**SOMMAIRE**

DEDICACE

REMERCIEMENTS

INTRODUCTION

1. APPROCHE METHODOLOGIQUE
2. Analyse de l’existant
3. Description de l’application
4. L’architecture
5. Fonctionnalités
6. Les forces et limites de l’existant
7. Points forts
8. Points faibles
9. PROPOSITION DE SOLUTION
10. Définition des objectifs
11. Objectifs généraux
12. Objectifs spécifiques
13. Présentation de la solution
14. Niveau technique
15. Infrastructure et outils
16. Architecture détaillée
17. Niveau organisationnel : architecture
18. Gouvernance
19. Changement culturel
20. Implémentation de la solution
21. Scripts et environnements
22. Configuration de la pipeline CI/CD et outils de monitoring
23. Stratégie de déploiement continue et gestion de versions.
24. VALIDATION DES TESTS
25. Validation fonctionnelle
26. Validation de l’infrastructure
27. Validation du monitoring et des alertes
28. CONCLUSION ET PERSPECTIVE
29. Conclusion
30. Perspectives et amélioration
31. ANNEXES

Annexe 1 : Script Terraform Complets

Annexe 2 : Playbooks Ansible complets

Annexe 3 : Configuration de Prometheus et Grafana

Annexe 4 : Configuration de CodeBuild et CodeDeploy

Annexe 5 : Schémas d’architecture et diagrammes

**INTRODUCTION**

De manière générale, dans l’industrie logicielle, la mise en production est non seulement une étape ultime attendue impatiemment par le métier ou le client mais aussi, constitue un véritable parcours de combattant pour toutes les équipes qui interviennent dans la chaine.

En effet, ce processus fait intervenir plusieurs équipes : les développeurs, opérationnels et testeurs. Et généralement les objectifs sont différents biens qu’au final tous sont unanime sur une seule chose celle que l’application soit en production.

Les développeurs cherchent à innover et à faire évoluer leur code en misant sur l’application des bonnes pratiques en la matière. Les opérationnels de leur côté cherchent à stabiliser le système d’information. Quant aux testeurs ils se concentrent sur les différents rapports et à documenter les différents tickets ou defects à remonter à l’équipe de dev. En plus de travailler de manière dispersé, chaque équipe utilise ces propres outils. Ce mode de fonctionnement aboutit toujours à des situations conflictuelles surtout quand les bugs commencent à apparaitre. Les conséquences qui en découlent sont : le retard dans les différents livrables, des coûts supplémentaires et la satisfaction du client est remise en cause bien que celle-ci soit un enjeu de détails. Il devient plus que primordiale d’adopter une autre approche qui permettrait d’unifier les équipes et les faire parler le même langage autour d’un objectif commun, celui de la satisfaction du client avec des livrables de qualités et dans les délais impartis. Ainsi, développeurs, opérationnels et testeurs travaillent ensemble avec un même processus et des outils connus de tous. C’est que le DevOps qui peut permette cela. Cette notion est basée sur la synergie entre les développeurs et les opérationnels qui est l’alignement de l’ensemble des équipes du système d’information sur un objectif commun tout en réduisant les conflits et éviter le retard dans les livraisons. C’est dans ce cadre que ce positionne notre projet de fin d’études.

Il s’agit pour nous de montrer comment tirer un large profit de l’adoption du DevOps au sein d’une équipe en mettant en place un processus d’intégration continue/livraison contenue avec les meilleurs outils en la matière dans un projet concret.

1. **APPROCHE METHODOLOGIQUE**
2. **Analyse de l’existant**

*Objectifs*

Notre objectif est de décrire le mode de fonctionnement c’est-à-dire les actions concrètes des différentes équipes concernées avant l’adoption de la culture DevOps d’une part, et d’autre part, montrer les limites.

1. **Description de la solution et les méthodes de travail**

Le projet utilisé, est constitué d’une partie front(front-end) et d’une partie back(back-end). La première est conçues avec du Nodejs et la seconde est basée sur l’architecture micro service réalisé avec java et springboot.

L’application est développée dans un environnement AGILE SCRUM. Au sein de l’équipe de développement dirigée par un scrum master, il y’a deux sous équipes : une dédiée au front-office et l’autre au back-office. La première s’occupe de tout ce qui concerne l’expérience utilisateur (interfaces d’utilisation, ergonomie, design…). Elle ajoute une couche au-dessus de la logique qui est sensée faire les traitements et qui est abstrait pour l’utilisateur. Quant à la seconde, c’est elle qui met en place toute la logique métier définie par les différentes règles de gestions.

Ces deux équipes travaillent en parfaite harmonie car il faut que les fonctionnalités soient techniquement mises en place par une (équipe) avant que l’autre ne vienne y ajouter une couche pour masquer cette complexité technique via des interfaces attrayantes et cohérentes. Pour se faire, le code source doit être accessible par tous.

Le code se trouve sur un outil de versionning de code (GitHub). Il y’a deux différents dépôts (back et front) et pour chaque dépôt, il existe 4 branches :

* Develop (branche dédiée à la fusion des développements)
* Stage (pour les tests réalisés par l’équipe des testeurs)
* Master (dédiée au code final et stable pour la production)
* Release (copie de la branche master avant chaque nouvelle mise en production)

Chaque développeur crée sa branche à partir de la branche develop. Et dès qu’il finit ses travaux, il pousse son code sur cette branche dont la fusion (merge) passe par un pull request qui est une étape de validation de son travail. Une fois que son travail est jugé conforme aux règles définis dans l’outils de contrôle de la qualité du code(SonarQube), alors son code est fusionné (mergé) dans celui de la branche sans conflit au cas contraire, il est notifié du rejet de sa demande de merge avec des détails pour l’aider à corriger les parties qui ne sont pas alignées avec la politique de qualité. Une fois les objectifs du sprint atteint, l’équipe des développeurs préparent alors une livraison à l’équipe de teste. Cette livraison doit forcément se faire sur un autre environnement qu’on appelle l’environnement de teste ou de qualification afin de permettre aux développeurs de continuer sur les autres sprints. Il faut donc demander la mise en place de cet environnement avec toutes ses exigences aux opérationnels. Dans cette demande, il faut fournir la liste des personnes à autoriser, les machines sur lesquels ces personnes auront accès au cas où il y’a plusieurs machines dans l’environnement (serveurs web, base de données, authentification…). Celui-ci est mis en place par l’équipe des opérationnels ou appelé aussi équipe système car c’est eux qui ont en charge toute l’infrastructure. Ils maintiennent et veillent sur la stabilité des environnements existants, en fournissent de nouveaux en fonction des besoins…

Un environnement est généralement une machine physique ou virtuelle sur laquelle est montée toutes les ressources nécessaires pour produire des cas d’utilisation conforment aux règles métier. Généralement, il y’a peu ou quasiment pas de différence avec l’environnement de production. Toutefois, cet environnement n’est pas accessible au client final. Il est y’a un autre environnement qu’on n’appelle pre-prod. C’est ce dernier qui est généralement ouvert aux équipes de teste du client final afin aussi de valider le travail fournir par toute l’équipe (dev, testeurs…) du prestataire en charge de développer la solution. Quand cette étape est validée, alors l’application est déployée sur le dernier environnement appelé l’environnement de production. C’est sur cet environnement que les clients finaux ont accès à l’application.

Pour déployer la solution et ce peu importe l’environnement, l’équipe de dev doit solliciter l’action des opérationnels en leur livrant l’application packagée pour le déploiement et voici les actions des opérationnels:

* Arrêts des conteneur (s’il y’a n’a)
* Suppressions des anciennes images
* Pousser l’image dans le registry
* Tirer l’image sur les hôtes des environnements
* Lancer de nouveaux conteneurs
* Vérifier les logs

Il faut remarquer que généralement des incompatibilités sont observées lors de l’exploitation des différents environnements. Par exemple l’accès à internet pour charger les dependences(librairies) dont dépend fortement la solution, problèmes d’accès et/ou d’écriture dans un répertoire, un fichier…

1. L’architecture

L’architecture utilisée pour la solution de notre cas d’étude est une architecture en micro-services. C’est un style architectural qui consiste à décomposer les fonctionnalités d’une application en de petits services fonctionnels, autonomes et communiquant entre eux via des APIs. Le choix a été porté sur cette architecture pour les raisons suivantes ;

* Flexibilité
* Scalabilité
* Résilience
* Gain de temps

1. Description des fonctionnalités

Tous ces services sont des images docker avec des bases de données différentes.

* Service catalogue
* Commande(orders)
* Livraison(shipping)
* Paiement
* Utilisateur(users)

1. Forces et limites de l’existant
2. Points forts :

En terme de points forts, mis à part le choix architectural, il n’ya véritablement pas de points forts à soulignés.

1. Points faibles

Pendant le cycle de développement de la solution, le chemin n’est toujours pas aussi linéaire tel que présenté actuellement, même si c’est l’idéal que souhaite toute équipe. Par exemple, au sein de l’équipe de l’éditeur de l’application, il peut avoir assez d’aller-retours entre les équipes de tests, les développeurs et celle des opérationnels. Lors de leurs travaux, les testeurs relèvent les dysfonctionnements, les cas de non-respect des règles métiers, les potentielles failles de sécurité… Ils créent ensuite des tickets soit en se servant d’outils tel quels Redmine, soit en utilisant un simple fichier Excel en fonction du projet. Ces différents points sont par la suite affectés à l’équipe de dev pour correction ou aux opérationnels si cela relève du fonctionnement des environnements. Une fois les corrections terminées, l’application est encore ramenée en environnement de qualification ainsi de suite jusqu’à ce que tout soit validé. Ce processus peut aussi être observé sur l’environnement de pre-prod donc entre le client final et l’éditeur de la solution.

Parlons à présent, des limites de ce mode de fonctionnement. En fait un tel dispositif dans un projet informatique sérieux comme le nôtre, entraine une série de préjudices et des risques qui peuvent malheureusement compromettre la réussite du projet d’une part et avoir un impact négatif sur la satisfaction des utilisateurs finaux d’autres part. Citons-en quelques-unes :

* Ralentissement du Time-to-market
* Cycles de développement longs : non-respect des délais

Les processus manuels et les silos entre les équipes entrainent des cycles longs retardant la mise en production de nouveaux produits ou fonctionnalités

* Difficultés à répondre aux besoins changeants

L’incapacité de déployer rapidement des modifications rend difficile l’adaptation aux besoins évolutifs des utilisateurs.

* Augmentation des risques
* Perte de qualité : qualité du logiciel réduite

L’absence d’automatisation des tests et de l’intégration continue peut conduire à augmenter les défauts de la solution par ricochet, une diminution de sa qualité globale

* Instabilité en production

Les déploiements manuels et les configurations complexes augmentent le risque d’erreurs de déploiement et d’incidents en pré-prod et/ou en production.

* Difficulté à Scaler
* Complexité accrue :

La gestion d’un environnement complexe devient de plus en plus difficile à mesure que le projet se développe.

Il faut parler également des coûts que peuvent engendre ce model par ce que dans un tel model, les coûts ne sont pas quasiment maitrisables.

De ce qui précède, il convient de retenir qu’un tel mode de fonctionnel est loin d’être parfait pour une équipe qui se veut productive et résiliente donc de pourvoir faire face aux changements auxquels est soumis le marché. C’est pourquoi nous proposons une solution concrète, celle d’adopter la culture DevOps. Dans ce qui suit, nous verront une implémentation complète de celle-ci avec les outils les plus prisés en la matière, et nous analyserons son impact.

1. **PROPOSITION DE SOLUTION**
2. **Définition des objectifs**

Notre objectif est de changer le mode de fonctionnement afin d’y adopter des pratiques plus transparentes et plus avantageuses. Il va s’agir de créer un cadre dans lequel Dev et Ops seront tous alignés sur les mêmes objectifs.

1. Objectifs généraux

Pour y arriver on procèdera comme suite :

* Utiliser et mettre en place des outils connus de tous pour faciliter l’interaction des dev et des Ops.
* Explorer les meilleurs pratiques pour mettre en œuvre une solution permettant d’augmenter la productivité.
* Mettre l’accent sur la surveillance en utilisant des solutions qui font l’unanimité auprès de toute la communauté.
* Garantir la satisfaction des utilisateurs finaux

1. **Objectifs spécifiques**

Il est question ici de zoomer sur les pratiques et les outils à utiliser afin d’atteindre nos objectifs.

* Réduire le temps moyen au maximum de résolution des incidents en mettant en place un pipeline CI/CD totalement automatisé et connu de tous.
* Augmenter la fréquence des déploiements grâce à une infrastructure adéquate.
* Automatiser 80% des tâches manuelles afin de minimiser la lenteur dans les interactions entre les équipes d’une part et d’autre part de réduire les coûts qui en décolleraient.

1. **Présentation de la solution**

Dans cette section, nous décrirons en détails toute la démarche à suivre ainsi que tous les outils que nous utiliserons dans la solution proposée tant au niveau organisationnel qu’au niveau technique.

1. **Niveau technique**

En parlant de technique il nous faut impérativement toucher le contexte technique ainsi que ses exigences.

1. **Infrastructure et outils**

Nous allons utiliser de nouveaux outils tel que ***terraform*** et ***ansible*** pour toutes les questions de la mise en place de l’infrastructure ainsi que tout ce qui est lié au provisionning de celle-ci. On se basera sur des scripts qu’on pourra formaliser pour produire des modèles (templates) qui seront une base commune de travail et de définition des différents environnements que prendra en charge notre infrastructure. Cette manière de faire assure l’action collarorative.

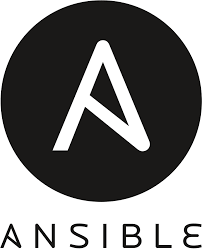
Bien que les deux outils soient des outils far pour le DevOps, force est de constaté que leur périmètre d’action est différent. Commençons par les présenter et définir leur périmètre par la suite.



* **Terraform**

C’est un outils souvent qualifié d’architecte d’infrastructure. Il nous permet de créer, modifier, supprimer des ressources d’infrastructure cloud grâce à du code d’où l’appellation « Infrastructure As Code (IaC) ». Grâce au code, on peut déclarer simplement dans un fichier, l’état désiré de notre infrastructure et il calculera les étapes nécessaires pour atteindre l’état souhaité. En plus, il est multi-cloud c’est-à-dire, il est compatible avec une large gamme de fournisseurs cloud (AWS, GCP, AZURE...).

Enfin, l’une des caractéristiques qui lui confère sa puissance est qu’il est capable de maintenir immutable l’état d’une infrastructure c’est-à-dire il peut détecter des changements ou modification non intentionnelles.



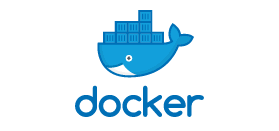
* **Ansible** (gestionnaire de configuration)

Il est spécialisé dans la configuration et le management de système existant. Aussi, Son volet impératif, nous permet de lui donner les actions à exécuter sur les machines cibles et il le fait avec perfection. En plus, il n’a pas besoin d’un agent pour s’exécuter ; ce qui fait qu’on peut l’utiliser facilement sur les OS. Enfin, il peut exécuter plusieurs fois les tâches sans effets de bords c’est pourquoi l’on dit qu’il est idempotent.

|  |  |
| --- | --- |
| ***Terraform(IaC)*** | ***Ansible*** |
| Infrastructure as code | Configuration et management |
| Déclaratif | Imperatif |
| Multi-cloud | Agenless |
| Etat immuable | Idempotent |

En terme de périmètre, Terraform est idéal pour la création et la gestion de l’infrastructure de base (réseaux, serveurs virtuels, stockage…). On dit souvent que terraform provisionne l’infrastructure. Quant à Ansible, il est parfait pour la configuration des système existants (installation de logiciels, configuration de services, déploiement d’applications). On le qualifie souvent aussi de configurer les serveurs ou machines provisionnées.

Ainsi, la combinaison de ces deux outils, donne une puissance au DevOps pour accomplir ses tâches sereinement.



* **Docker/Docker-compose**

Docker est incontournable dans l’arsenal du DevOps car il joue un rôle central. Il permet de conteneuriser les applications en les encapsulant dans un environnement isolé et contenant tout ce dont l’application a besoin pour fonctionner. Même si nous avons plusieurs conteneurs, il n’ya aucun risque de conflit à cause du caractère isolation des conteneurs. L’exécution d’un conteur est très rapide. Aussi, il permet d’assurer la scalabilité en permettant de créer et de détruire en un temps records un conteneur.

Quant à docker-compose, il est un complément de Docker et nous permet de gérer des applications de Docker composées de plusieurs conteneurs liés entre eux. On parle de gestion et exécution d’applications multi-conteneurs.



* **AWS** (fournisseur cloud)

C’est également un acteur incontournable du fait de son rôle dans la mise en œuvre des pratiques DevOps. En plus d’être le leader dans le domaine. En fournissant, une large gamme de services, il prend en compte les méthodes argiles ainsi que l’automatisation des processus déploiements. Il est tout en un. Nous utiliserons plusieurs de ses services pour implémenter notre solution par exemple :

* Outils DevOps natifs (codeBuild, codeDeploy, codePipeline)

CodeBuild/codeDeploy ont un rôle comparable à ce que fait Jenkins

* EC2
* EKS (Elastic kubernete Service)
* cloudFormation
* Etc



* **Outils de monitoring**

Nous utiliserons prometheus et grafana.

***Prometheus*** : c’est un système de monitoring open source qui collecte les métriques à intervalle réguliers d’une ou d’un ensemble d’applications et les stocks temporairement.

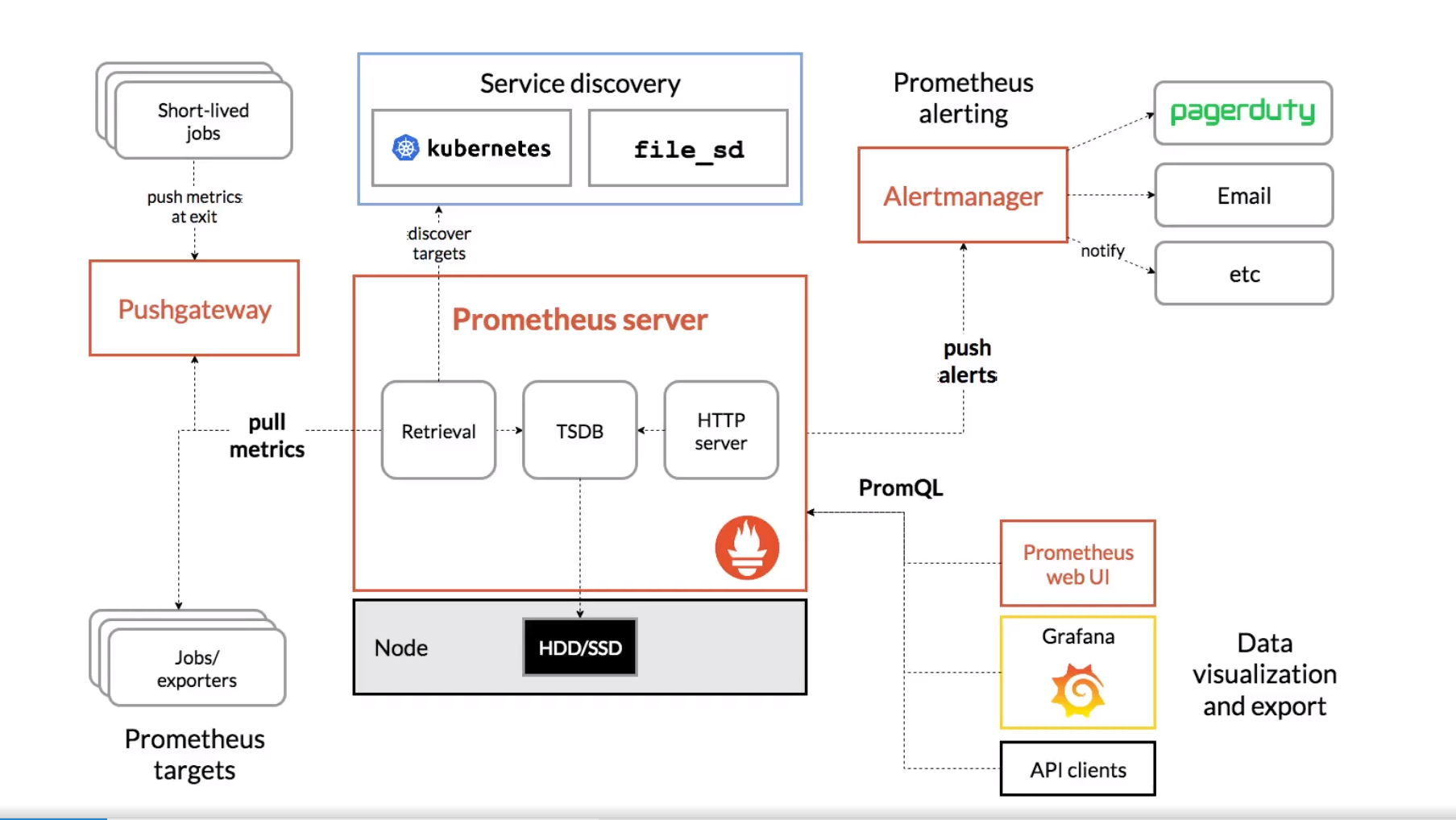
***Grafana*** : est une plateforme open source de visualisation. Elle est réputée pour la création et la personnalisation des tableaux de bords interactifs à partir des données collectées par prometheus.



A ces deux outils, pour une meilleure diffusion des notifications, nous ajouterons **Slack** qui est une plateforme d’échange à partir de laquelle nous créons un groupe dans lequel toutes les équipes sont regroupées. Enfin, créons également un mail de diffusion pour certaines notifications bien spécifiques telles que les sauvegardes, l’exécution des jobs…



***Kibana*** : est une interface web de la suite Elastic Stack. Il est principalement utilisé pour la recherche, la visualisation et l’analyse des données logs. Il est très souvent utilisé aussi bien par les testeurs que par les développeurs.



1. **L’architecture détaillée**

b-1) Les environnements

1. *Développement*

Nous utiliserons de simple instances EC2 pour configurer notre environnement de développement et les test initiaux. Toutefois ce n’est pas une exigence en soit. L’environnement de développement peut être bien configuré en interne avec un accès sur internet surtout pour joindre le compte GitHub pour y déverser le code source.

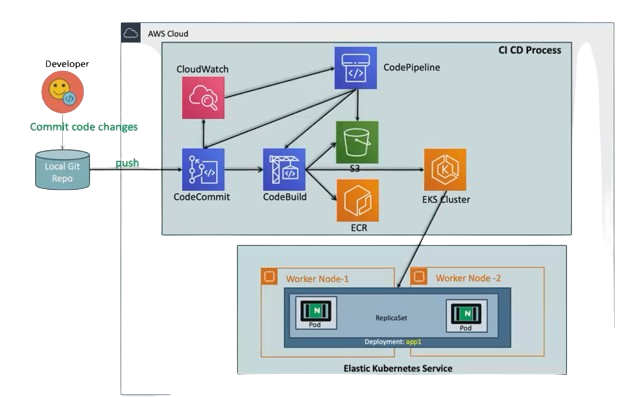
1. *Test ou stage*

On utilisera EKS pour gérer les conteneurs déployés et nous permettre de faire les tests fonctionnels au niveau de l’architecture d’une part, et d’autre part, les tests fonctionnels liés aux exigences métiers qui sont à la charge de l’équipe de qualification.

1. *Production*

Quant au déploiement final de la solution, elle se fera sur EKS avec l’option autoscalling et un monitoring renforcé.

b-2) **Schéma de l’architecture complète**





Nous utiliserons ***kubernetes*** comme orchestrateur de conteneur afin de donner plus de souplesse à la gestion de nos ressources et de déléguer une grande partie de la phase de déploiement continue.

1. **Niveau organisationnel**

Une telle solution doit son succès à une meilleure organisation des équipes qui font le travail.

1. **Gouvernance**

A ce niveau, nous préconisons une équipe de modèle en T. Dans ce type d’équipe, chaque membre possède une expertise approfondie dans un domaine (développement, infrastructure, test…) tout en ayant une compréhension globale des autres domaines. Nous rappelons que nous sommes dans une équipe pluridisciplinaires car on n’a les développeurs eux-mêmes repartis en deux sous-groupes (les dev back et les dev front), l’équipe de qualification qui gère tous ce qui concerne les tests fonctionnels et l’équipe des opérationnels.

1. **Culture DevOps au sein de l’équipe**

Ici quand on parle d’équipe c’est tous les acteurs mis ensemble. C’est-à-dire tout ce qui se fait ou est décidé dans le cadre du projet est connu de tous. Par exemple, l’infrastructure mise en place est connue, les outils utilisés sont adoptés par tous. Voici quelques rôles important qu’il faudra avoir dans l’équipe (c’est pas un impératif de les avoir tous) :

* **Ingénieur DevOps**

Il assure la coordination entre les développeurs et les opérationnels, automatise les processus, met en place et assure la maintenance de pipeline CI/CD

* **Développeurs**

Il créé et maintient le code de l’application. Il utilise le pipeline et tire profit de l’automatisation du processus de déploiement et d’intégration continue.

* **Testeurs**

Elaborent les plans de tests, les exécutent manuellement ou de manière automatisée et assurent la qualité du produit.

* **Opérationnels** (administrateurs système, base de donnée, de réseaux…)

Gère l’infrastructure (serveur, réseaux, stockage)

* **Spécialiste en sécurité**

Il garantit la sécurité des applications et des infrastructures

1. **Implémentation de la solution**

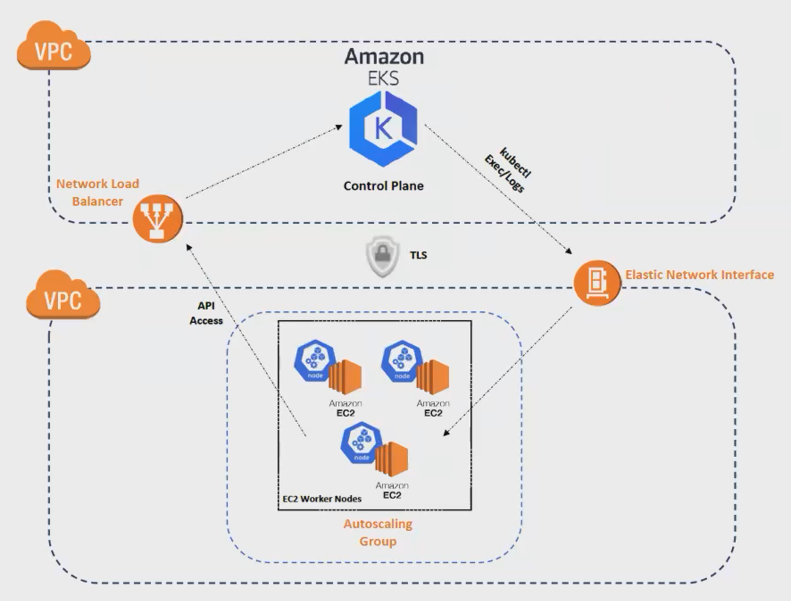
Dans cette section, nous mettrons en œuvre étape par étape tout le processus en mettant en place tous les outils dont on n’a parlé en détails précédemment. En effet, nous allons construire les environnements tests, pre-production, production, le pipeline CI/CD à l’aide d’un fournisseur(AWS) en misant sur l’automatisation et clôturer par le monitoring.

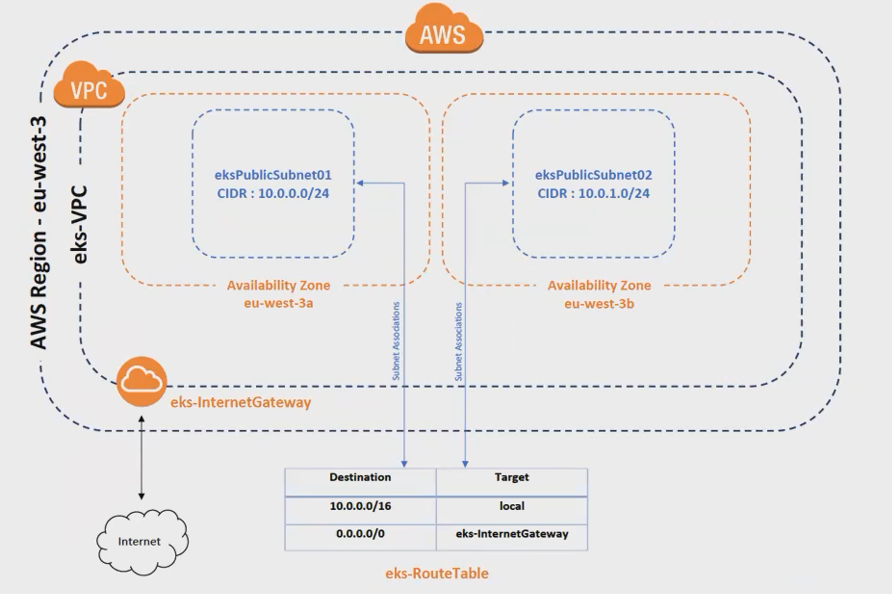
1. **Scripts de déploiement pour créer les ressources AWS nécessaires.**

Il s’agit des ressources suivantes : environnements de test et de production à l’aide de scripts Terraform er Ansible:

Les scripts Terraform vont nous permettre de mettre en place les bases de l’architecture entre autres les VPC, les subnet (sous-réseaux), les groupes de sécurité, le cluster EKS avec les différents rôles IAM…

Les scripts ansible vont nous permettre de provisionner les machines cibles en installant les dépendances requises, aider au déploiement des applications…





***Script Terraform*** :

Ils sont utilisés pour provisionner l’infrastructure AWS y compris les ressources telles que les instances EC2, les clusters EKS, les groupes de sécurité, les VPC et les rôles IAM (voir un exemple en annexe 1).

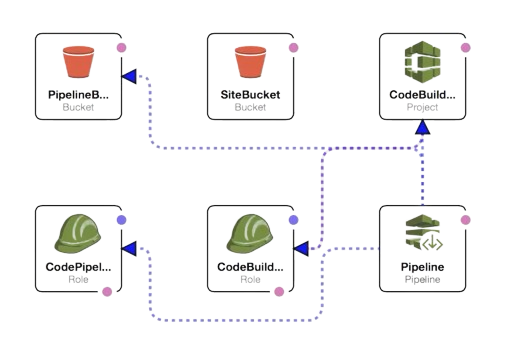
***Script playbooks Ansible :***

Les playbooks Ansible sont utilisé pour configurer les instances EC2, déployer les applications et gérer les configurations des environnements.

1. **Configuration du pipeline CI/CD et des outils de monitoring**

Cette étape est cruciale pour assurer un déploiement fluide et assurer une surveillance efficace des applications. Il sera question de détailler les étapes de configuration nécessaires pour mettre en œuvre ces systèmes en utilisant AWS, CodeBuild, CodeDeploy, prometheus et Grafana.

Par ailleurs, pour toutes actions qui seront déléguées aux outils DevOps de AWS, il faut obligatoirement un rôle pour accomplir sa tâche. C’est en fait des jobs qui seront créés par action et grâce aux rôles ces jobs vont interagir entre eux.

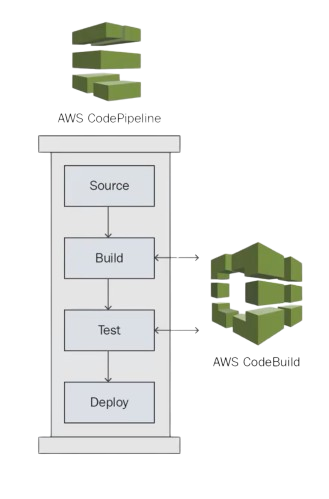


1. **Intégration avec Github**

La première étape consiste à :

* Intégrer Github avec CodeBuild et CodeDeploy pour déclencher automatiquement les build à la suite de push sur la branche principale.
* Configurer les permissions nécessaires afin que CodePipeline d’AWS puisse se connecter au dépôt Github.

***Configuration de CodeBuild* :**



Il sert à compiler et à lancer les tests des micro services. Il est généralement composé des phases suivantes :

* **Install** : installe les dependences nécessaires
* **Build** : compile le code et construit les images Docker
* **Post\_build** : Pousse l’image construite vers le registre Docker ou ECR

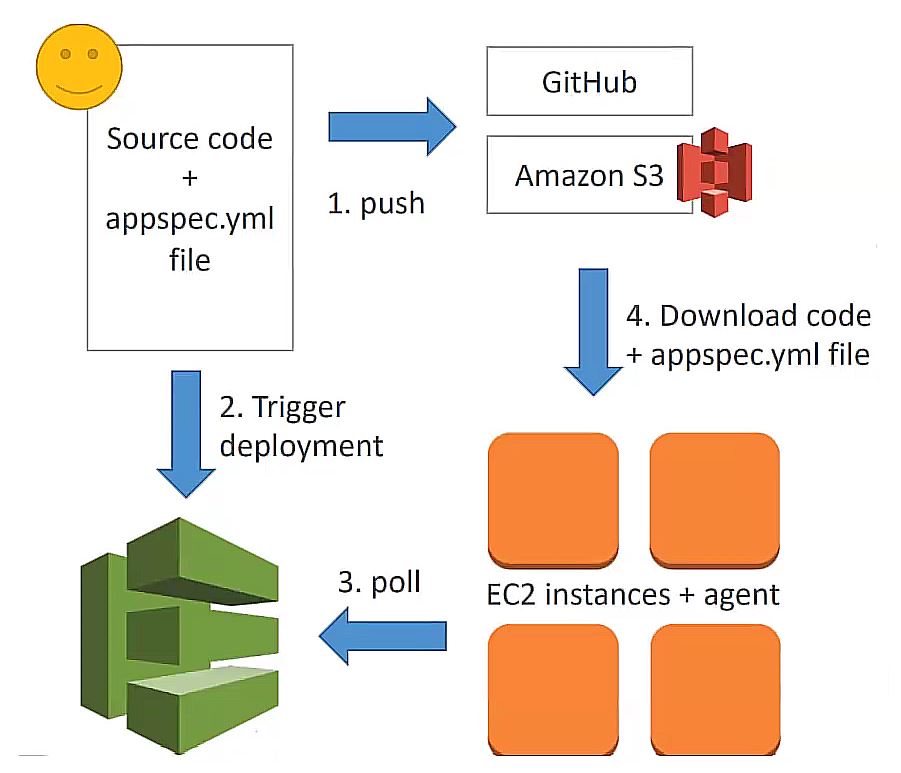
Pour donner les instructions à l’image gitlab ou on n’a gitlab-cli ou encore jenkinsFile pour Jenkins, pour CodeBuild on n’a **buildspec.yaml**

**Fonctionnement :**

*A sa configuration, il va demander une source. Avant le 25 Juillet 2024 l’on mettait codeCommit mais maintenant AWS l’a remplacé par GitHub. Ensuite on lui définit un rôle qui sera demandé afin de permettre au job qui lui est associé de faire son travail. Aussi, vient définition de son contexte qui l’ensemble des ressources nécessaires à l’exécution de nos taches d’une part et d’autre part tout ce qui est variables. Il faut également configurer le cache pour optimiser le build en terme de téléchargement de librairie dont a besoin l’application qu’on construit et conserver l’artefact.*

Nous avons un exemple de code en annexe.

***Configuration de AWS CodeDeploy :***

******

Il sert à déployer les micro-services sur EKS. Ce script est constitué de deux grandes parties

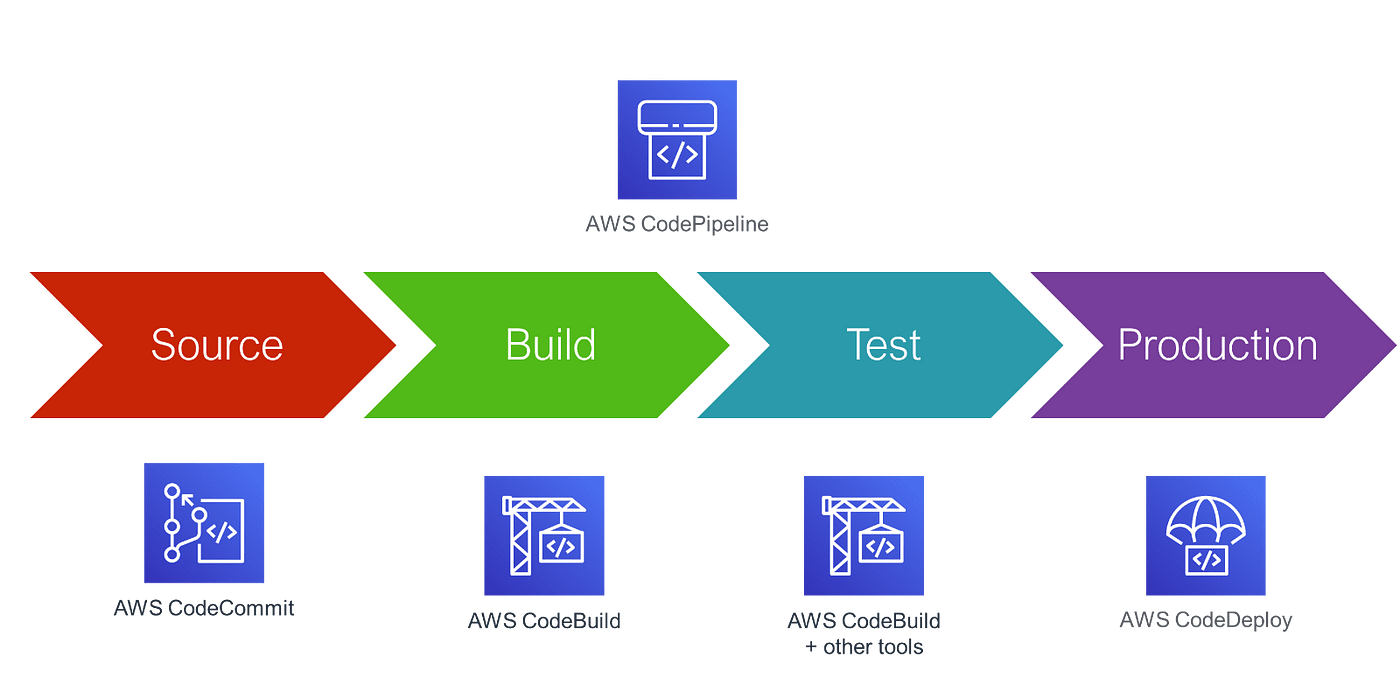
* Définition de l’application CodeDeploy
* Création du groupe de déploiement avec les configurations nécessaires pour un déploiement blue/green

Nous avons un exemple de code en annexe.

**Fonctionnement** :

*Il a un fonctionnement qui est difficile à cerner. C’est un job mais qui ne se déclenche pas tout seul. Tout part d’un fichier appspect.yml grâce au trigger, il lira ce fichier. Ensuite, Il y’a un agent qui est installé sur l’instance EC2 qui demande au job s’il y’a du travail. Il demandera à l’instance d’aller chercher les fichiers depuis S3. C’est dans le fichier appsepc qu’on n’a le séquencement d’exécution des scripts récupérés pour le déploiement.*

***Configuration de AWS CodePipeline :***

******



Son rôle consiste à orchestrer le processus CI/CD. Son script est structuré en trois (3) étapes :

* **Source** : récupère le code source depuis Github
* **Build** : compile et test le code avec CodeBuild
* **Deploy** : déploie l’application avec CodeDeploy

Nous avons un exemple de code en annexe.

1. **Configuration des outils de monitoring**

***Configuration de prometheus***

Comme nous l’avons dit, il est utilisé pour colleter les métriques des microservices ou de l’infrastructure. Sons code est constitué de deux (2) parties clés :

* **Scrape\_interval** : intervalle de temps entre chaque métrique.
* **Scrap\_configs** : configuration des cibles à surveiller (API Kubertes, nodes, pods, …)

Nous avons un exemple de code en annexe.

***Configuration de grafana***

Comme nous l’avons déjà dit, il permet de visualiser les métriques collectées par prometheus. Il se compose des parties suivantes :

* Configuration de l’authentification anonyme
* Définition des utilisateurs et des permissions.

***Configuration de kibana***

Il est utilisé pour visualiser et analyser les logs collectés. Il fonctionne conjointement avec Elasticsearch qui index les logs. Pour qu’il fonctionne bien sur notre environnement, il lui définir un rôle IAM pour kibana et elasticksearch (voire en annex).

Le fichier de configuration de kibana est constitué des parties suivantes :

* **Server.port** : le port sur lequel écoute Kibana
* **Server.host** : l’adresse sur laquelle écoute Kibana
* **Elasticsearch.hosts** : adresse(s) du ou des clusters elasticsearch
* **Kibana.index** : index elasticsearch dans Kibana

Nous avons un exemple de code en annexe.

***Configuration des alertes***

Pour recevoir les alertes en cas de problèmes, Prometheus doit être configuré avec Alertemanager pour envoyer des notifications via slack et par mail.