Jeudi, le 02/12/2021

RAPPORT PROJET *TWEETOSCOPE*

Auteurs : Oussama El M'Tili, Worou AKIYO, Divin KIMALA

Encadrants : Virginie Galtier, Frederic Pennerath

1. Introduction

Pour mettre en place notre application, permettant de prédire la popularité d'un tweet en utilisant les outils théoriques vus en cours de modèles statistiques, la solution mise en place a suivi une **démarche DevOps**.

Ce choix se justifie d'une part par la facilité et la flexibilité que nous offre cette démarche en accélérant le temps de développement et de mise en production d'une fonctionnalité, comme par exemple le fait d'ajouter le nœud gérant la collecte des tweets provenant du générateur de Tweet. D'autre part, cette démarche nous a permis de réduire les erreurs lors des livraisons et d'assurer une continuité de service de l'application.

2. Démarche

Après avoir créé le dépôt git distant permettant de gérer notre code source, nous utilisions **GitLab CI,** outil qui nous a permis de mettre en place les étapes du cycle de notre application.

Les étapes du cycle (de la pipeline) de notre application sont les suivantes :

1) **Test :** Dans un premier temps, si vous regardez nos anciens commit, nous avons voulu créer un stage permettant de faire tourner d'abord une image docker sur le dépôt distant. Cela a posé un problème, et ce stage n'a pas été intégré dans notre pipeline, mais nous effectuions systématiquement de **tests** de l'application en général avant de faire un push dans le répertoire distant. Ces **tests** consistaient à faire fonctionner l'application sur **Minikube** en local.
2) **Quality** : Ce stage est implémenté dans notre pipeline et permet de générer un fichier index.html dans le répertoire **codequality-code**, permettant de faire un reporting du code produit.
3) **Build** : permet de construire 9 images docker dans le répertoire distant.
4) **Deploy** : le dernier stage consiste à déployer ou à faire de la livraison continue en déployant l'application sur le cluster de l'école.
   Pour ce faire, nous savions faire du déploiement via Gitlab en utilisant d'autres services tiers (Microsoft Azure) mais sur le cluster de l'école, la procédure à faire était de récupérer les deux fichiers qui se trouvent dans le dossier **Deploiement/cluster** et d'exécuter la commande **kubectl apply -f NOM_FICHIER.yml**

Comme dit précédemment, le stage du build permet de dockeriser la solution et construit 9 images se trouvant dans le dossier **Container Registry** dans le répertoire distant. Ces images ont été construites dans des Dockerfiles.

3. Mise à l'échelle de l'application sur le cluster de l'école

Dans un premier temps, nous travaillions avec Minikube (donc un seul nœud) et ensuite avec Kubernetes.

Pour déployer les différentes composantes de l'application, nous avons utilisé Kubernetes qui nous a permis de livrer différentes composantes de l'application sur plusieurs nœuds plus précisément 5 nœuds (voir vidéo).

Enfin, pour nous assurer que notre solution est tolérable aux pannes, dans le fichier yml de Kubernetes, nous avons systématique répliquer (à l'aide du tag replicas) toutes les images docker sur différents nœuds (5 la plupart du temps), en rendant ainsi la solution proposée au moins 4 (peut varier si deux pods répliqués sont dans le même nœud) fois tolérables aux pannes.

Dès qu'il y a un changement sur le répertoire distant et après que la pipeline ait passé sur le dépôt distant, si le changement provient de la branche master distant, nous récupérons facilement le changement car nous avions mis la valeur de ImagePullPolice à Always (oblige Kubernetes à chercher toujours dans le répertoire distant au lieu de ce qu'il y a de le cache quand j'utilise la commande kubectl apply -f FICHIER.yml).