Matthieu DARTOIS - 21113417

Vinh-Trung THIEU - 21415515

Rapport de projet IaC

Responsable de l'UE : Naceur MALOUCH

Professeurs : Binh-Minh BUI-XUAN, Alfred DEIVASSAGAYAME, Arthur ESCRIOU

UE : Cloud et Réseaux Virtuels

Année universitaire : 2024-2025

Sommaire

I. Structure du projet

Pour ce projet, nous avons décidé de laisser les dossiers des TME 6 et 7 afin de pouvoir observer l'évolution de notre travail, bien qu'ils ne soient pas utiles au lancement du cluster. Ainsi, dans le TME 6, nous avons manipulé Docker pour pouvoir construire des images avec les ressources fournies dans les énoncés (*node-redis* et *redis-react*). Dans le TME 7, nous avons expérimenté avec *docker-compose* pour automatiser le démarrage de conteneurs, puis nous avons appréhendé Kubernetes pour orchestrer les conteneurs. Nous avons donc pu créer nos premiers fichiers de déploiement Kubernetes.

À la racine du projet, le script Bash *project.sh* est le seul script à utiliser pour manipuler notre cluster. Le fichier *README.md* sert de guide pour son utilisation et un fichier *.state* sera généré à sa première utilisation pour garder à jour l'état du cluster et modifier le comportement du script selon l'état observé. Le répertoire *configs* est utilisé pour stocker toutes les configurations des ressources Kubernetes. Nous reviendrons plus tard sur le détail du contenu de ces fichiers. Afin que l'utilisation du script soit le plus simple possible, nous recommandons de laisser le script dans le même répertoire que le dossier *configs*. Sinon, vous avez la possibilité de donner la référence du répertoire contenant le dossier *configs* et le fichier *.state* en deuxième argument du script.

Nous utilisons un profil Minikube appelé *crv-cluster-iac* pour ne pas écraser une éventuelle configuration déjà présente. Pour ce qui est de la reproductibilité, Minikube utilise une machine virtuelle, tant que la machine hôte supporte la virtualisation et qu'elle peut allouer 6 Go de mémoire RAM et 4 cœurs de CPU au cluster, le projet devrait se lancer sans soucis.

Tout le code de notre projet est présent sur GitHub, nos images sont présentes sur DockerHub. Nous avons utilisé les sites officiels de Grafana, Prometheus, Kubernetes, Redis, mais aussi différents tutoriels que nous avons pu trouver et l'IA d'Anthropic Claude, pour comprendre les problèmes rencontrés et automatiser certains calculs pour tester notre infrastructure.

II. Composants du cluster

A. Redis-React

Le frontend React de notre cluster a été légèrement modifié afin d'être exposé grâce à un serveur Nginx plutôt que la solution native utilisant yarn ou npm. En effet Nginx permet d'exporter une image Docker plus petite et de configurer le serveur de façon plus fine. Ainsi, notre Dockerfile (dans le dossier *TME_6/node_redis*) prend en compte ce changement et le fichier de configuration *nginx.conf* est présent dans

notre dossier *TME 6*. Nous avons fait ce choix, car nos Services Kubernetes sont exposés grâce à des Ingress, il aurait été donc possible d'exposer le serveur Nginx pour servir un chemin avec un préfixe comme "*http://adresse:port/app*" et non pas "*http://adresse:port/*" par défaut.

Aussi, pour que le lancement du cluster ne dépende d'aucune adresse IP statique, nous avons également modifié la structure du frontend pour ajouter un fichier de configuration *conf.js* qui sera exporté aux clients. Ainsi, notre script de démarrage calcule l'adresse exposée du Service de l'Ingress Controller et l'injecte dans plusieurs déploiement, dont celui de *redis-react*. Dans ce cas, au démarrage du Pod, l'adresse par défaut de *conf.js* est remplacée par l'adresse du service calculée précédemment grâce au script *config-init.sh* de *redis-react*, ce qui permet à n'importe quel client de se connecter au serveur *node-redis*, tout en n'ayant pas besoin de reconstruire l'image à chaque changement d'adresse.

Le sujet du projet indique que nous ne chercherons pas à gérer la montée en charge du frontend. Nous n'avons donc pas implémenté cette fonctionnalité.

Notre image *redis-react* est disponible publiquement sur Docker Hub en tant que : *faioa/crv:tme6-redis-react-image*.

B. Node-Redis

Ce composant est le serveur qui permet au frontend de communiquer avec la base de données Redis. Comme il est sans état, il est très simple de faire la mise à échelle : nous créons un HorizontalPodAutoscaler qui permet de créer plus de Pods à la demande, selon des conditions sur la consommation des ressources matérielles par les Pods déjà créés.

Le Dockerfile de ce composant se trouve dans le dossier *TME_6/node-redis* et l'image est disponible publiquement sur DockerHub en tant que *faioa/crv:tme6-node-redis-image*.

C. Redis & Redis-Replicas

Redis est le service de base de données utilisé dans ce projet. Nous créons une base de données principale à laquelle se connecte plusieurs réplicas pour accepter plus de lecture en parallèle et décharger la base de données principale. Il est nécessaire de ne garder que les données du nœud principal puisque les nœuds secondaires ne font que recopier son contenu. Ainsi, nous utilisons un SatefulSet pour déployer le Pod de la base de données principale, puisque, comme indiqué par la documentation de Kubernetes, les StatefulSets sont utilisés à la place des ReplicaSets (=Deployments) pour gérer les applications avec état, ce qui est le cas

de Redis. Minikube gère nativement l'affectation dynamique de PersitentVolumes, nous n'avons donc qu'à préciser le champ *.spec.volumeClaimTemplates* du StatefulSet pour générer automatiquement le PersistentVolumeClaim de notre base de données.

Nous ne gérons pas la mise à échelle de notre base de données pour le moment. Ce n'était pas un objectif principal du projet et nous avons préféré nous concentrer sur d'autres fonctionnalités, comme la mise en place d'un exporter pour que Prometheus puisse récupérer les métriques de nos Pods Redis. Ainsi, nous avons utilisé l'exporter de **olivier006**. Néanmoins, comme l'image Docker de cet exporter nécessitait que l'utilisateur se connecte à Dockerhub, nous hébergeons une copie de l'exporter sur notre propre repository en tant que *faioa/crv:olivier006_redis_exporter* pour permettre de récupérer l'image sans passer par cette étape de connexion.

Nous avons fait le choix de créer un *redis-exporter* par base de données. En effet, il aurait fallu implémenter la découverte dynamique des Pods Redis, mais cela aurait été trop compliqué le temps alloué à ce projet. Au prix d'un impact plus élevé sur les performances (plus de conteneurs créés), nous avons une configuration relativement simple et utilisable telle-quelle en implémentant la mise à échelle (sous réserve de mettre en place un mécanisme de synchronisation entre les bases de données principales).

D. Prometheus

Prometheus est un outil de monitoring permettant de récupérer des données sur des cibles. Prometheus permet de configurer un sous-chemin que le serveur devra gérer. On peut donc facilement ajouter une règle d'Ingress pour sur le chemin "*http://adresse:port/prometheus*". De plus, en s'inspirant des différentes configurations de Prometheus disponibles sur Internet, nous avons défini un mécanisme de découverte des Pods concernés. Comme nous avons dédié un espace de nom aux outils de monitoring, il a fallu donner un ClusterRole à Prometheus pour donner à ses Pods l'autorisation d'inspecter les autres namespaces. Cette découverte dynamique se fait par le biais d'annotations définies dans le champ *.spec.template.metadata.annotations* des Deployments et StatefulSets. Les annotations nécessaires sont :

- *prometheus.io/scrape: true|false*
- *prometheus.io/port: port*
- *prometheus.io/path: endpoint*.

Ainsi, ces annotations sont présentes sur les Pods *node-redis*, *redis*, *redis-replica*, *node-exporter* (dont nous parlerons juste après), *ingress-nginx* (l'IngressController), et *prometheus*.

Comme Prometheus est une application avec état, nous utilisons aussi un StatefulSet. De cette manière, même si nous ne cherchons pas à mettre à l'échelle cette partie du cluster dans ce projet, il est facilement possible d'ajouter cette fonctionnalité.

E. Node-Exporter

Ce composant est maintenu par les développeurs de Prometheus et permet de récupérer des métriques sur l'utilisation des ressources matérielles d'un Node du cluster. Pour répondre à cette problématique, les DaemonSets sont nécessaires, car ils assurent que tous les nœuds du cluster ont au moins un Pod. De plus, ces Pods ont besoin d'accéder à tous les namespaces et nous avons donc besoin de créer un nouveau ClusterRole pour cela. Aussi, nous avons ajouté les annotations de Prometheus.

F. Ingress Controller & Ingress

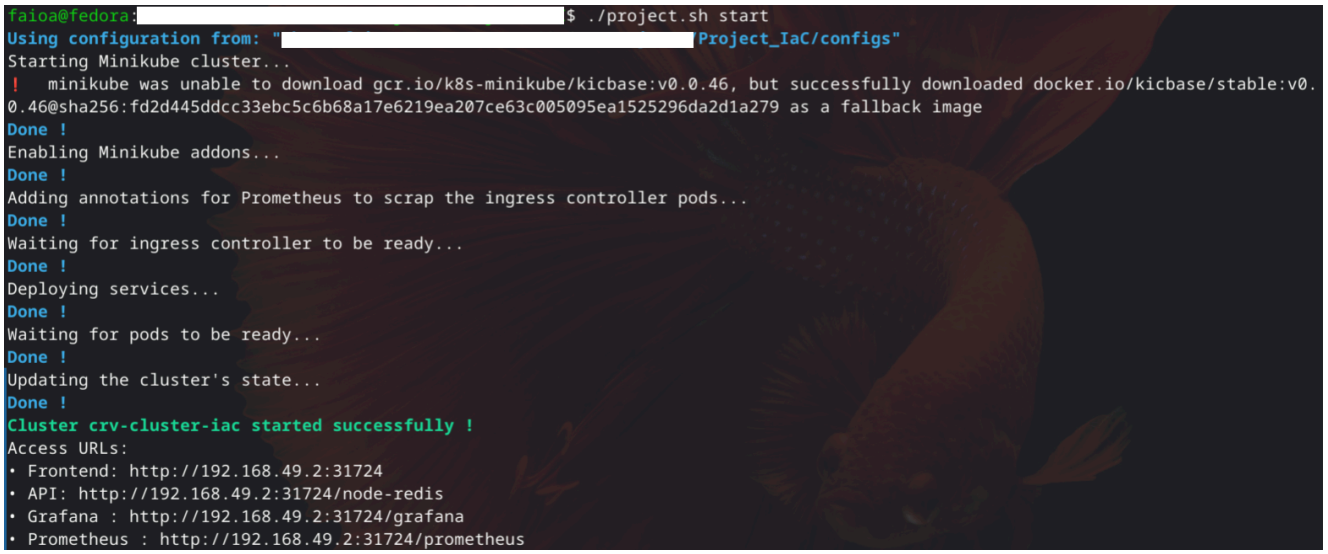
Comme nous l'avons évoqué précédemment, nous avons choisi d'utiliser un IngressController pour exposer les services de notre cluster. Minikube fournit un addon *ingress* qui permet de configurer automatiquement le contrôleur d'Ingress Nginx. Cette méthode à l'avantage d'être concise (une seule ligne de commande dans notre script), mais nous empêche de modifier facilement les déploiements. Pour exposer les métriques du contrôleur à Prometheus, nous aurions pu choisir de mettre à jour tous les Pods existant avec les annotations nécessaires, mais cela n'aurait pas été viable pour la mise à échelle des Pods. Nous avons donc fait le choix de modifier le déploiement via la commande *kubectl* pour que tous les futurs Pods soient créés avec les annotations. Il suffit maintenant de créer un autre HorizontalPodAutoscaler pour implémenter la mise à l'échelle de l'IngressController.

G. Grafana

Grafana est un autre outil de monitoring qui permet de configurer des sources de données sur lesquelles faire des requêtes afin de construire des graphiques. Nous nous sommes inspiré de la configuration d'exemple du site officiel de Grafana (Lien vers le guide) pour configurer un StatefulSet répondant à nos besoins. Ici aussi, nous ne cherchons pas à mettre cette partie du cluster à l'échelle, mais si nous devions le faire, il ne resterait qu'à mettre en place un mécanisme de cohérence des données entre les différentes instances de Grafana.

Des annotations spécifiques sont nécessaires dans l'Ingress pour laisser passer les cookies de session des clients, sans quoi seule la page de login de Grafana est accessible pour les clients, même si le message de succès est présent.

Nous n'avons pas mis de source de données par défaut pour le moment, mais nous savons qu'il est possible de rajouter cela dans la configuration de Grafana. L'utilisateur du cluster peut ajouter manuellement prometheus grâce à l'URL obtenu à la fin de la commande de démarrage, comme dans l'exemple ci-dessous.



III. Autoscaling

Pour gérer la montée en charge des Pods, nous avons utilisé les HorizontalPodAutoscalers fournis par Kubernetes. Ceux-ci permettent de créer plus de Pods lorsque ceux déjà présents sont trop sollicités. Pour faire la répartition des ressources, nous avons priorisé l'IngressController, qui est le goulot d'étranglement de notre infrastructure en permettant à l'Autoscaler de créer un second contrôleur si besoin. Aussi, les Pods *redis* et *node-redis* reçoivent plus de ressources pour assurer la disponibilité de la base de données. Un Pod de chacun de ces deux types est créé au lancement du cluster, et ce nombre peut être augmenté à 3 avec les Autoscalers. Sur les 6 Go de mémoire RAM alloués au cluster, près de 4.5 Go sont attribués à nos services et le reste est réservé pour les services du cluster.

IV. Bibliographie

Sites Internet consultés :

Anthropic. Claude AI. https://claude.ai/.

Grafana. “Deploy Grafana on Kubernetes | Grafana Documentation.” Grafana Labs, https://grafana.com/docs/grafana/latest/setup-grafana/installation/kubernetes/.

Kubernetes. “Kubernetes Documentation.” Kubernetes, https://kubernetes.io/docs/home/.

Prometheus. Configuration | Prometheus. https://prometheus.io/docs/prometheus/latest/configuration/configuration/.

Redis. Redis Configuration. https://raw.githubusercontent.com/redis/redis/7.2/redis.conf.