Rapport de Déploiement et orchestration de systèmes à grande échelle

Équipe D

Ihor Melnyk

Ayman Bassy

Tobias Bonifay

Mathieu Schalkwijk

SI5-AL

Sommaire

TD1 : Implémentation et Déploiement

Version locale

Développement en local du serveur en python (Flask) et intégration avec Redis et Postgres dans un docker-compose dans un premier temps.

Déploiement kube

Création des fichiers à plat (.yml) pour le déploiement et service. Service déployé est de type Load Balancer pour pouvoir y accéder depuis l'extérieur grâce à une IP publique attribuée par OVH à ce LB.

Déploiement de la base de données Postgresql et Redis à l'aide de Helm dans le but de simplifier le déploiement. A la création de ces bases de données on spécifie à helm un fichier qui contient des valeurs customisées pour personnaliser la configuration de la base de données, par exemple on veut telle table accessible par tel utilisateur etc.

Nous avons regardé comment marche Kompose mais nous avons décidé de ne pas utiliser pour mieux comprendre et maîtriser les fichiers de déclarations yml à plat.

Ansible

Nous avons également écrit des playbooks ansible utilisant des modules k8s et helm pour automatiser la mise en place de notre application dans un cluster kube, par exemple si on est amené à changer de provider cloud et qu'on a un nouveau cluster il suffirait de lancer le playbook et les déploiements, services et bases de données seraient en place en quelques minutes seulement, presque aucune commande lancée à la main.

Ansible étant un très puissant langage déclaratif, on écrit comment doit être l'état final et lui va faire le nécessaire pour y arriver. Sa force est en idempotence, c'est-à-dire si l'état souhaité est déjà atteint il ne va rien faire au système et passer à la tâche suivante. De plus avec les conditions qu'on peut ajouter sur certaines actions on est capable de faire énormément de choses.

Difficultées rencontrées

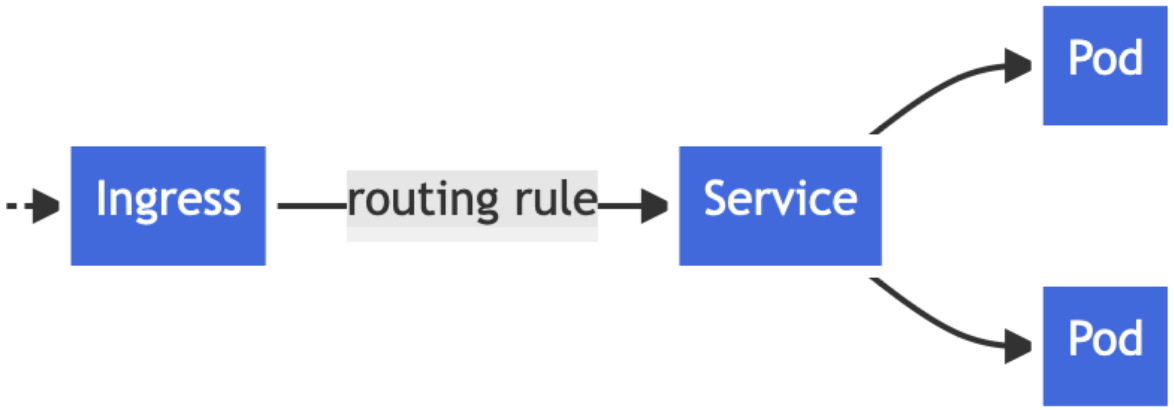
Temps pour implémenter le serveur bien supérieur à celui prévu.

Un problème a été rencontré avec Postgres, les règles de connexion ne laissaient passer aucune connexion. La solution était donc d'autoriser la connexion depuis n'importe quelle source car dans le cadre de notre projet il nous fallait absolument une solution rapide et fonctionnelle. La bonne solution serait de jouer encore avec les réglages de la base de données pour trouver la cause de ce bug.

TD2 : Ingress, GitOps et ArgoCD

Ingress

Ingress nous permet d'utiliser une unique adresse IP pour accéder à nos services de l'extérieur du cluster. Ingress en soi est une règle de routage qui nécessite un Ingress Controller pour être appliquée. Dans le cas où nous avons des services dans des namespaces différents et qu'on souhaite les rendre accessibles de l'extérieur, on va devoir créer un Ingress par namespace. Grâce au nom de domain (<prefix>.orch-team-d.pns-projects.fr.eu.org) on va pouvoir exposer autant de services qu'on le souhaite en donnant un <prefix> différent à chaque service.



Terraform

Terraform est un outil qui permet de provisionner les machines en communiquant avec le cloud provider et également de configurer les machines mais très souvent utilisé plutôt pour le provisionnement. Du fait qu'il doit communiquer avec un Cloud Provider le code va être dépendant également de ce que le Provider demande. La courbe d'apprentissage est assez raide.

Nous avons écrit un fichier terraform qui déploie uniquement Polymetrie avec le Service associé et ça a pris plus de 60 lignes alors que les mêmes actions écrites avec Ansible prennent moins de 20 lignes. Donc on a préféré utiliser Ansible de plus que dans le cadre de notre projet on est pas amenés à communiquer avec le Cloud Provider pour provisionner des nouvelles machines.

ArgoCD

ArgoCD nous permet d'automatiser le déploiement et la mise à jour pour réduire le besoin d'interventions manuelles et les erreurs potentielles. Il utilise Git comme source de vérité, ce qui facilite le suivi des modifications et la collaboration. Il nous offre aussi une visibilité en temps réel sur l'état des applications et des environnements.

Pour déployer ArgoCD dans notre environnement Kubernetes, nous avons provisionné l'url de notre repository github dans le fichier "`application.yaml`" et spécifié qu'il devait

surveiller les changements sur “head”. Nous avons aussi précisé les options “prune” et “selfheal” pour réconcilier les ressources entre le repository et notre cluster.

Pour qu’ArgoCD se connecte au repository Git, nous avons utilisé l’interface d’ArgoCD. A terme, il serait préférable d’utiliser un token GitHub dans un secret kubernetes.

Nous avons laissé par défaut le polling régulier (toutes les trois minutes) vers le repository GitHub. Cette méthode nous convient, car nous n'avons pas besoin de réactivité immédiate pour appliquer les changements de configuration. Un des inconvénients de la méthode est qu’elle nous oblige à attendre quelques minutes après un push pour voir nos modifications appliquées, ce qui peut être embêtant quand on enchaîne des changements de configurations. Pour éviter ce problème, nous pourrions créer un namespace non surveillé par argocd qui nous servirait à appliquer nos changements à la main, puis reporter les changements sur le code déployer en cas de succès.

Nous avons décidé de n’utiliser qu’un seul repository pour les sources de l’application et la configuration ArgoCD, en précisant simplement à ArgoCD le répertoire qu’il doit surveiller dans le repository (dans le fichier “`application.yaml`”).

Problème d’argoCD

Toutefois, ArgoCD pose certaines difficultés dans le cadre de test rapide de changement de configuration des manifestes qui lui sont liés. Typiquement pour notre déploiement polymétrie, ou dans le cadre d’un souci avec HPA, nous avons dû “jouer” avec les configurations du déploiement pour investiguer, mais nous étions bloqués par ArgoCD.

Une solution serait d’avoir un autre namespace de test où l’on travaillerait, et quand seulement c’est bon, alors l’on pourrait commit les changements sur le repository et donc voir les modifications apparaître sur le bon namespace.

Pipeline GitHub Workflow

De base, ArgoCD ne déployait pas l’application Python lors d’un push un nouveau changement car le tag de l’image de Docker Hub (“latest”) n’était pas modifié dans les configurations. Nous avons donc implémenté un système de version d’image avec un workflow GitHub.

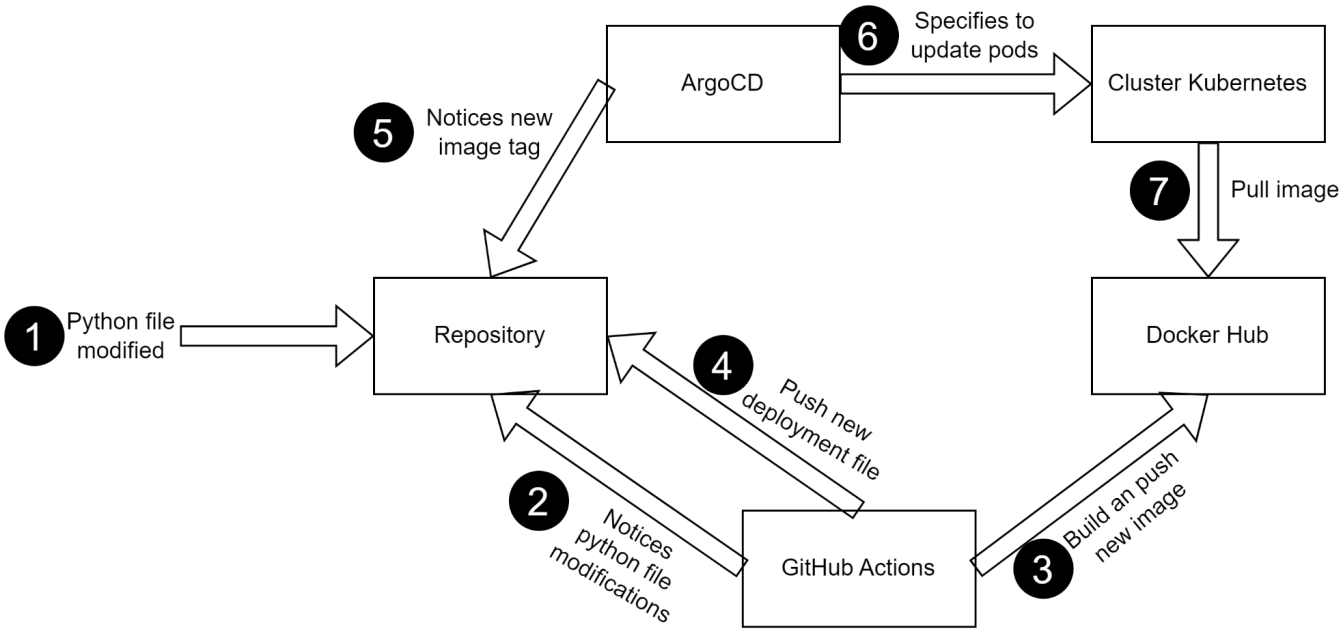
Le pipeline de workflow GitHub, configuré dans notre projet, automatise le processus de construction et de déploiement d'images Docker pour notre application Python. Il y a des déclenchements sur les 'push' de la branche 'master', lors de la modification des fichiers main.py ou Dockerfile. Le but est d’auto-incrémenter la version de l'image Docker dans un fichier YAML avec le commit automatique des changements.

Nous construisons et déployons automatiquement l’image Docker: Docker Buildx pour la construction de l'image et avec gestion du cache pour optimiser le processus de construction et connexion à Docker Hub et déploiement de l'image Docker avec la nouvelle version en utilisation les secrets githubs.

Ce pipeline assure une mise à jour continue et automatisée de l'image Docker, synchronisant ainsi le déploiement avec les dernières modifications du code source.

Déroulement de la pipeline lors d'une modification du code

1. Push d'une modification du code Python
2. Détection de la modification du code Python par GitHub
3. Build de l'image et push sur Docker Hub en incrémentant la version
4. Push sur le repository du nouveau numéro de version de l'image (dans le fichier "`polymetrie_deployment.yml`")
5. ArgoCD détecte le changement de version
6. L'opérateur ArgoCD commande d'appliquer la nouvelle version de l'image Docker pour le Déploiement
7. Les pods sont redémarrés avec la nouvelle version de l'image



TD 3 : Prometheus et Grafana

Prometheus

Prometheus est utilisé dans un contexte Kubernetes afin de pouvoir récupérer des informations de monitoring sur des pods via des services ou non, via ce que l'on appelle les éléments de monitoring, à savoir dans notre contexte, le "ServiceMonitor".

En effet, dans un contexte de déploiement à haute échelle, avec parfois un très grand nombre de nœuds et pods, il faut pouvoir surveiller notre système en temps réel, afin de pouvoir rapidement détecter les zones de panne dans un environnement donné, qui dans le pire des cas pourrait être celui de production.

Dans notre cas, nous avons mis en place la stack prometheus (incluant Grafana) dans le but d'avoir une vue élargie et centralisée de notre cluster, et pouvoir surveiller notre cluster en temps réel.

Prometheus n'est pas la seule solution existante, en effet d'autres solutions plus performantes et mieux sécurisées ont vu le jour, comme Datadog. Mais prometheus est un projet open-source mis à jour régulièrement, sur lequel l'on peut ajouter au besoin des spécificités via des contributions sur le projet.

Grafana

Grafana est le service UI qui vient se lier à Prometheus afin de réaliser l'affichage des informations prélevées par Prometheus. Il suffira de déclarer à Grafana la data source qui sera utilisée, dans notre cas prometheus, afin qu'il récupère les données de metrics de prometheus, à afficher au travers de dashboard.

Les dashboards sont le moyen d'afficher et de rendre plus expressifs un ensemble de metrics, afin que celle-ci représente quelque chose, et puisse rapidement être analysé dans un objectif de monitoring.

L'api prometheus permet dans le langage de développement utilisé, d'envoyer des metrics plus personnalisé, qui ensuite, avec le langage de requête promQL (propre a la data source Prometheus), permet la récupération de ces metrics.

Dans notre cas d'utilisation, Grafana nous a permis de récupérer dans deux dashboard différent, respectivement les données de metrics infrastructurel de certains namespace de notre cluster, et de récupérer les données de metrics "business" correspondant au nombre de vues par site.

Fonctionnement et Déploiement

Le projet kube-prometheus-stack nous met à disposition un chart helm très complet (de tout de même plus de 10 000 lignes), qui permet de faire l'installation de la stack prometheus entièrement. En effet, il a fallu que l'on implémente à côté un fichier de Value qui surcharge les définitions par défaut, mais avec ce chart, nous avons déployé l'ensemble de la stack, qui englobe (en plus des services **ClusterIp** pour chacun) :

- **L'Opérateur Prometheus :** Permet d'ajouter les ressources customisées de Prometheus. (Monitoring.coreos.com)
- **Prometheus StatefulSet :** Permet d'avoir l'endroit où seront stockés les metrics récupérées par prometheus, et donc la "data source" prise par Grafana.
- **Prometheus Node Exporter DaemonSet :** Permet de déployer sur chaque nœud, un node exporter qui va récupérer les metrics spécifique de nos nœuds
- **Prometheus State Metrics ReplicaSet :** Permet d'avoir au sein du cluster une unité de scraping de metrics des objets Kubernetes.
- **Grafana :** L'instance Grafana
- **Ingress Grafana et Prometheus** : Point d'entrée depuis l'extérieur respectivement Grafana et Prometheus.

La stack fait aussi le déploiement de service monitor par défaut, afin de monitorer certains composants du cluster.

Les services Monitor sont ce qui va permettre à Prometheus de détecter les objets Kubernetes, afin qu'il aille scraper les données.

Afin que l'on puisse exécuter notre objectif business, il fallait que l'on puisse récupérer les metrics exposée sur le /metrics. Il a donc fallu que l'on déploie notre propre serviceMonitor afin que Prometheus sache où est notre endpoint afin de récupérer nos données. Par la suite, via Grafana, l'on pouvait les afficher sur les dashBoards.

Difficulté globale

Prometheus - ServiceMonitor

Nous avions eu un gros problème pour permettre à Prometheus de récupérer nos metrics exposées depuis notre pod polymétrie.

En effet, l'erreur très précisément a été que dans le service monitor, il fallait indiquer le label du déploiement lui-même, déclare dans la partie metadata, et non le label déclare dans la spec de celui-ci. Ni les labels que l'on retrouve effectivement dans le service et le pods lui-même. Ce qui est en soit logique étant donné que le déploiement gère le déploiement d'une ou plusieurs instances.

Ce problème a été très long et compliqué à débugger, en effet, nous étions toujours induits en erreur par prometheus à cause de sa configuration par défaut, ou son service discovery faisait qu'il arrivait à détecté via le service monitor le service, mais n'arrivait à y accéder. En effet, cela a enlevé le soupçon sur le service monitor, ce qui m'a pousser la recherche du côté de Prometheus, qui n'était du coup pas la source du problème.

Le fait que nous n'ayons pas déclaré de label dans les metadata du déploiement, étant donné que nous n'avions pas encore les compétences au début du projet nous a trahis.

Grafana

Grafana est très complet et assez complexe, mais grâce à la ressource disponible en ligne, nous avons pu trouver comment réaliser un affichage de nos données business. Cependant, nous avons eu des difficultés à rendre le processus générique.

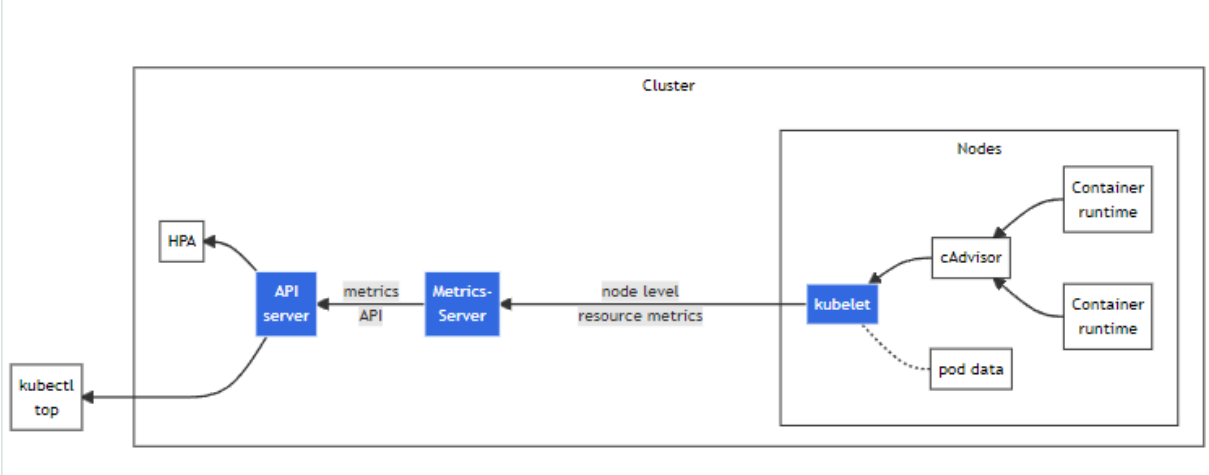
En effet, à chaque fois que l'on ajoute un nouveau site comme "membre", l'on commence a accepté de compter le nombre de vues qu'il reçoit. Mais pour afficher ce nouveau site et les nouvelles données reçues, nous devons à chaque fois à la main créer une nouvelle metrics, qui sera récupéré par prometheus. Nous voulions automatiser ce processus, afin qu'à chaque ajout de site partenaire, l'on ait une nouvelle métrique qui apparaît, mais sans avoir à modifier le code. Nous n'avons pas réussi à aller au bout de cela étant plus compliqué à réaliser que prévu.

TD 4 : HPA et tests de charge

HorizontalPodAutoscaler

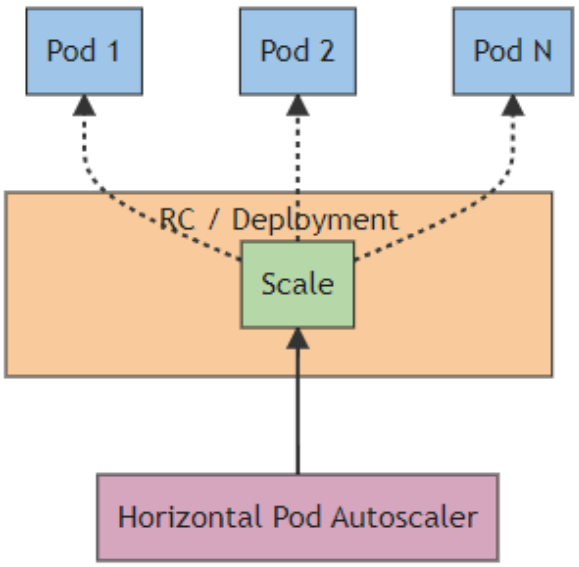
L'Horizontal Pod Autoscaler est une instance de pods déployable directement via l'Api de Kubernetes. Cela déploie un pods qui va avoir la main sur notre déploiement, et va gérer le scaling horizontal de celui-ci en fonction de la ressource consommée par le pods issue du déploiement. Pour ce faire, le HPA va utiliser ce que l'on appelle le "Metric-server".

Le Metric-server est une instance déployée dans un pod au niveau du nœud maître. En effet, celui-ci va récupérer en temps réel des métriques liées aux ressources consommées par les nœuds via leurs kubelets. Grâce à cela, l'API server de Kubernetes est informé en temps réel de ces informations de ressources, qui sont persistées dans l'etcd.



Dans notre cas, on voit que le metrics-server est déjà déployé par OVHcloud, dans le node master, dans le namespace de kube-system.

Le HPA devine la consommation de ressource de notre pods via une requête envoyée vers l'api server de notre cluster (qui elle connaît les ressources de chaque pods via le metrics server qui aggregate les valeurs de metrics issue d'un même nœud).



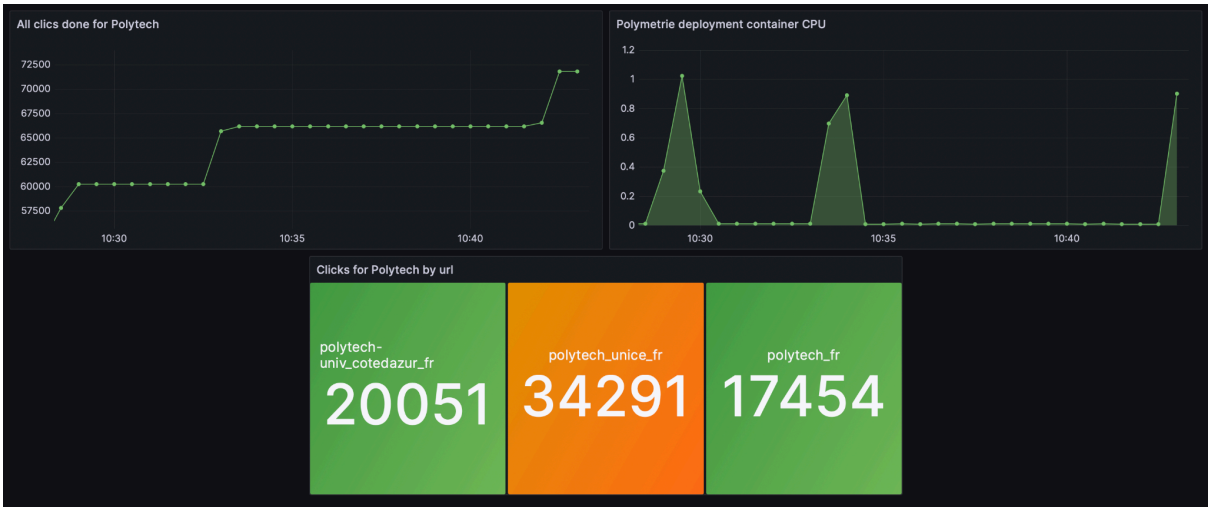
Problématique liée au HPA

Nous avons des problèmes au niveau du déploiement du HPA la première fois. En effet, le HPA impose que le manifeste de déploiement, parmi les specs déclare le minimum de ressources qui sera utilisé. Donc via “ressources.requests.cpu”. Cette étape est obligatoire, car elle oblige nos pods à finalement utiliser une quantité de CPU, et donc permet au HPA d’avoir matière à comparer pour décider de quand scalée le nombre de pods.

Par la suite, l’on précise au HPA via le “scaleTargetRef”, le nom du déploiement auquel l'ont fait allusion, et le HPA s’occupera donc de gérer de manière dynamique le nombre de ressources utiles.

Dashboard

Notre dashboard avec custom metrics de Polymetrie. On affiche l’évolution du nombre total des clics fait pour le client Polytech, les compteurs par url du Polytech et finalement la charge CPU sur les Pods de Polymetrie. Cela permet d’envoyer par exemple un certain nombre de requêtes et ensuite voir combien d’entre elles ont bien été traitées et combien de CPU ça a sollicité.



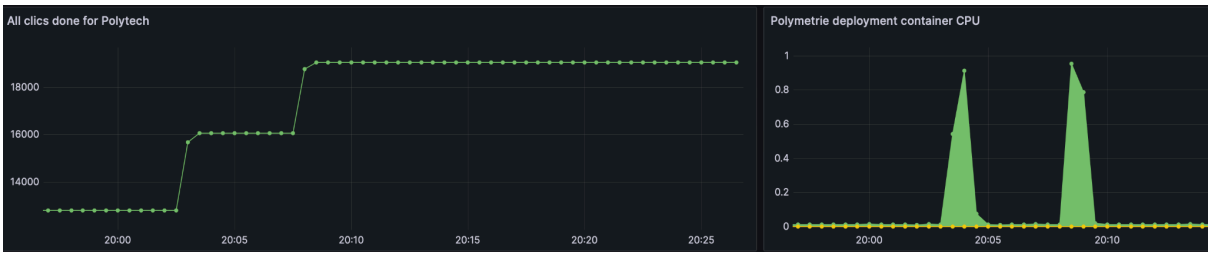
K6

Installation avec le git du K6 (git clone -> make deploy) qui va installer l’opérateur dans un namespace dédié. Cet opérateur entre autres va gérer la création des ressources du type K6.

Pour lancer un test de charge il faut créer une ConfigMap qui contiendra le script .js (dans la limite de 1 Mo) ensuite créer une ressource de type K6. Une fois le script exécuté, il faut supprimer cette ressource. Pour automatiser ce processus un script **run-test.sh** a été créé.

Il suffit de créer un script .js et de le lancer avec ./run-test.sh <test-script-name>.js

Campagne de tests



CPU limité à 1 cœur, deux passages avec 1000 requêtes simultanés toutes les secondes pendant 10s. **Résultat attendu:** compteur incrémenté de 10 000 par chaque passage. **Résultat obtenu:** le premier passage a incrementé le compteur de 3300 environ et le second de 3000 environ. **30% du succès**. La mémoire n'a pas subi de pression, niveau normal aux alentours de 300 Mo pendant le test.

Même test mais cette fois avec seulement 100 requêtes simultanés. **Résultat:** 100% des requêtes sont passées, le compteur a bien pris 1000 clics en plus.

Objectif : Trouver la limite d'un Pod qui a exactement un cœur.

CPU limité à 1 cœur par pod. **Replica est à 1**, donc un seul Pod traite les requêtes.

Requêtes (nb sur 10s)	Taux de réussite (% de requêtes traités)	CPU	Nombre Pods
2500	94,08%	1 cœur	1
2000	99,2%	1 cœur	1
1800	100%	1 cœur	1
5000	78,6%	1 cœur	2
2500	94,6%	1 cœur	2

Maintenant on va augmenter le nombre de **réplicas à 2** et voir si on arrive à traiter 2500 requêtes avec un taux de réussite à 100%.

Requêtes (nb sur 10s)	Taux de réussite (% de requêtes traités)

5000	78,6%
2500	94,6%

Le fait d'avoir 2 instances de polymétrie n'a pas amélioré significativement les performances, mais a tout de même réduit la charge sur une seule instance; Donc probablement le problème est ailleurs. **(le throttling ?) (le nœud surchargé?)**

En passant à une limite de **1,5 CPU** par pod et **2 instances** de Polymetrie

Requêtes (nb sur 10s)	Taux de réussite (% de requêtes traités)
2500	100%

Conclusion intermédiaire : **scaling horizontal inefficace**

Mauvaise pratique de mettre des limites CPU par Pod !

No limits CPU

Requêtes nb sur **30s** (rate)	Taux de réussite (% de requêtes traités)	CPU used	Nombre Pod
5400 (rate 180/s)	98.5%	1.08	1
5400 (rate 180/s)	95.2%	0.8 et 0.6	2
5700 (rate 190/s)	98.6%	0.9	1
6000 (rate 200/s)	98,8%	0.9	1
9000 (rate 300/s)	91.9%	1.57	1
9000 (rate 300/s)	96,5%	1.17 et 0.86	2

45 000 (rate 1500/s)	20.4%	1.78	1
45 000 (rate 1500/s)	22.6%	1.58 et 1.23	2

On observe une différence un peu plus visible au niveau des performances entre 1 instance et 2 instances en augmentant fortement la charge. Mais peut être la durée de test est trop courte donc il faut augmenter la durée de notre test.

Un test sur 10 minutes :

Requêtes	Taux de réussite (% de requêtes traités)	CPU used	Nombre Pod
300 000 (500/s)	54,7%	4.6	1
300 000 (500/s)	61.6%	2.5 et 1.35	2

On observe en effet que sur un temps plus long 2 instances arrivent à traiter un plus grand nombre de requêtes qu'une seule instance. L'écart certes est plus grand avec le test sur 30s mais tout de même n'est pas significatif.

Conclusion

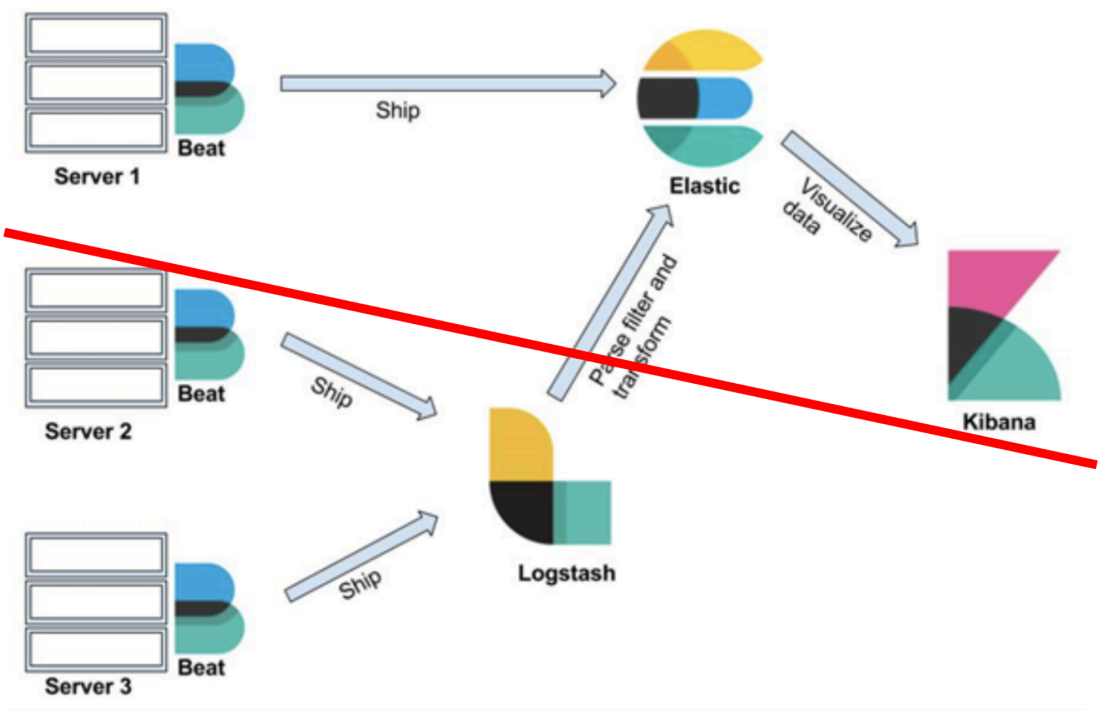
Les tests nous ont permis de déterminer les limites de l'actuelle architecture et pointer le doigt vers le problème avec le scaling. Le scaling horizontal n'apporte pas malheureusement des améliorations significatives dans notre cas. Actuellement on peut affirmer que l'application peut traiter de manière sûre jusqu'à **170 clics par seconde**.

Difficultées

La syntaxe des métriques que le programme python doit exposer est assez stricte et peut poser problème lorsque nous voulons nommer des métriques avec une url. Les résultats de tests de charge n'étaient pas en cohérence avec nos attentes.

TD5 : ELK, Gestion des Logs

	Descriptif	Context utilisation ELK de log
Elasticsearch	Base de données avec moteur de recherche intégré. Capable d'ingestion d'un très grand volume de données.	Utiliser pour stocker les données de log, et permet de rapidement récupérer les logs.
Logstash	Instance utilisée dans la récolte de données, et de traitement de celui-ci dans un pipeline configurable.	Utiliser pour recevoir, agréger et envoyer des logs vers Elasticsearch. Peut aussi aller les récupérer lui-même dans le volume d'un pods.
Kibana	Kibana est une interface de visualisation des données qui s'intègre avec Elasticsearch.	Offre une interface de visualisation des logs.
FileBeat	Instance basée sur le framework Beat (a l'image de MetricBeat ou autre…). Utilisé dans la récolte de données uniquement, il remplace à son apparition Logstash, étant plus léger que ce dernier.	Permet de faire de la récupération de données stockées dans des volumes précis, de manière très récurrente.



Ci-dessus, l'on a la stack au complet, en bas du trait rouge la version avec Logstash qui effectue des opérations sur les éléments. En haut la version que nous avons déployée, avec FileBeat qui envoie directement les logs vers Elasticsearch.

Elastic Search

Lors du premier déploiement d'Elastic Search, nous avons dû apporter des modifications à un paramètre du noyau Linux "**vm.max_map_count".** Celui-ci définit le nombre maximal d'adresses virtuelles pouvant être mappées dans l'espace d'adressage d'un processus.

Etant donnée qu'elasticsearch gère à la fois l'index et le stockage des données (document), alors il faut configurer le noyaux pour qu'il permettent à elasticsearch d'allouer plus d'adresse mémoire.
Puis nous avons vu qu'il existait différents types de nœuds dans le cluster. Qui sont :

- **Data node** : stock et index de la donnée localement
- **Master node** : gère l'état de santé du cluster
- **Warm Data node** : inclut une fonctionnalité qui maintient une donnée longtemps
- **Hot Data node** : maintient un ensemble de données à chaud pour répondre à un fort volume de requête.
- **Ingest node** : permet de configurer un pipeline de préprocesseur de la donnée.

Dans le cadre du déploiement d'Elastic Search, nous avions un souci de CPU, en effet, lors du déploiement via le manifeste fourni dans le tutoriel, nous avions atteint la limite possible de CPU. Au début, nous n'avions pas compris le problème, mais très vite en voyant la barre de CPU au max sur "OpenLens", nous avons vite compris.

En effet, parmi le nœud, certains nécessitent plus de CPU que nous pouvons gérer pour leur bon fonctionnement, surtout le node Ingest, souvent couplé à une fonctionnalité de stockage. L'ingest utilise la définition d'expression régulière donnée dans la configuration afin de faire un traitement sur les données avant qu'elle soit renvoyée (et partitionnée) vers les nœuds de stockage.

Finalement, la solution a donc été de faire un nœud qui inclut tout. En effet, on peut se permettre d'avoir un seul nœud qui fasse du stockage, indexage, gère son état de santé et fasse de l'ingest, car la taille de notre cluster et donc le nombre de logs n'est pas si grand.

Mais dans un contexte plus conséquent, il aurait fallu faire les bonnes séparations. Une bonne pratique dans le contexte de création de cluster elastic et typiquement d'avoir des nœuds dédiés à l'ingest et rien d'autre. En effet, quand la requête rentre dans le cluster, dans tous les cas, elle sera traitée par le nœud d'ingest, même si elle est prise en charge par un autre nœud qui ne l'est pas, il redirigera la donnée vers un nœud d'ingest. Cette séparation est faite, car avoir un même nœud d'ingest et de stockage peut être à risque, en effet, nous ne sommes pas à l'abri d'une surconsommation de CPU.

FileBeat & Kibana

FileBeat est déployé au travers d'un DaemonSet, ce qui permet d'avoir une instance de récupération de log en temps réel, est ce dans chacun de nos noeud.

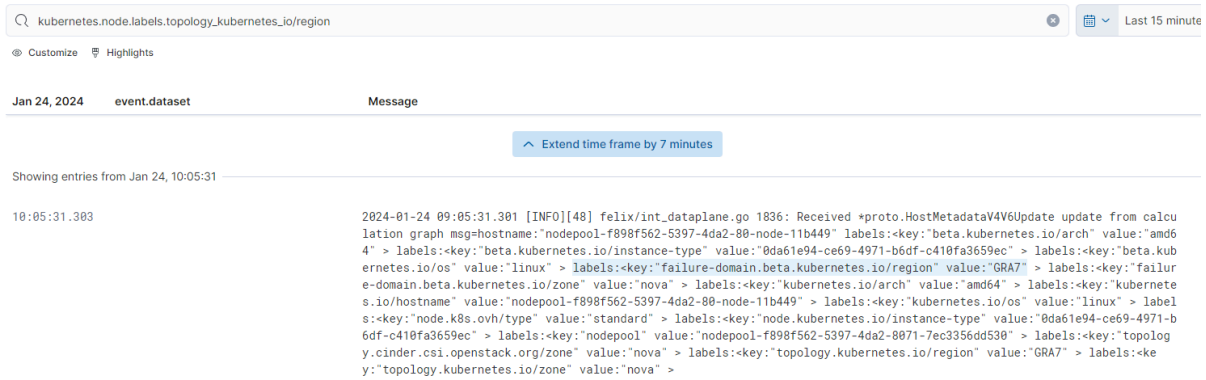
Au déploiement de celui-ci, nous lui indiquons le volume dans lequel il ira chercher les valeurs, et dans les configMap, nous lui passons le label du service d'elastic search.

Kibana quant à lui est configuré pour être lier au cluster elastic, et permet donc de visualiser les logs en temps réel, et de réalisé des dashboards, plus ou moins dynamique avec canvas.

Mise en contexte

Dans notre contexte, nous avons mis en place cette stack afin de traquer en temps réel depuis une interface plus agréable, les logs de notre application polymérie. Nous avons par exemple pu avoir les logs en temps réel de notre pod lorsque, nous lui envoyons des requêtes d'url.

Nous avons aussi pu voir la région dans lequel nos nœuds sont présents :



Il s'agit de "GRA7" :

Annexe (Credential)

Grafana & Prometheus

http://grafana.orch-team-d.pns-projects.fr.eu.org/
http://prometheus.orch-team-d.pns-projects.fr.eu.org/
Credential : admin - icorp-92i

ArgoCD

http://argocd.orch-team-d.pns-projects.fr.eu.org/
Credential : admin - f3iUT3vkPZiWz9HO (secret name : argocd-initial-admin-secret)

ELK

https://kibana.orch-team-d.pns-projects.fr.eu.org/
https://elastic.orch-team-d.pns-projects.fr.eu.org/
Credential : elastic - 76Kf07IG28ae29VGL6GW5TPg (secret name : elastic-cluster-es-elastic-user)