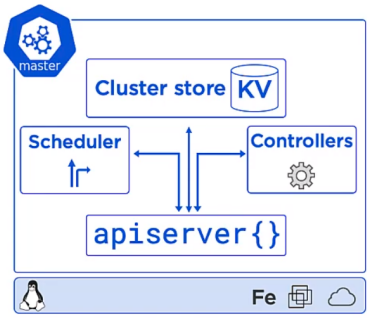

FIGURE 11 – Architecture du Master

Le noeud Master de Kubernetes c'est le responsable des opérations de contrôle et des planifications du cluster K8s grâce à ses quatre composants :

- API server : C'est le composant du Master avec lequel tous les autres composants soit internes ou externes au Master utilisent pour communiquer entre eux, il expose une API RESTful [5] sur un port sécurisé et consomme du JSON et du YAML.

- Cluster store KV (Key-Value) : c'est là que la configuration et l'état du cluster lui-même, ainsi que toutes les applications, sont stockés. Il est basé sur la base de données NoSQL distribuée etcd. Il est super critique pour les opérations de cluster et dans les grands clusters volumineux, il sera probablement la première chose qui sera mise sous pression.

- Controllers : c'est un ensemble de contrôleurs avec chacun son rôle spécifique comme le contrôleur des noeuds, des déploiements,du réseaux..., Il y a à peu près un contrôleur de tout ce qui se trouve dans le cluster.

- Scheduler : Il surveille toujours l'état du cluster afin de prendre des décisions selon la configuration souhaitée d'affecter ou de retirer des Pods [6] aux nœuds appropriés.

Pour que Kubernetes fonctionne correctement et surtout en production, il est nécessaire d'avoir une configuration **multi-master**, autrement dit une duplication des noeuds Master pour éviter l'indisponibilité. Du coup, la question qui se pose, qui aura la décision finale pour agir sur le cluster ? C'est grâce à une affection [7] d'un leader parmi le groupe des Masters qui va lui permettre d'avoir tous les droits pour les modifications alors que les autres sont en mode lecture seul. Au cas où le leader n'est plus disponible un autre leader est affecté.

> **Noeud Worker** :

5. une API conforme aux contraintes du architectural REST et permettant une interaction avec les services Web RESTful
6. contient un ou plusieurs conteneurs
7. L'affectation dépend du nombre des Master sur un même réseau par rapport aux autres

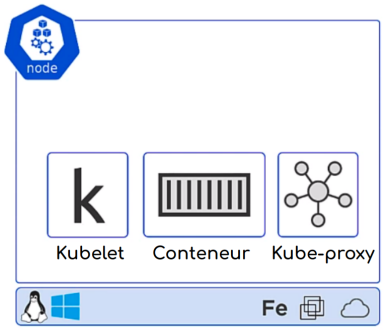

FIGURE 12 – Architecture du Worker

C'est l'équivalent d'un sous-réseau au sein du cluster où l'application sera lancée et se compose de trois éléments :

- Kubelet : c'est l'agent Kubernetes principal qui reçoit les commandes depuis l'Api Server du Master, lance les Pods et envoie des rapports au Master.

- Conteneur : contient l'application conteneurisé généralement un conteneur Docker.

- Kube-proxy : C'est le responsable de la partie réseau du Worker, il garantie que chaque Pod contient une unique adresse IP et joue le rôle d'un load balancer [8] du trafic.

> **Pods Kubernetes** :

Si Kubernetes par définition gère le lancement des conteneurs, ces derniers doivent obligatoirement être lancé sur ce qu'on appelle des Pods. C'est l'unité atomique de déploiement et de scalabilité sur Kubernetes qui représente un environnement partagé d'exécution dans le cas d'un Pod multi-conteneur.

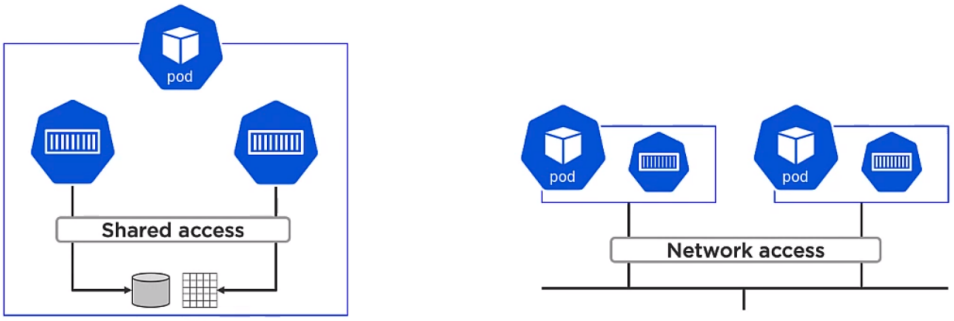

FIGURE 13 – Type de Pods

> **Services Kubernetes** :

C'est un objet Kubernetes, comme les Pods, utilisée pour désigner la solution qui permettre d'avoir une adresse stable pour atteindre les Pods afin de résoudre le problème de changement d'adresse IP des Pods, car ils sont susceptibles de tomber en panne ou d'être relancés. le Kube-proxy fait partie du Service qui va s'occuper du load balancing.

La question qui se pose cette fois c'est comment le Service route le trafic vers les Pods ? C'est grâce à une solution simple et puissante appelée **Label** qui est une sorte de tag pour dire que si le

8. répartition du trafic selon des règles définies

tag du Services Kubernetes et celui du Pod correspondent alors le trafic est routé vers ce dernier. L'exemple suivant illustre le cas d'utilisation du Service K8s pour diriger le trafic seulement vers les Pods ayant comme label : prod,be,1.4 :

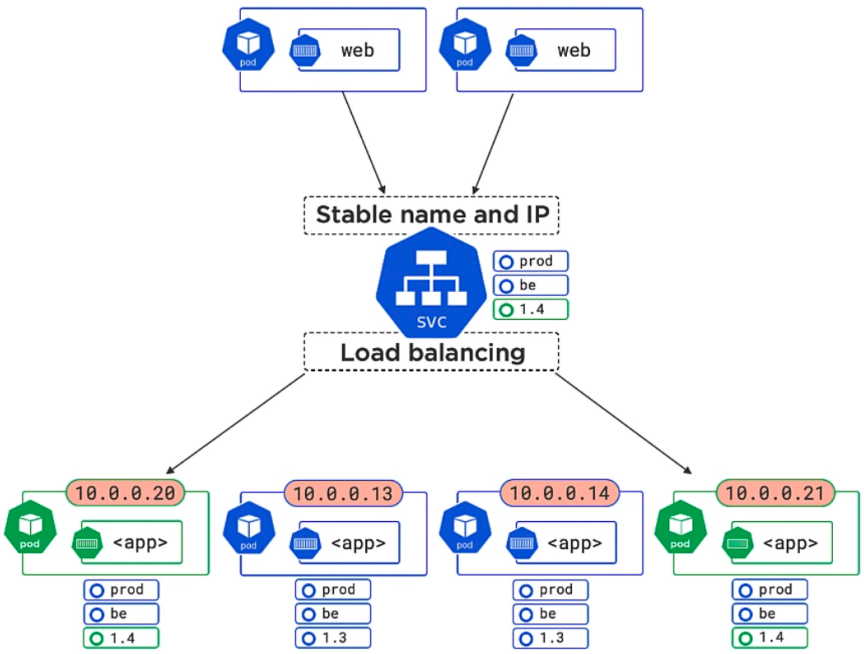

FIGURE 14 – Services Kubernetes

> **Deploy Kubernetes** :

l'objet Kubernetes Deploy est intéressant en particulier car en plus de gérer la création des Pods et leurs duplications pour l'auto-correction, il permet de réaliser des retours en arrière et des mises à jours sur les Pods. C'est l'unité standard de déploiement d'un conteneur sur Kubernetes.

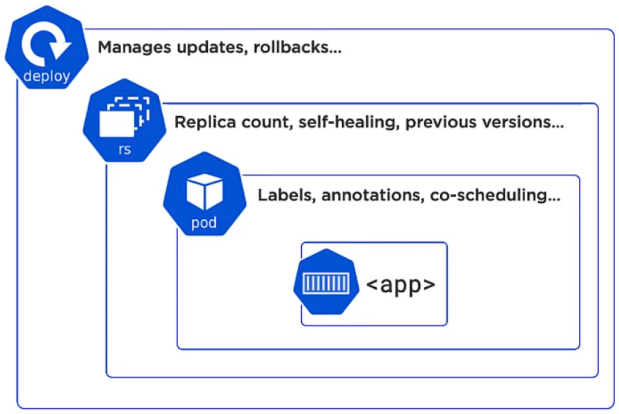

FIGURE 15 – Deploy Kubernetes

> **Kubectl CLI** [9] :

Après avoir expliqué l'essentiel concernant les composants du cluster K8s, cette partie s'intéresse en détail sur l'utilisateur.

Pour que Kubernetes fonctionne correctement l'utilisateur doit demander au Master la création de plusieurs objets Kubernetes comme des Workers, des Pods, Service, Deploy...

9. Interface de ligne de commande

Pour établir cet échange l'utilisateur utilise **Kubectl**, comme illustré sur la figure 10 qui est une ligne de commande de Kubernetes qui prend comme paramètre des fichiers YAML décrivant un état désiré sur le cluster K8s et pointe vers l'API Server du Master API qui authentifie ces fichiers, les autorise et les valide, puis il demande aux autres fonctionnalités du Master de les déployer et de les gérer. Voici un exemple simple d'un fichier YAML pour l'objet Deploy :

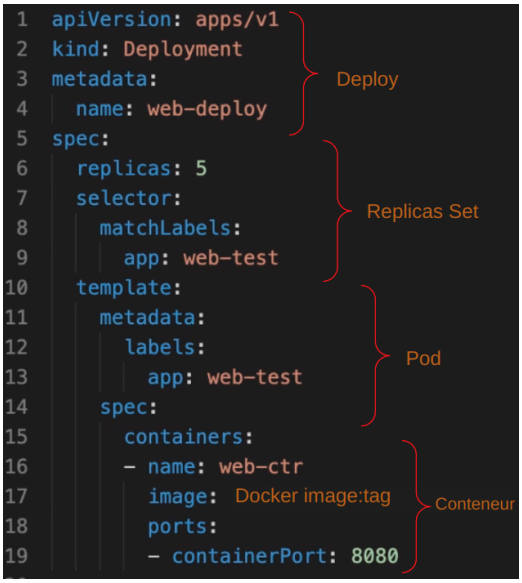

FIGURE 16 – Fichier YAML de déploiement

- **Helm "Automatisation du déploiement en PaaS"** :

D'après CircleCi [**?**] :" Helm c'est un outil qui automatise la création, le packaging, la configuration et le déploiement des applications Kubernetes en combinant vos fichiers YAML de configuration dans un seul package réutilisable ".

Au lieu de lancer à chaque fois des commandes **Kubectl** pour déployer chaque fichier YAML et pour rendre ces fichiers paramétrables selon l'application déployée, Helm automatise tout ce processus avec une seule commande magique : **Helm install nom_package**

Pour cela il suffit de regrouper les fichiers de configuration dans un seul dossier nommé dans le contexte Helm **une Chart** et c'est le nom de ce dossier qui sera utilisé dans la commande Helm pour déployer comme le montre l'exemple d'un package nommé chart-demo suivant :

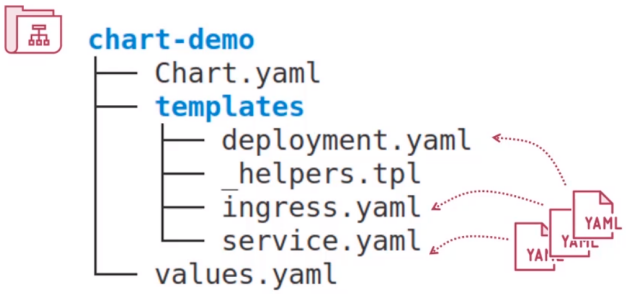

FIGURE 17 – Structure d'une Chart Helm

- Chart.yaml contient le nom de la Chart, la version et d'autres métadonnées [10].

- templates contient les fichiers de configurations de Kubernetes.

- values.yaml contient des variables utilisées dans les fichiers Kubernetes en utilisant une insersion dans le fichier cible, par exemple : name : {{ .Values.MyName }}

- helpers.tpl contient des templates utilisées dans les fichiers Kubernetes pour faciliter les modifications du code.

- **HashiCorp Vault "Gestion des secrets"** :

D'après le site officiel [**?**] : "HashiCorp Vault est un système de gestion des secrets et du chiffrement basé sur l'identité. Un secret est tout ce dont vous souhaitez contrôler étroitement l'accès, comme les clés de chiffrement API, les mots de passe et les certificats. Vault fournit des services de chiffrement protégés par des méthodes d'authentification et d'autorisation. À l'aide de l'interface utilisateur, de la CLI ou de l'API HTTP de Vault, l'accès aux secrets et autres données sensibles peut être stocké et géré en toute sécurité, étroitement contrôlé (restreint) et auditable."

- **Confluence "gestion des connaissances"** :

Confluence est un espace de travail d'équipe où les connaissances utilisées et la documentation dans les projets sont partagées. Les pages dynamiques offrent à l'équipe un endroit pour créer, capturer et collaborer sur n'importe quel projet ou idée.

- **Kanboard "gestion des tâches"** :

10. données sur une donnée

Kanboard est un gestionnaire de tâches visuel qui aide les collaborateurs sur un projet de gérer l'affectation des tâches et faire un suivi continue et global sur l'activité de l'équipe.

- **Retour sur les technologies** :

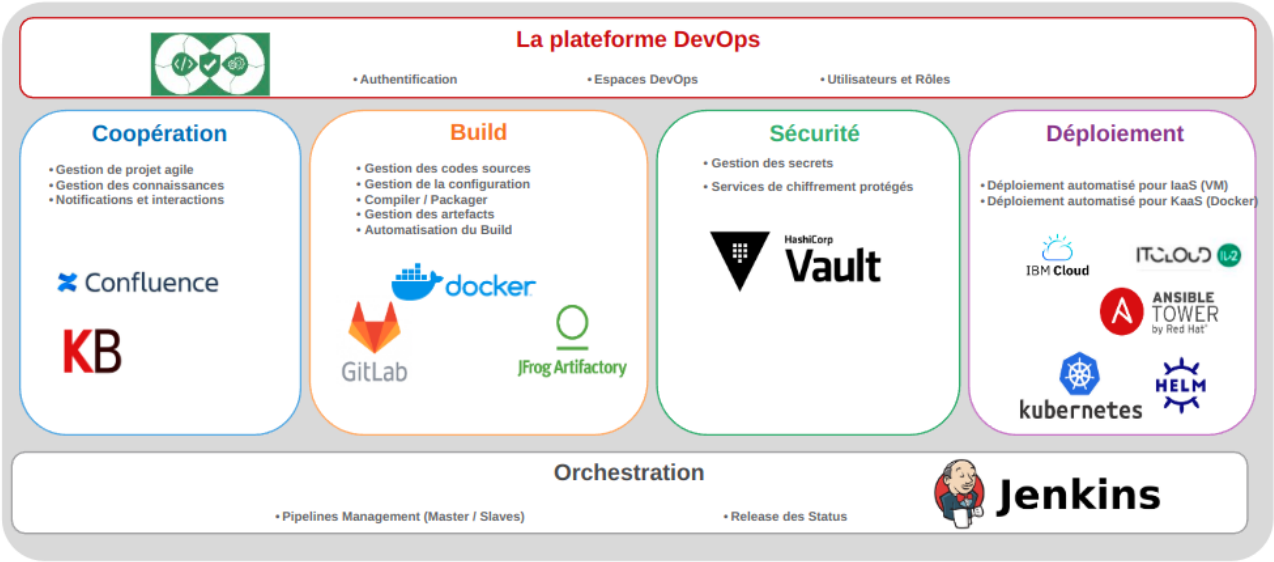

FIGURE 18 – La Stack technique

2 Implémentation

2.1 L'approche d'ingénierie

- **Le workflow Gitlab** : il s'agit d'organiser le travail de toute l'équipe et des robots autour du code source. Ceci va conditionner et modeler les pipelines d'intégration et de déploiement.

- **Les scénarios des pipelines** : Quels types de pipelines et quelles actions exécuter dans chaque pipeline. Déterminer les scénarios de déclenchements basés sur les événements et la logique enchaînement.

- **Le cycle de vie des artefacts et des releases** : penser à l'organisation des binaires en termes de construction, de versioning et de déploiement. Maîtriser le lien entre le binaire et la localisation du code source. La politique de release management : qu'est-ce qui fait d'un livrable une release? Quels niveaux d'exigences en termes de qualité, de sécurité et de fonctionnalités.

- **Automatisation et optimisation** : Avoir le réflexe d'automatiser le maximum les différentes étapes ainsi les tâches de sauvegarde et du redémarrage des serveurs ou des Pods tout en garantissant une optimisation du code, de la configuration et de la consommation des ressources.

un résumé du processus globale du développement d'un projet est présenté sur le schéma suivant :

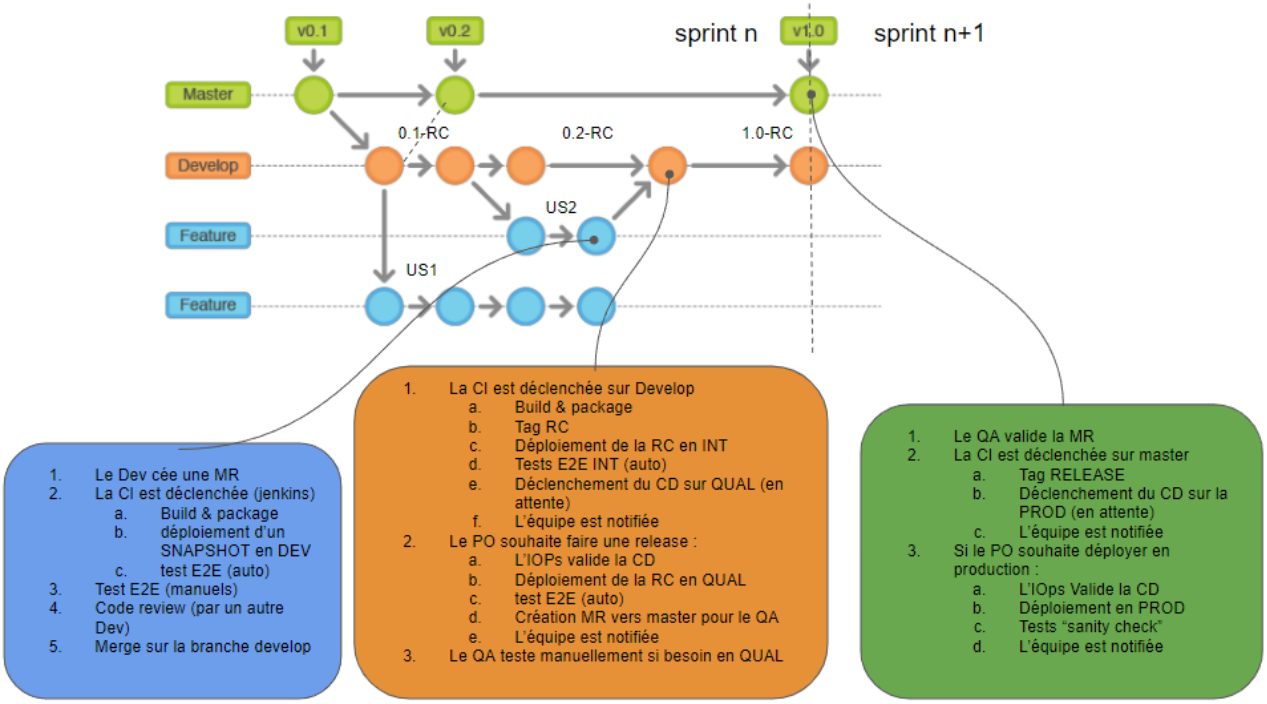

FIGURE 19 – Processus de l'approche d'ingénierie

Notant que :

Environnement	**Propriétaire**	**Utilité**
DEV	Développeurs	Permet au développeur de tester son US
INT	Développeurs	Intégrer et tester un ensemble de US
QUAL(REC)	PO/QA	Qualifier une Release
PROD	PO/Utilisateurs	Utiliser une Realease

Artefacts	Origine	Environnement de déploiement
SNAPSHOT	Construit par la CI depuis la branche feature	DEV
RELEASE CANDIDATE	Construit par la CI depuis la branche Develop	INT et QUAL
RELEASE	Promotion de l'artefact RC sur Artifactory	PROD

2.2 Design de la TOOLCHAIN

2.2.1 Architecture de la CD en mode IaaS

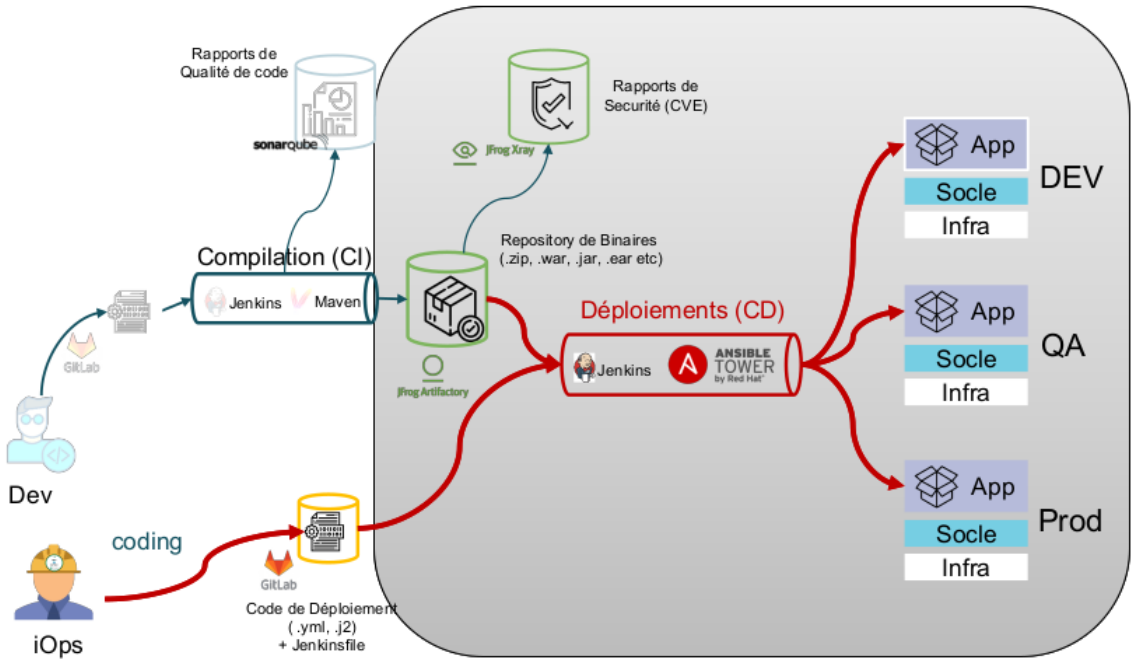

FIGURE 20 – la CD en mode IaaS

Lors du déploiement dans les VMs sur l'IT Cloud, l'IOPs développe le code de déploiement sur le playbook Ansible qui se compose de deux parties.

D'une part, le déploiement du socle avec les rôles qui déploient les fichiers de configurations pour l'application en mode service paramétrés par rapport à l'environnement cible (DEV, QA, Prod).

D'autre part, le déploiement des binaires livrés par la CI "qui est gérée par les DEV", puis analysés par Jfrog Xray pour les vulnérabilités et stockés sur le Jfrog Artifactory sous forme d'un zip, war ou un jar seront téléchargés par un rôle sur Ansible tower en utilisant un "User - Mot de passe" stocké manuellement sur Jenkins et envoyé vers Ansible Tower via une fonction dédiée.

Enfin, le lancement de l'application se fait en lançant le service associé avec un "**sytemctl start nom_service**" toujours en utlisant Ansible tower.

2.2.2 Architecture de la CD en mode PaaS

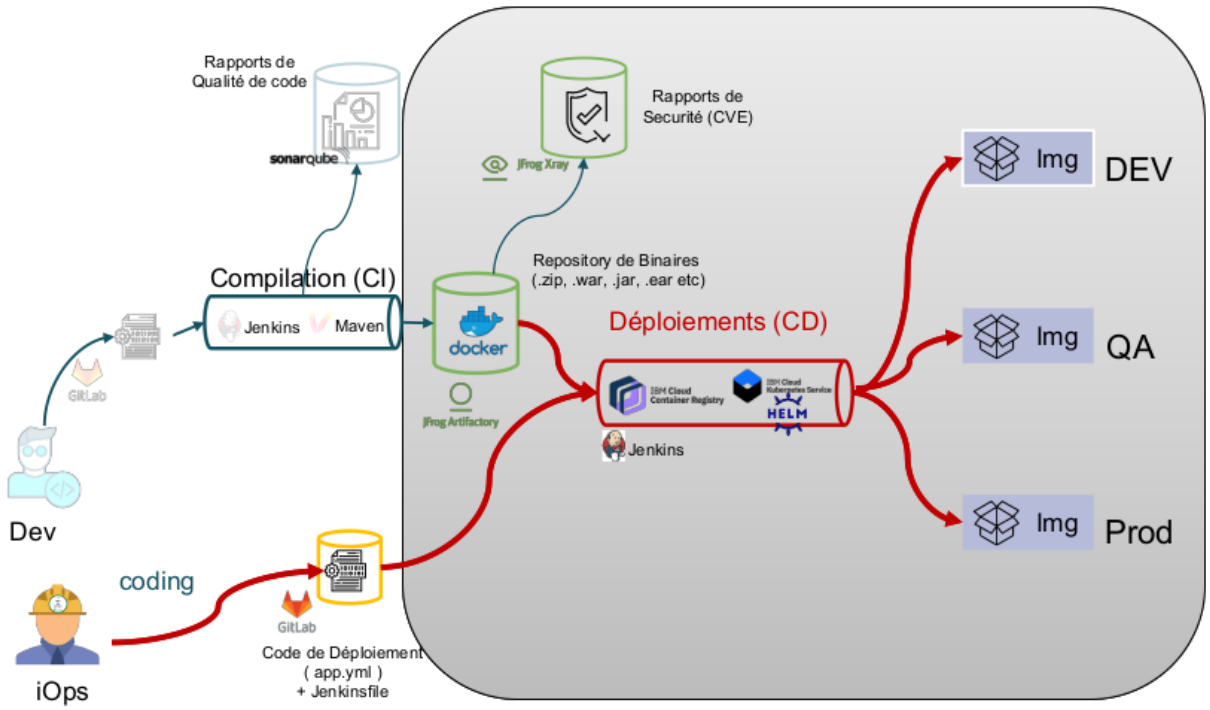

FIGURE 21 – la CD en mode PaaS

Concernant le déploiement en mode PaaS (KaaS), L'IOps développe le code de Déploiement avec les fichiers YAML de Kubernetes ainsi que les fichiers configurations paramétrés selon l'image Docker déployée, l'environnement cible, et contiennent la taille des Pods, des options spécifiques, les secrets à récupérer depuis Hvault etc.

La question qui se pose c'est comment Kubernetes récupère l'image Docker construite par les DEV?

C'est l'agent Jenkins qui est lui même déjà lancé sur un Pod récupère depuis le Master Jenkins les secrets nécessaire pour s'authentifier à la fois à l'artifactory et à l'IKS Registry [11] et s'en occupe pour récupérer l'image Docker depuis l'artifactory et l'envoyé vers l'IKS Registry sur CLoud où le cluster Kubernetes a accès.

Après cette étape, l'agent devra s'authentifier au Cluster Kubernetes en utilisant les secrets de la même manière et puis lance la commande magique **Helm install nom_Chart** pour déployer l'image.

11. c'est un Registre à image privée pour stocker et partager les Images avec IKS