École nationale supérieure d'informatique et de mathématiques appliquées

Projet Systèmes Distribués pour le Traitement de Données
Analyse des vues sur Wikipedia

Maxime Gourgoulhon
Alexandre Podlewski
Julie Saouli
Loïc Schamber
Maxime Turlure

ISI
3A

21/01/2019

Contents

1 Introduction

La vocation de notre projet est de réaliser une analyse globale de la consultation des pages sur l'encyclopédie en ligne Wikipédia. Au travers de l'API Wikimedia disponible gratuitement et librement sur internet nous souhaitons effectuer des traitements sur les articles de l'encyclopédie afin d'un dégager les tendances sur une période donnée et les afficher sur une application Web.

2 Description des composants logiciels

Pour construire notre application nous avons utilisé les composants logiciels suivants (que nous allons détailler dans cette partie) : Kubernetes, MongoDB, Spark et Kafka.

2.1 Orchestration : Kubernetes

2.1.1 Choix de l'orchestrateur

Les problématiques d'une application distribuée sont la facilité de déploiement et la scalabilité de l'application. Pour répondre à ces problèmes les orchestrateurs sont des outils de prédilection. La vocation d'un orchestrateur est de configurer, de coordonner et gérer un ensemble de machines pour qu'elles effectuent un ensemble de tâches de manière distribuée et coordonnée.

Pour ce projet, nous avons choisit Kubernetes comme orchestrateur. Kubernetes est un orchestrateur couramment utilisé pour déployer des application de taille moyenne et dont il existe plusieurs variantes selon l'utilisation qui en est faite : minikube (déploiement en local), AWS EKS Elastic Container Service (déploiement sur AWS) et kubeadm (la version généraliste et la plus utilisée). Notre choix c'est porté sur Kubernetes par intérêt. Cet orchestrateur est très suivi ces derniers temps et beaucoup de ressources existent sur internet pour apprendre à le configurer. La disponibilité de la documentation a donc été un facteur de choix décisif.

2.1.2 Fonctionnement de Kubernetes

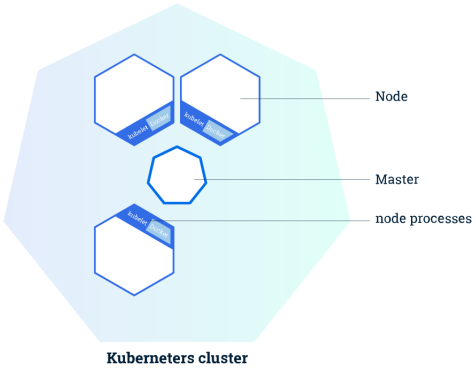

Figure 1: Organisation d'un cluster Kubernetes

Kubernetes est un orchestrateur basé sur Docker pour exécuter les taches demandées. Le rôle de Kubernetes est de distribuer une charge de travail sur un ensemble de machine distinctes appelé "cluster". Kubernetes est capable de gérer la réplication. C'est-à-dire qu'une tâche peut être répliquée plusieurs fois afin de s'assurer qu'elle sera bien effectuée, même en cas de panne d'une des machines.

Ce cluster est constitué de deux types de machines différentes (Figure 1) : les nodes qui exécutent les tâches et le master qui gère l'ensemble des nodes.

Le master

Le master agit comme le cerveau du cluster. C'est lui qui décide de la répartition du travail sur les différents nodes et la communication entre ces derniers. Il gère la mise à l'échelle de l'ensemble du cluster en fonction de la charge du travail. Il s'assure que le travail continuera en cas de panne d'un ou de plusieurs nodes en redémarrant

la tâche sur un node sain. Bref son rôle est vital.

Les nodes
Les nodes représentent la majorité des machines présentes dans le cluster. Elles ont pour rôle d'effectuer le travail qui a été demandé par le master et de le tenir au courant lorsqu'un évènement survient (erreur, fin de la tâche).

Admettons que l'on souhaite réaliser une tâche. Premièrement, il faut informer le master du travail à réaliser. Le master va ensuite créer le nombre de containers Docker suffisant pour exécuter cette tâche (en fonction de la réplication souhaitée). Ces conteneurs sont appelés "pods". Dès lors qu'un des nodes du cluster est disponible, le pod est lancé sur le node disponible. En cas de panne du node, le master en sera informé et redémarrera un autre pod sur un node sain.

2.2 Stockage : MongoDB

Le rôle d'un Système de Gestion de Base de Données Distribuée (SGBDD) est de stocker de manière persistante un ensemble de donnée de manière cohérente. La différence majeure avec un système de gestion de base de données classique est sa capacité à répartir les informations sur différentes bases de données situées sur différentes machines et donner l'impression à l'utilisateur de n'interagir qu'avec une seule et même base. L'intérêt d'un SGBDD est de diviser les informations dans différentes bases (pour en réduire la taille), dupliquer les données pour qu'en cas de panne, rien ne soit perdu.

Nous avons choisit MongoDB comme SGBDD. MongoDB utilise une approche NoSQL pour stocker les données. MongoDB propose la possibilité d'ajouter du "sharding" à notre base de données, c'est-à-dire de répartir les données dans plusieurs bases de données différentes. Tout comme Kubernetes, MongoDB suit l'architecture Master-Slave, ici le noeud maître est appelé "primary node" et les esclaves sont nommés "secondary nodes". Le primary node reçoit les requêtes de la part d'un client et est chargé d'appliquer les traitements nécessaires avant de transmettre la requêtes sur les secondary nodes.
La simplicité de la mise en place de MongoDB par rapport à d'autres SGBDD explique en partie la raison de notre choix. Il est aussi le plus populaire des SGBDD sur des projets de taille moyenne et donc dispose d'un grand support et d'une large documentation.

2.3 Traitement : Spark

Lorsque l'échelle est raisonnable, un calcul peut facilement se distribuer sur les différents coeurs d'une machine. Mais lorsque celui-ci fait intervenir plusieurs méga, voir giga-octets de données, cela ne suffit généralement pas. Pour cela on fait appel à des outils d'analyse distribuée. Leur rôle est de diviser un calcul sur plusieurs machines afin d'augmenter considérablement le nombre de coeurs disponibles pour une opération.

Nous avons fait appel à Spark pour analyser nos données de manière distribuée. Spark suit lui aussi l'architecture Master-Slave. Le noeud master reçoit les requêtes du client. Il analyse ces requêtes et réalise une subdivision du travail en sous-tâches. Ces sous-tâches sont ensuite assignées aux slaves qui les exécutent et renvoient les résultats au master. Le master joue le rôle de coordinateur et s'assure que les tâches s'exécutent toutes et dans le bon ordre. En cas de panne, il relance la tâche sur un slave disponible.

2.4 Agent de message : Kafka

Dans une application distribuée, on a souvent besoin de faire communiquer les différents composants de cette dernière. Il faut donc un outils capable de faire dialoguer ces composants. Pour cela, on utilise un agent de message. L'agent de message est chargé de recueillir est de propager les messages qui sont émis sur le système. L'agent de message doit garantir que tout message qui sera émit sera bien délivré à toutes les machines qui doivent lire ce message. Le message sera donc dupliqué par l'agent de message pour être sûr qu'en cas de panne, le message sera bien délivré à tout le monde.

Nous avons choisit Kafka comme agent de message. Kafka fonctionne sur le principe publisher/subscriber. C'est-à-dire qu'un processus peut souscrire à un évènement à tout moment (appelé "topic" sur Kafka) et commencer directement à recevoir tout les messages de ce topic une fois qu'il y a souscrit. Si un processus publie un message sur ce topic, ce message sera reçu par tous les processus ayant souscrit à ce topic.

Kafka fonctionne aussi sur le principe de Master-Slave. Kafka est basé sur Zookeeper. Zookeeper agit comme un master par dessus le noeud maître de Kafka. Il permet en outre de contrôler le processus d'élection de leader si le noeud maître de kafka tombe en panne. Dans une autre mesure il permet de gérer une configuration partagée par l'ensemble des noeuds Kafka.

3 Architecture logicielle

3.1 Architecture globale

Le schéma ci-contre Figure 2 représente l'architecture globale du cluster Kubernetes. Le système utilise les éléments de la pile logicielle décrite précédemment.

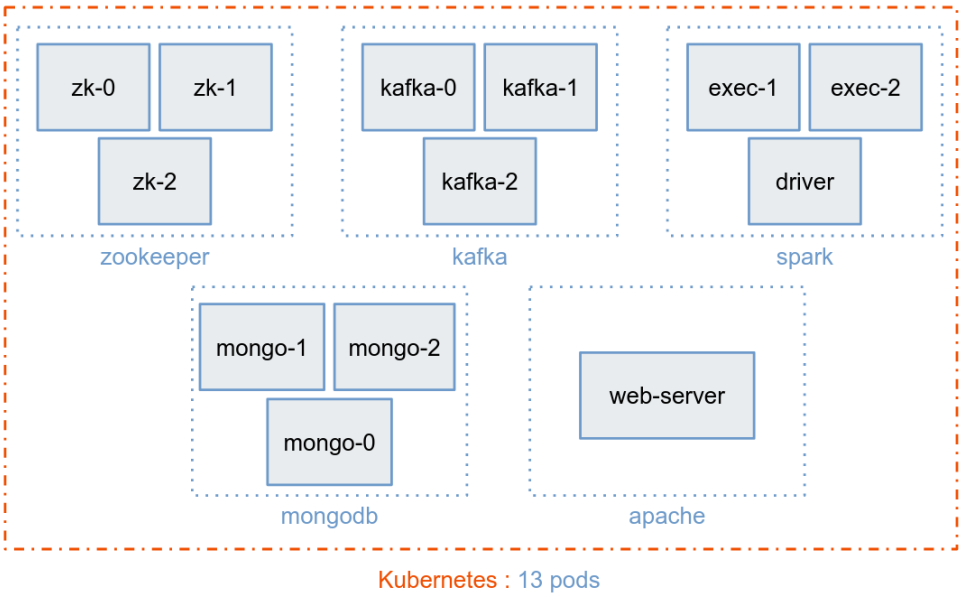

Figure 2: Schéma de l'architecture globale du cluster

Le service Spark est composé d'un driver et de deux exécuteurs qui requiert chacun 1 coeur de calcul et 1Go de RAM. Tandis que les services Zookeeper, Kafka et MongoDB sont répliqués trois fois afin d'assurer la tolérance aux pannes et requiert chacun 0.5 coeur de calcul et 1Go de RAM. Le serveur web Apache est quant à lui déployé sur un pod à part et requiert 0.5 coeur de

calcul et 1Go de RAM.

Les services de notre applications ont donc besoin de 8 coeurs de calcul et de 13Go de RAM. Il faut ajouter à cela coût de l'utilisation de Kubernetes. Le cluster de l'application compte au total treize pods répartis sur sept instances t2.large d'AWS. Les machines t2.large d'AWS dispose de 2 vCPU et de 8Go de RAM chacune. Dans l'état actuel du projet chaque node de Kubernetes est utilisé à 55% de ses capacités (en terme de de coeurs de calcul) et à 30% de ses capacités (en terme de mémoire).

3.2 Processus d'analyse

L'application est divisée en deux parties : une partie écriture et une partie lecture/requête. Les pipelines de chacune des parties sont présentées en Figure 3.

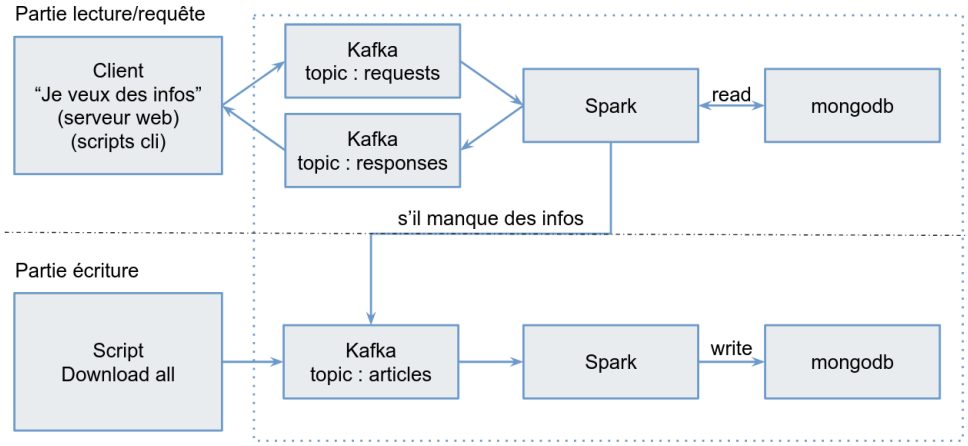

Figure 3: Schéma du fonctionnement de l'application

La partie écriture consiste à peupler la base de données en récupérant progressivement les données de chaque nouveau jour depuis l'API Wikimedia. Ces données sont téléchargées par des scripts puis envoyées via le topic "articles" de Kafka. Spark réceptionne ces données et les insèrent dans la base de données tout en vérifiant l'absence de doublon.

La partie lecture/requête consiste à répondre aux requêtes clients provenant du site web. Le serveur web appelle les scripts Python permettant d'effectuer les requêtes et obtenir les résultats via Kafka. Pour cela, on définit deux topics, respectivement "requests" et "responses". Les requêtes provenant du topic "requests" sont récupérées par Spark, qui se charge alors d'interroger la base de données MongoDB. En fonction des résultats de la requête à la base de données, Spark détermine s'il manque des informations. Si c'est le cas, Spark télécharge les données manquantes et les envoie à Kafka via le topic "articles" afin que ces données puissent être rajoutées dans la base de données. Finalement, Spark renvoie via le topic "responses" la réponse à la requête client, comprenant les données extraites de la base de données ainsi que les nouvelles données téléchargées.

En fonction des requêtes client, Spark applique également des traitements sur les données à renvoyer comme la constitution d'un classement des articles en fonction du nombre de vues sur une période donnée.

3.3 Tolérance aux fautes

Kubernetes est capable de détecter automatiquement lorsqu'un pod devient inaccessible et d'en informer le master afin qu'il puisse en démarrer un nouveau. Un nombre de pods répliqués minimal peut être indiqué (typiquement deux), permettant d'assurer que Kubernetes s'efforcera de toujours maintenir le nombre de pod supérieur à ce minima. En ce qui concerne une possible indisponibilité du master, il est possible de mettre en place des services permettant de gérer la perte du master (réplication du master, sauvegarde de l'état du cluster dans un service etcd) mais qui ne sont pas mis en place sur notre cluster.

Afin d'assurer une certaine tolérance aux pannes, chaque service (MongoDB, Spark, Kafka) est répliqué trois fois. Cette réplication permet également d'augmenter la disponibilité du cluster.

MongoDB permet la réplication d'une base de données en plusieurs instances contenant le même set de données. En cas de panne du primary node, un nouveau primary node est élu afin de permettre au cluster de continuer à fonctionner. Dans notre application, MongoDB est répliqué trois fois ce qui permet la tolérance d'au moins une panne.

Spark est divisé en un driver et des exécuteurs. En cas de panne d'un exécuteur, un nouvel exécuteur est instancié afin de traiter la tâche ayant échoué, et ce même si tous les exécuteurs tombent en panne. Cependant, le driver, unique, est un single point of failure. En utilisant Mesos à la place de Kubernetes, il aurait été possible de sauvegarder l'état du driver et de le redémarrer en cas de panne.

Kafka permet la réplication des instances et des partitions de logs gardant la trace des messages de chaque topic. En cas de panne d'une instance leader, chacun des réplicas est éligible à devenir le nouveau leader. Dans notre application, Kafka est répliqué trois fois ce qui permet de tolérer jusqu'à deux pannes (donc une seule instance survivante). Zookeeper, utilisé en complément de Kafka, est également répliqué trois fois ce qui permet de tolérer une panne.

En testant la robustesse de notre application, nous avons corroboré ces valeurs théoriques. Comme nous estimons que les machines AWS sont suffisamment fiables, nous avons configuré le déploiement de notre application pour qu'elle tolère au maximum 1 faute sur le service mongo, 2 fautes sur le service kafka et 1 faute sur le service Spark (tant qu'elle ne concerne pas le driver).

4 Application

4.1 Présentation

Notre projet ayant pour but de permettre de présenter les tendances sur Wikipédia, notre application est centrée sur deux critères particulièrement importants:

- L'évolution des vues d'un article en fonction du temps.
- Le top des articles les plus vus sur une période donnée.

Nous souhaitions pouvoir disposer de l'application à la fois en mode console, plus pratique lors du développement, mais également afficher les informations sur une interface web, plus accessible et agréable pour les utilisateurs. Ainsi nous avons séparé l'application en deux parties: des scripts Python contenant la partie fonctionnelle de l'application et un serveur PHP hébergeant un site permettant d'effectuer les requêtes de manière plus intuitive et mettant en forme les résultats.

4.2 Scripts Python

Deux scripts Python sont utilisés afin de faire fonctionner le système :

analyse.py

Ce script sert à récupérer et enregistrer les données d'un jour ou d'une période donnée. Il effectue une requête à l'API Wikimedia puis envoie les informations à notre système via Kafka afin de permettre l'enregistrement les données dans MongoDB.

request.py

Ce script a un double rôle en fonction de ses paramètres : il permet de récupérer les vues d'un article ou bien de récupérer les articles les plus vus sur une période donnée. Il fait la requête correspondante via Kafka afin de récupérer les données voulues depuis la base de données et les afficher dans la console au format JSON.

4.3 Interface web

Les traitements en ligne de commande n'étant pas intuitifs pour la majorité des utilisateurs, une interface web a été mise en place. Cette dernière possède un formulaire pour chaque requête et affiche les résultats de ces requêtes soit sous forme de tableau pour les tops, soit avec un graphique pour l'évolution des vues d'un article. La Figure 4 propose un aperçu de chacune des représentations.

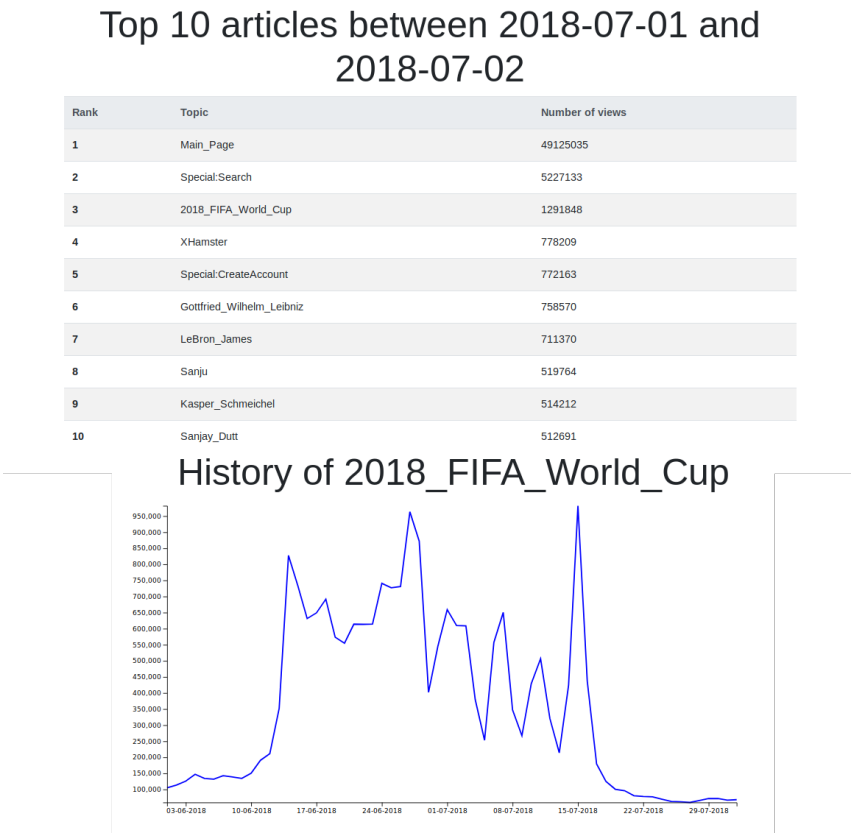

Figure 4: Aperçu du site web : en haut, un top 10 des articles sur la période du 1er au 2 juillet 2018 ; en bas, un graphique montrant l'évolution des vues sur l'article 2018_FIFA_World_Cup du 3 juin au 29 juillet 2018.

Le déploiement du serveur a été complexe puisque pour pouvoir communiquer avec les autres éléments du système, le serveur web devait être a l'intérieur du cluster et cela rendait l'accès au serveur par les utilisateurs compliqués. La solution à été de se servir du noeud Master du cluster comme proxy afin d'accéder au noeud hébergeant le serveur.

5 Cycle de vie

5.1 Problématique d'automatisation

Déployer une application distribuée peut représenter un travail conséquent. En général, ce type d'application s'appuie sur un grand nombre de composants qu'il faut configurer en permanence lorsque l'on souhaite redéployer une nouvelle version de l'application. Ce genre de travail peut représenter une perte de temps (ce serait long de se connecter à chaque machine et d'y exécuter plusieurs commandes pour l'installation) et tout faire à la main peut aussi engendrer des erreurs ; on cherche alors à automatiser ce travail.

5.2 Solution : Ansible

Nous utilisons les services d'AWS pour deployer notre application. Fort heureusement, AWS dispose d'un outil nommé boto3 permettant d'interagir avec les services Amazon depuis le langage de notre choix. Dans un premier temp, c'est cette approche qui a été choisie. Le déploiement du cluster s'effectuait alors grâce à un script Python utilisant boto3. Mais le développement par cet méthode était lent et souvent sujette aux bugs.

Nous avons donc décidé de changer de stratégie de déploiement en choisissant Ansible. Ansible est un outil de déploiement automatisé basé sur des configuration YAML. Ces configurations décrivent l'état des systèmes que l'on souhaite déployer (paquets à installer, état du système de fichier, lancement de tâches...). Ansible est compatible avec différents fournisseurs d'infrastructures Cloud, dont AWS. Il est capable de démarrer des instances EC2 AWS, de s'y connecter et d'y exécuter les commandes demandées.

```
1  ---
2  - hosts: kubernetes
3    tasks:
4      - name: Install curl
5        become: true
6        apt:
7          name: "{{ packages }}"
8          update_cache: yes
9        vars:
10         packages:
11         - apt-transport-https
12         - curl
13     - name: Run ps command
14       command: docker ps
```

Figure 5: Un exemple de script Ansible

Ansible utilise le format YAML pour décrire l'état des machines à configurer. La Figure 5 représente la structure d'un fichier YAML utilisé par Ansible. Dans l'entête du fichier, on retrouve le nom du groupe de machines sur lesquelles on souhaite exécuter le script `hosts: kubernetes` ; ici on applique le script sur les machines du groupe `kubernetes`.
Vient ensuite l'ensemble des tâches à exécuter sur ces machines dans le noeud `tasks`. Pour chaque tâche, on décrit le nom de cette tâche (utile pour le débuggage) avec l'attribut `name`. Sur l'exemple Figure 5, on présente deux types tâches différente : une installation et un lancement de commande shell. Pour une installation de paquet sur une machine, on utilise le noeud `apt` puis on spécifie la liste de packages à installer. Pour lancer une commande shell, on utilise simplement le noeud `command` suivi de la commande.

5.3 Cycle de vie de l'application

Le déploiement de notre application est subdivisé en différents scripts Ansible qui ont tous un rôle particulier, que nous allons détailler dans l'ordre de leur exécution.

create_instances.yml

Ce script est le premier à s'exécuter. Il est chargé de démarrer le bon nombre d'instances EC2 sur AWS. Il configure aussi le "security group" et le réseau des machines pour qu'elles puissent communiquer entre elles et avec l'extérieur. Pendant l'initialisation, les groupes de machines sont créés. Dans notre solution, nous distinguons deux groupes de machines : le groupe master pour la machine master et le groupe nodes pour les machines nodes.

setup.yml & master.yml & slaves.yml

Dans un premier temps, le script `setup.yml` installe les paquets communs aux deux groupes de machines (le master et les nodes), c'est-à-dire Docker et Kubernetes. Le script `master.yml` configure ensuite la machine master de Kubernetes, démarre Kubernetes sur le noeud master et rend accessible le cluster aux machines "esclaves". Le script `slaves.yml` récupère le jeton d'authentification sur le master et permet aux nodes de rejoindre le cluster Kubernetes.

mongodb.yml & kafka_ansible & website_ansible.yml

Le script `mongodb.yml` deploie un service MongoDB sur le cluster Kubernetes. Il instancie le bon nombre de réplicas et attend que tout les réplicas soient prêt pour configurer le cluster MongoDB (élection du leader, connexion des noeuds secondary au cluster).

`kafka_ansible` déploie un service Kafka sur le cluster Kubernetes. Ce script commence par instancier les pods Zookeeper (subsection 2.4) et attend que ces derniers soient opérationnels. Une fois que Zookeeper correctement lancé, les pods Kafka sont lancés.

Finalement, `website_ansible.yml` permet de déployer le serveur Apache pour le site web.

spark/setup.yml & spark/run_job.yml

`spark/setup.yml` installe Java sur le noeud master de Kubernetes et télécharge Spark sur le noeud master car Spark dispose d'un déploiement particulier. En effet, Spark créé lui même les pods pour les workers et le master Spark puis les ajoute au cluster Kubernetes. On autorise donc l'utilisateur Spark à créer des pods sur Kubernetes.

Le script `run_job.yml` permet quant à lui d'envoyer une tâche sur le cluster Spark et de l'exécuter.

web-ui/setup.yml

Ce script ajoute un service dans le cluster Kubernetes permettant de monitorer et contrôler l'état du cluster via une interface web. Ce script configure l'UI du site web et le lance dans un pod Kubernetes.

stop_instances.yml

Ce script permet de stopper toutes les instances EC2 du cluster Kubernetes afin de pouvoir détruire le cluster.

6 Problèmes techniques

6.1 Automatisation du déploiement

Coder le déploiement avec boto3 était lent car son utilisation nécessitait de développer beaucoup de choses soi-même. Il fallait, par exemple, écrire plusieurs lignes de code simplement pour récupérer l'IP d'une machine. Lors de l'installation de paquets sur une machine EC2, une erreur survenait sans que l'on puisse en déterminer l'origine. La connexion SSH sur les machines était hasardeuse : lorsqu'Amazon nous indiquait que l'instance était prête, il fallait généralement attendre un temps arbitraire pour que le serveur SSH soit lancé sur la machine distante.

C'est pourquoi nous avons décidé d'utiliser plutôt Ansible, où toutes les fonctions de gestion sont déjà codées et fonctionnent très bien. Nous avons jugé que le temps pour apprendre à l'utiliser était négligeable par rapport au temps que nous aurions pris en utilisant boto3.

6.2 Compréhension de l'écosystème Kubernetes

Il existe plusieurs version de Kubernetes et de nombreux de tutoriels sur internet tous différents ; il faut alors recouper les informations de plusieurs sources pour réussir à faire quelque chose de complexe.

Utiliser Kubernetes requiert une bonne connaissance de son fonctionnement général. Déployer un service dans le cluster nécessite de correctement configurer le cluster mais aussi les fichier décrivant la façon dont Kubernetes doit déployer ce service. Tout ceci s'effectue dans des fichiers YAML. Il existe une multitude d'attributs pour décrire le déploiement d'un service. Malheureusement pour nous, nous ne pouvons pas nous permettre de tous les connaître. Nous faisons donc généralement confiance aux sources que nous trouvons sans nous attarder sur les détails de configuration.

Cependant, nous avons parfois dû comprendre certaines parties de ces fichiers. Par exemple, nous avons passé beaucoup de temps à installer Kafka et Zookeeper sur le cluster Kubernetes à cause des "taints" et des "tolerances" qui conditionnaient le déploiement du service (pas de volume persistant disponible, pas assez de CPU disponible, ...).

A force de pratique nous avons appris de nombreuses commandes qui nous permettait de comprendre les erreurs levées lors du lancement du service (`kubectl describe pods`). Nous avons donc pu trouver son origine et régler le problème.

6.3 Communication entre les composants de l'application

Une première approche pour la mise en place du site web était d'installer directement un serveur web sur la machine master, elle n'arrivait cependant pas à accéder aux éléments du cluster Kubernetes pour y lancer les analyses. Une solution a alors été de placer le service web sur un pod directement dans le cluster. Afin de pouvoir y accéder depuis le monde extérieur, un serveur nginx sur l'instance master fait office de proxy permettant de lier le monde extérieur sur son IP publique à l'IP privée du pod sur lequel se trouve le serveur web.