

Déploiement et orchestration de systèmes à grande échelle



Team G

Al Achkar Badr
Ben Aissa Nadim
Gazze Sourour
Yahiaoui Imène
Zoubair Hamza

Sommaire

TP 1 :	2
A. Une version locale en docker-compose	2
B. En route vers Kube !	2
1. Configuration de l'environnement de travail :	2
2. Déploiement du kubernetes Dashboard :	5
3. Déploiement de l'application sur Kubernetes :	6
4. Les problèmes rencontrés	9
C. Industrialisation avec Ansible	9
TP 2	11
A. Terraform	11
B. Ingress controller	16
C. Argo CD	17
TP3:	23
A. Kube-Prom-Stack	23
B. Un peu de Graphes	28
Métriques et Visualisation : Dashboard Grafana de Polymetrie	31
Métriques système :	31
Métriques métier :	32
TP 4	33
A. HPA	33
1. Explication du mécanisme HPA :	33
2. Implémentation du HPA :	33
3. Les problèmes rencontrés	35
B. Évaluation des ressources	35
TD5	38
A. Exploration du stack ELK	38
1. Installation	38
2. Explication	38
3. Les problèmes rencontrés	40
B. Comparaison Splunk VS Kibana	41
Bibliographie	42

TP 1 :

A. Une version locale en docker-compose

B. En route vers Kube !

1. Configuration de l'environnement de travail :

- **Quelles informations pouvez-vous déterminer avec la commande "kubectl cluster-info" ?**

La commande **kubectl cluster-info** fournit des informations générales sur l'état du cluster Kubernetes auquel on est connecté.

```
$ kubectl cluster-info
Kubernetes control plane is running at https://q9nkqe.c1.gra7.k8s.ovh.net
CoreDNS is running at https://q9nkqe.c1.gra7.k8s.ovh.net/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://q9nkqe.c1.gra7.k8s.ovh.net/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

- **Kubernetes control plane is running at [URL]:** Cela indique l'adresse principale du control plane de votre cluster Kubernetes. Le control plane est constitué de composants tels que l'API Server, le Controller Manager, le Scheduler, et d'autres.
- **CoreDNS is running at [URL]:** Cela indique l'URL où CoreDNS est accessible. CoreDNS est utilisé pour les services DNS dans le cluster.
- **Metrics-server is running at [URL]:** Cela indique l'URL où le serveur de métriques est accessible. Le serveur de métriques est utilisé pour collecter des métriques de ressources et d'utilisation dans le cluster.

a. Kubernetes control plane :

Lorsqu' on déploie Kubernetes, on obtient un cluster qui se compose d'un ensemble de machines de travail, appelées nœuds, qui exécutent des applications conteneurisées. Chaque cluster possède au moins un nœud de travail.

Le(s) nœud(s) worker(s) hébergent les Pods qui sont les composants de la charge de travail de l'application. Le plan de contrôle gère les nœuds de travail et les pods dans la grappe. Dans les environnements de production, le plan de contrôle s'exécute généralement sur plusieurs ordinateurs et une grappe s'exécute généralement sur plusieurs nœuds, ce qui assure la tolérance aux pannes et la haute disponibilité.

b. CoreDNS :

CoreDNS est un serveur DNS flexible et extensible qui peut servir de DNS pour le cluster Kubernetes. Comme Kubernetes, le projet CoreDNS est hébergé par la CNCF. Kubernetes crée des enregistrements DNS pour les services et les pods.

Kubernetes publie des informations sur les Pods et les Services qui sont utilisés pour programmer le DNS. Kubelet configure le DNS des Pods afin que les conteneurs en cours d'exécution puissent rechercher les services par leur nom plutôt que par leur adresse IP. Les services définis dans le cluster se voient attribuer des noms DNS. Par défaut, la liste de

recherche DNS d'un Pod client comprend l'espace de noms propre au Pod et le domaine par défaut du cluster.

c. Metrics-Server:

Kubernetes Metrics Server est un outil de surveillance des métriques de ressources pour Kubernetes. Le Kubernetes Metrics Server mesure l'utilisation du CPU et de la mémoire dans le cluster Kubernetes. Ce serveur de métriques collecte les métriques et affiche ensuite les statistiques d'utilisation des ressources pour votre cluster (nodes, pods, etc).

- **De combien de workers est constitué le cluster livré ?**

Pour le nombre de noeuds workers : `kubectl get nodes -o wide`

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	CONTAINER-RUNTIME
nodepool1-90179ca9-c791-4b38-ad-node-169897	Ready	<none>	29d	v1.27.4	162.19.51.195	<none>	Ubuntu 22.04.3 LTS	5.15.0-88-generic	containerd://1.6.20
nodepool1-90179ca9-c791-4b38-ad-node-e94563	Ready	<none>	29d	v1.27.4	162.19.53.180	<none>	Ubuntu 22.04.3 LTS	5.15.0-88-generic	containerd://1.6.20
nodepool1-90179ca9-c791-4b38-ad-node-ff5af3	Ready	<none>	29d	v1.27.4	162.19.54.120	<none>	Ubuntu 22.04.3 LTS	5.15.0-88-generic	containerd://1.6.20

Notre cluster est constitué de 3 nœuds workers.

Chaque nœud est décrit avec des détails tels que l'état, les rôles, l'âge, la version de Kubernetes, l'adresse IP interne, l'adresse IP externe (le cas échéant), le système d'exploitation, la version du noyau, et le runtime de conteneur utilisé.

- **Quelles sont les ressources disponibles (CPU/RAM) ?**

Nous pouvons visualiser l'utilisation des ressources à l'aide des commandes suivantes : `kubectl top pods` et `kubectl top nodes`

Chaque nœud a une capacité totale de 2 unités de CPU, 6926356 kilo-octets de mémoire RAM, et 50620216 kilo-octets de mémoire ROM. Les ressources actuellement disponibles pour l'allocation sont de 1840 milli-unités de CPU et 5275668 kilo-octets de mémoire RAM.

```
$ kubectl describe node nodepool1-90179ca9-c791-4b38-ad-node-169897 | grep -E -A 5 'Capacity|Allocatable'
```

Capacity:	
cpu:	2
ephemeral-storage:	50620216Ki
hugepages-2Mi:	0
memory:	6926356Ki
pods:	110
Allocatable:	
cpu:	1840m
ephemeral-storage:	33387320Ki
hugepages-2Mi:	0
memory:	5275668Ki
pods:	110

```
$ kubectl describe node nodepool1-90179ca9-c791-4b38-ad-node-e94563 | grep -E -A 5 'Capacity|Allocatable'
```

Capacity:	
cpu:	2
ephemeral-storage:	50620216Ki
hugepages-2Mi:	0
memory:	6926344Ki
pods:	110
Allocatable:	
cpu:	1840m
ephemeral-storage:	33387320Ki
hugepages-2Mi:	0
memory:	5275656Ki
pods:	110

- **Quels sont les namespaces déjà créés ?**

```
$ kubectl get namespaces -A
NAME                STATUS    AGE
default             Active    2d1h
kube-node-lease     Active    2d1h
kube-public          Active    2d1h
kube-system          Active    2d1h
```

Les namespaces existants sont : **default**, **kube-node-lease**, **kube-public**, et **kube-system**.

- **default** : c'est le namespace par défaut, si on crée une ressource sans préciser le namespace elle est déployée dans default.
- **kube-node-lease** : Cet espace de noms contient les objets Lease associés à chaque nœud. Les baux de nœuds permettent au kubelet d'envoyer des battements de cœur afin que le plan de contrôle puisse détecter la défaillance d'un nœud.
- **kube-public** : Les ConfigMaps de Kubernetes sont par espace de noms et ne sont généralement visibles que par les mandants qui ont un accès en lecture à cet espace de noms. Pour créer une carte de configuration que tout le monde peut voir, nous introduisons un nouvel espace de noms kube-public. Cet espace de noms, par convention, est lisible par tous les utilisateurs. Notez qu'il s'agit d'une convention (tout exposer dans kube-public), et non quelque chose qui est fait par défaut dans Kubernetes. kubeadm n'expose que le ConfigMap cluster-info, et rien d'autre.
- **kube-system** : C'est l'espace de noms pour les objets créés par le système Kubernetes. Cet espace contient les comptes de service qui sont utilisés pour faire fonctionner les contrôleurs Kubernetes. Ces comptes de service bénéficient d'autorisations importantes (créer des pods n'importe où, par exemple).

- **Quels sont les pods actifs et leur rôle ?**

```
$ kubectl get pods -n kube-system
NAME                                READY   STATUS    RESTARTS   AGE
calico-kube-controllers-654868b4c-xcb29  1/1     Running   0           29d
canal-4bj5b                          2/2     Running   0           29d
canal-jrbtl                          2/2     Running   0           29d
canal-kbf6s                          2/2     Running   0           29d
coredns-9dc5d84b6-gxmf1              1/1     Running   0           29d
coredns-9dc5d84b6-m87pr               1/1     Running   0           29d
kube-dns-autoscaler-7944bcf59d-nvjlr   1/1     Running   0           29d
kube-proxy-lx4jf                      1/1     Running   0           29d
kube-proxy-vh7v2                      1/1     Running   0           29d
kube-proxy-z7v8l                      1/1     Running   0           29d
metrics-server-68fdc444fc-vr118       1/1     Running   0           29d
wormhole-2b8jl                       1/1     Running   0           29d
wormhole-7ppjw                       1/1     Running   0           29d
wormhole-wdcf7                       1/1     Running   0           29d
```

Nous avons trouvé alors :

- **calico-kube-controllers** : Ce pod fait partie du plugin réseau Calico. Il est responsable de la gestion de diverses ressources dans le cluster Kubernetes liées au réseau, telles que la gestion des attributions de pool IP.
- **canal-4bj5b**, **canal-jrbtl**, **canal-kbf6s** : Ces pods font partie du plugin réseau Canal, qui est basé à la fois sur Calico et Flannel. Ils sont impliqués dans des tâches liées au réseau, assurant la communication entre les pods et la gestion de la superposition réseau.

- **coredns-9dc5d84b6-gxmfl, coredns-9dc5d84b6-m87pr** : Ces pods hébergent CoreDNS, qui est un serveur DNS flexible et extensible pour Kubernetes. CoreDNS assure la découverte de services basée sur DNS à l'intérieur du cluster.
- **kube-dns-autoscaler-7944bcf59d-nvjlr** : Ce pod héberge le redimensionneur automatique de DNS Kubernetes, qui ajuste le nombre de répliques DNS en fonction de l'utilisation des ressources dans le cluster.
- **kube-proxy-ix4jf, kube-proxy-vh7v2, kube-proxy-z7v8l** : Ces pods exécutent le proxy Kubernetes (kube-proxy), qui gère la communication réseau pour les services au sein du cluster.
- **metrics-server-68fdc444fc-vrll8** : Ce pod héberge le serveur de métriques Kubernetes, qui collecte et expose des métriques sur l'utilisation des ressources dans le cluster.

Les résultats indiquent qu'aucun pod n'a été trouvé dans les namespaces kube-node-lease et kube-public :

```
$ kubectl get pods -n kube-node-lease      $ kubectl get pods -n kube-public
No resources found in kube-node-lease namespace.  No resources found in kube-public namespace.
```

2. Déploiement du kubernetes Dashboard :

- Quelle est la fonction de la commande "kubectl proxy" ?
Quelle est la différence avec "kubectl port-forward" ?
Quels sont les cas d'usage de chacun ?

La commande `kubectl proxy` et la commande `kubectl port-forward` sont les deux utilisées dans Kubernetes pour permettre l'accès à des services et des ressources à l'intérieur du cluster. Cependant, elles ont des fonctionnalités et des cas d'utilisation légèrement différents.

- **kubectl proxy** : Crée un serveur proxy ou une passerelle au niveau de l'application entre localhost et le serveur API Kubernetes. Cette commande facilite l'accès à l'API Kubernetes en utilisant l'adresse du proxy, permettant une interaction plus simple avec l'API via un navigateur web ou des outils prenant en charge les proxies HTTP. De plus, elle peut servir du contenu statique via un chemin HTTP spécifié. Toutes les données entrantes passent par un port et sont transférées vers le port du serveur API Kubernetes distant, à l'exception du chemin correspondant au contenu statique.

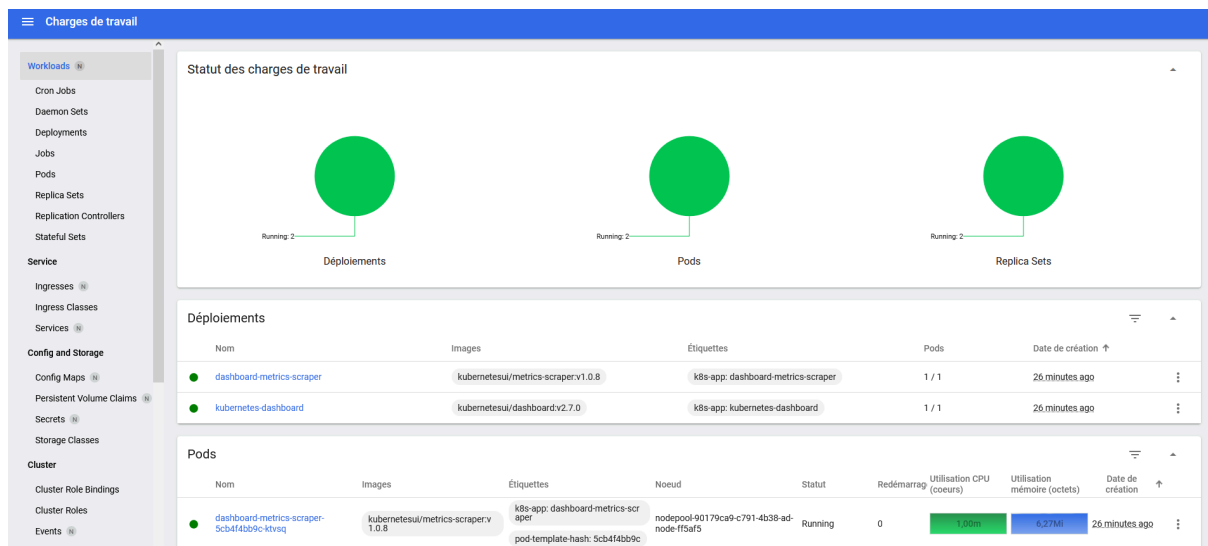
- **kubectl port-forward** : Forward un ou plusieurs ports locaux vers un pods. La session de forwarding se termine lorsque le pod sélectionné meurt ou bien si elle est arrêtée manuellement, une nouvelle exécution de la commande est nécessaire pour reprendre le transfert.

Le principal cas d'utilisation de **kubectl proxy** est l'accès au serveur API Kubernetes, tandis que **kubectl port-forward** est utilisé pour interagir directement avec un pod ou un service spécifique depuis la machine locale. Le port forwarding est souvent employé pour accéder aux ressources internes du cluster et faciliter le débogage. Par exemple, en utilisant **kubectl port-forward** avec un pod Redis, vous pouvez faire avancer le port du pod vers un port local, permettant un accès local à Redis et des opérations spécifiques.

- Installation de Kubernetes Dashboard :

Lors de l'installation de Kubernetes Dashboard, plusieurs ressources sont généralement créées. Ces ressources peuvent inclure des objets Kubernetes tels que des Deployments, Services, Roles, RoleBindings, Secrets, etc. Voici une explication des relations et de l'intérêt de certaines de ces ressources :

- **Deployment** : Le Deployment définit les spécifications pour le déploiement de l'application Kubernetes Dashboard. Il spécifie le conteneur Docker à utiliser, la stratégie de déploiement, le nombre de réplicas...
- **Service** : Le Service expose l'application en tant que service réseau stable avec une adresse IP et un port. Il permet aux autres composants du cluster de communiquer avec l'application Kubernetes Dashboard.
- **Service Account** : Le Service Account définit les autorisations et l'accès aux ressources pour les pods associés au déploiement de l'application. Il s'agit de la manière dont les pods obtiennent des autorisations pour accéder aux ressources Kubernetes.
- **Role et RoleBinding** : Ces ressources définissent les autorisations au niveau du namespace pour les objets de l'API Kubernetes. Elles spécifient quelles actions un utilisateur ou un Service Account peut effectuer sur quelles ressources. Cela est utilisé pour restreindre les autorisations de l'application aux ressources nécessaires dans le namespace.
- **Secret** : Les Secrets peuvent être utilisés pour stocker des informations sensibles telles que les clés d'API, les mots de passe, etc. Ils peuvent être utilisés pour sécuriser l'application Kubernetes Dashboard en stockant les informations sensibles nécessaires à son fonctionnement.



3. Déploiement de l'application sur Kubernetes :

Au début, nous avons déployé une base de données postgres pour enregistrer les URL des clients et une base de données Redis pour mettre à jour le compteur de chaque page. Nous avons utilisé la commande suivante pour ajouter le référentiel de charts bitnami : `helm repo add bitnami https://charts.bitnami.com/bitnami`

Helm utilise ces référentiels pour rechercher et installer des charts Helm, qui sont des packages prédéfinis pour des applications Kubernetes. Ensuite, ce référentiel va nous permettre d'installer des applications à travers Helm sur notre cluster Kubernetes.

Pour installer la BD Redis : `helm install my-redis bitnami/redis --version 18.4.0 -n polymetrie`

```
hamza@LAPTOP-DL40T8DW:~/orchestration-at-scale-23-24-polymetrie-g$ helm install my-redis bitnami/redis --version 18.4.0 -n polymetrie
WARNING: Kubernetes configuration file is group-readable. This is insecure. Location: /home/hamza/.kube/config
WARNING: Kubernetes configuration file is world-readable. This is insecure. Location: /home/hamza/.kube/config
NAME: my-redis
LAST DEPLOYED: Wed Dec 6 21:05:24 2023
NAMESPACE: polymetrie
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: redis
CHART VERSION: 18.4.0
APP VERSION: 7.2.3
```

Cette commande `helm install` déploie toutes les ressources Kubernetes définies dans le chart Helm spécifié vers notre cluster Kubernetes.

Cette commande a généré 2 warnings, comme illustré dans la capture d'écran suivante : "WARNING: Kubernetes configuration file is group/world-readable". Ceci est dû au fait que le fichier `~/.kube/config` ne devrait être lisible que par l'utilisateur. Pour remédier à cela, il est recommandé d'utiliser la commande `chmod` afin de modifier les permissions de lecture/écriture du fichier.

`chmod g+r ~/.kube/config`

`chmod o+r ~/.kube/config`

ou simplement `chmod 600 ~/.kube/config`

```
$ ls -al ~/.kube/config
-rw-r--r-- 1 souro 197609 9734 Jan 10 11:05 /c/Users/souro/.kube/config
```

Nous avons fait la même chose pour postgres, en exécutant :

`helm install my-postgresql bitnami/postgresql --version 13.2.24 -f values-postgres.yaml`

L'argument `-f` sert à personnaliser notre déploiement en spécifiant un nouvel utilisateur de la base de données postgres, et une nouvelle base de donnée `td_1`. Le fichier `values-postgres.yaml` contient les valeurs personnalisées qu'on veut configurer. De plus, nous avons précisé des valeurs de `pgHbaConfiguration` afin de pouvoir se connecter à la base de données depuis notre application.

- Pour le déploiement de notre application, nous avons utilisé le fichier `deployment.yaml` qui contient les ressources suivantes :

- 1) Une ressource Deployment qui utilise notre image docker `hamza125/polymetrie-increment:latest`, nous avons poussé cette image dans un répertoire publique dockerhub afin de pouvoir l'utiliser.
- 2) Une ressource Service qui va exposer notre application sur le port 5000, nous avons utilisé le type ClusterIP pour cette première version.
- 3) Finalement une ressource Ingress pour accéder à notre application via un hostname bien précis : `polymetrie.orch-team-g.pns-projects.fr.eu.org`

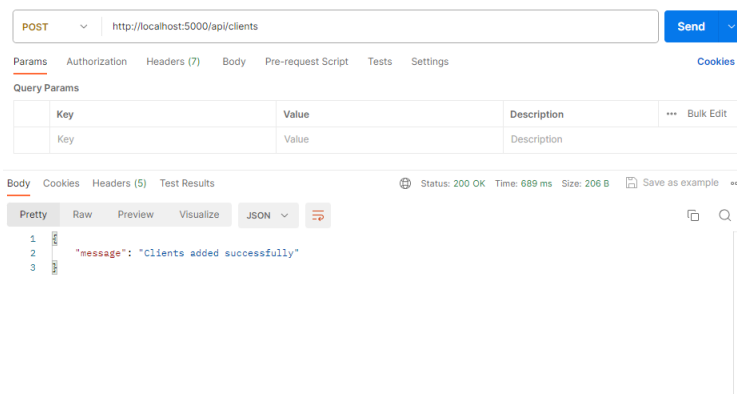
La figure suivante montre les pods que nous avons déployé :

```
$ kubectl get pods -n default
NAME                                READY   STATUS    RESTARTS   AGE
my-postgresql-0                    1/1     Running   1 (20d ago)  20d
my-redis-master-0                  1/1     Running   0           22d
my-redis-replicas-0                1/1     Running   0           22d
my-redis-replicas-1                1/1     Running   0           22d
my-redis-replicas-2                1/1     Running   0           22d
polymetrie-polymetrie-chart-fc76d6ff9-hr1ht  1/1     Running   0           6d12h
```

Pour exposer notre application, on utilise la commande port-forward :

```
$ kubectl port-forward polymetrie-polymetrie-chart-fc76d6ff9-hr1ht 5000:5000 -n default
Forwarding from 127.0.0.1:5000 -> 5000
Forwarding from [::1]:5000 -> 5000
```

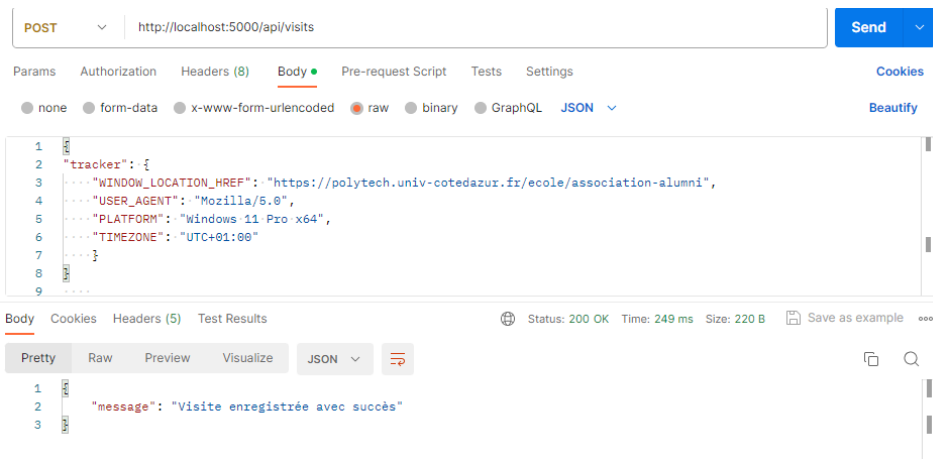
On exécute la première requête pour enregistrer les URL des clients dans la base de données postgres.



```
PS C:\Users\21264> kubectl logs polymetrie-increment-588b5d5664-pss2n
* Serving Flask app 'main'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://10.2.2.39:5000
Press CTRL+C to quit
127.0.0.1 - - [06/Dec/2023 20:29:02] "POST /api/clients HTTP/1.1" 200 -
```

Ensuite, on exécute une requête POST pour mettre à jour le compteur d'une page dans notre base de données redis.

A screenshot of a web browser's developer tools console. It shows a POST request to 'localhost:5000/api/fetch-redis'. The response is a JSON object: {"visits": [{"url": "https://polytech.univ-cotedazur.fr/ecole/association-alumni", "count": 2}]}. The browser's address bar shows the URL 'localhost:5000/api/fetch-redis'.



On constate que toutes les communications avec les bases de données sont fonctionnelles à travers notre déploiement Kubernetes.

4. Les problèmes rencontrés

Au début, nous utilisions un service ClusterIP pour accéder à l'application. Cette approche nous menait à faire un port-forward à chaque fois, alors nous voulions d'abord utiliser un service NodePort parce que nous ne savions pas que le provider nous fournit un Load balancer automatiquement quand on crée un service LoadBalancer. Le problème qu'on a eu c'est que les nœuds n'ont pas d'adresses IP externes. Alors, nous ne pouvons pas l'utiliser pour accéder à l'application. Finalement, nous avons constaté que le provider nous fournit une adresse IP pour accéder à nos services derrière un LoadBalancer, donc nous sommes partis sur cette approche.

C. Industrialisation avec Ansible

Pour cette partie, nous avons défini un fichier de déploiement à travers Ansible (voir fichier [deploy-app-and-dbs.yaml](#)).

Dans ce fichier, nous avons défini plusieurs tâches (tasks) en commençant par la création d'un namespace appelé "polymetrie-increment pour notre application.

Par la suite, nous avons utilisé le module Ansible "**kubernetes.core.helm**" pour orchestrer le déploiement des bases de données requises en utilisant Helm. La première instance de ce module déploie la dernière version du chart Redis, configurant les paramètres nécessaires et établissant un mot de passe d'authentification spécifié. De manière similaire, une deuxième instance du module déploie le chart PostgreSQL, avec des configurations spécifiques telles que les mots de passe pour l'utilisateur et d'autres paramètres d'authentification.

Finalement nous avons créé une dernière tâche qui va utiliser le fichier deployment cité dans la dernière partie pour déployer les ressources de notre application (Deployment, Service, ingress).

Pour exécuter un playbook Ansible, il est essentiel d'installer les dépendances nécessaires. Cela inclut les packages tels que Ansible, pip, python3, le module kubernetes via pip, ainsi

que `community.kubernetes` pour `ansible-galaxy` afin d'obtenir les modules Ansible requis. Une fois ces dépendances installées, la commande à exécuter est `ansible-playbook deploy-app-and-dbs.yaml`, permettant ainsi de lancer l'exécution des tâches définies dans le playbook.

Nous avons rencontré des difficultés lors de l'installation du package `community.kubernetes`, que nous avons résolues en basculant vers WSL. Ansible est conçu exclusivement pour la gestion d'hôtes Windows et ne peut pas s'exécuter directement sur un hôte Windows.

La raison derrière cela est que la plupart des fonctionnalités d'Ansible sont profondément ancrées dans des conventions propres à UNIX, ce qui empêche son fonctionnement natif sur Windows. Fondamentalement, Windows ne dispose pas de l'implémentation de l'appel système `fork()`, ce qui est essentiel pour le fonctionnement d'Ansible.

Cependant, il peut fonctionner sous le sous-système Windows pour Linux (WSL). Par conséquent, nous avons opté pour le tester à partir d'un environnement WSL.

TP 2

A. Terraform

- **Déploiement avec Terraform**

La gestion des dépendances, notamment la base de données PostgreSQL, a été optimisée grâce à l'intégration efficace de Helm dans le processus d'automatisation.

Voici un aperçu détaillé des différentes étapes de configuration au sein du fichier main.tf :

1. **Déclaration des Fournisseurs** : La première section du fichier Terraform a été consacrée à la déclaration des fournisseurs nécessaires pour le déploiement. Cette partie spécifie les versions minimales compatibles des fournisseurs, incluant HashiCorp Kubernetes et Helm.

```
terraform {
  required_providers {
    kubernetes = {
      source = "hashicorp/kubernetes"
      version = ">= 2.0.0"
    }
    helm = {
      source = "hashicorp/helm"
      version = ">= 2.3.0"
    }
  }
}
```

2. **Configuration du Fournisseur Kubernetes** : La configuration du fournisseur Kubernetes a été établie en indiquant le chemin d'accès au fichier kubeconfig.

```
provider "kubernetes" {
  config_path = "~/.kube/config"
}
```

3. **Configuration du Fournisseur Helm** : La section suivante a été réservée à la configuration du fournisseur Helm, où le chemin d'accès au fichier kubeconfig a également été spécifié.

```
provider "helm" {
  kubernetes {
    config_path = "~/.kube/config"
  }
}
```

4. **Création d'un Espace de Noms Kubernetes** : Un espace de noms "terraform" a été créé dans Kubernetes pour isoler les ressources du déploiement.

```
resource "kubernetes_namespace" "polymetrie-increment-teamg"
{
  metadata {
    name = "terraform"
  }
}
```

5. **Déploiement de PostgreSQL avec Helm** : L'utilisation de Helm a été mise en œuvre pour déployer une instance de PostgreSQL. Les paramètres, tels que le nom,

l'espace de noms, le référentiel Helm, le graphique PostgreSQL, la version, et des valeurs personnalisées ont été spécifiés.

```
resource "helm_release" "postgresql" {
  name      = "my-postgresql"
  namespace = kubernetes_namespace.polymetrie-increment
  -teamg.metadata[0].name
  repository = "https://charts.bitnami.com/bitnami"
  chart      = "postgresql"
  version    = "13.2.24"
  values     = [file("../charts/values/postgres/values
    -postgres.yaml")]
}
```

- 6. Déploiement de l'Application Personnalisée :** L'application personnalisée a été déployée à l'aide d'un Deployment Kubernetes, d'un service, et d'un conteneur Docker. L'image "hamza125/polymetrie-increment:latest" a été utilisée, exposant le port 5000.

```
resource "kubernetes_deployment" "polymetrie-increment-teamg" {
  depends_on = [helm_release.postgresql]
  metadata {
    name      = "polymetrie-increment"
    namespace = kubernetes_namespace.polymetrie-increment-teamg.metadata[0].name
  }
  spec {
    replicas = 1
    selector {
      match_labels = {
        app = "polymetrie-increment"
      }
    }
  }
  template {
    metadata {
      labels = {
        app = "polymetrie-increment"
      }
    }
    spec {
      container {
        name      = "polymetrie-increment"
        image     = "hamza125/polymetrie-increment:latest"
        port {
          container_port = 5000
        }
      }
    }
  }
}
```

Le service est exposé sur le port 5000 dans le namespace "terraform" et dépend du déploiement de l'application.

```
resource "kubernetes_service" "polymetrie-increment-teamg" {
  {
    metadata {
      name      = "polymetrie-increment-service"
      namespace = kubernetes_namespace.polymetrie-increment
      -teamg.metadata[0].name
    }
    spec {
      selector = {
        app = "polymetrie-increment"
      }
      port {
        protocol = "TCP"
        port     = 5000
        target_port = 5000
      }
    }
  }
}
```

Nous avons utilisé la commande **terraform init** qui initialise le répertoire de travail Terraform en téléchargeant les plugins nécessaires, configurant les connexions avec les fournisseurs spécifiés, et créant un fichier d'état local. Cette opération prépare l'environnement pour les déploiements Terraform en établissant une base stable pour suivre l'état de l'infrastructure déployée. Il est recommandé d'exécuter cette commande lors du démarrage d'un nouveau projet Terraform ou après des modifications importantes dans la configuration.

```
$ terraform init
Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/helm from the dependency lock file
- Reusing previous version of hashicorp/kubernetes from the dependency lock file
- Using previously-installed hashicorp/helm v2.12.1
- Using previously-installed hashicorp/kubernetes v2.24.0

Terraform has been successfully initialized!

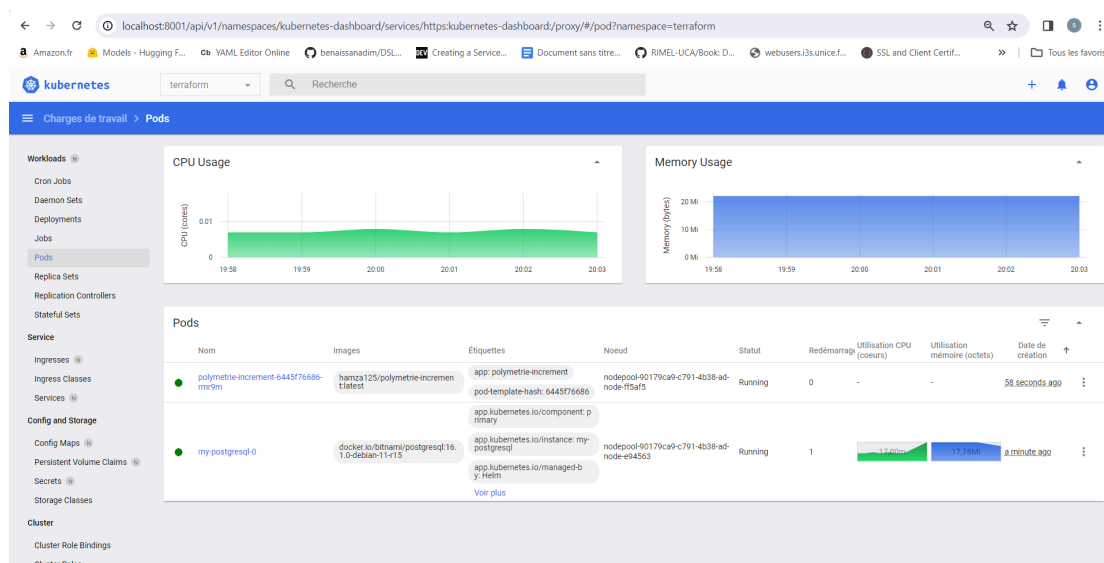
You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Nous avons exécuté la commande **terraform apply** qui déclenche l'application des modifications définies dans la configuration Terraform à l'infrastructure cible, générant un plan d'exécution des opérations nécessaires. Après confirmation de l'utilisateur, Terraform met à jour l'infrastructure, ajuste le fichier d'état pour refléter les changements, et finalise le processus en assurant la cohérence entre la configuration et l'infrastructure déployée. Cette commande est essentielle pour orchestrer efficacement les changements dans un environnement d'infrastructure programmable.

```
kubernetes_namespace.polymetrie-increment-teamg: Creating...
kubernetes_namespace.polymetrie-increment-teamg: Creation complete after 0s [id=terraform]
helm_release.postgresql: Creating...
helm_release.postgresql: Still creating... [10s elapsed]
helm_release.postgresql: Still creating... [20s elapsed]
helm_release.postgresql: Still creating... [30s elapsed]
helm_release.postgresql: Creation complete after 39s [id=my-postgresql]
kubernetes_deployment.polymetrie-increment-teamg: Creating...
kubernetes_deployment.polymetrie-increment-teamg: Creation complete after 4s [id=terraform/polymetrie-increment]
kubernetes_service.polymetrie-increment-teamg: Creating...
kubernetes_service.polymetrie-increment-teamg: Creation complete after 0s [id=terraform/polymetrie-increment-service]

Apply complete! Resources: 4 added, 0 changed, 0 destroyed.
```



- **Explication des outils : Ansible , Terraform et Helm**

- **Terraform** : il est souvent utilisé pour le provisioning de l'infrastructure de manière **déclarative**. Donc il est pertinent de l'utiliser si nous voulons créer notre infrastructure. Par exemple, pour créer un cluster Kubernetes chez un provider avec des ressources bien définies. Il est aussi très pertinent, si nous avons des déploiements multi-cloud. Terraform est considéré comme **stateful**. Il utilise un fichier d'état (state file) pour enregistrer l'état actuel de l'infrastructure gérée.

Terraform est "**immutable**" car il génère une nouvelle version de l'infrastructure à chaque modification, évitant ainsi de modifier directement les ressources existantes.

- **Ansible** : il est utilisé pour l'automatisation et la configuration. Il fonctionne d'une manière **procédurale** en exécutant les tasks dans l'ordre défini dans le playbook sur des hosts. Il donne plus de liberté pour déclarer les opérations et les conditions (par exemple selon les environnements prod, pré-pro). Pour déployer notre stack, nous avons trouvé que Ansible est plus simple avec le format Yaml et la logique de procédure. Ansible est **stateless**. Il fonctionne en poussant l'intention déclarée dans un playbook sans maintenir une connaissance de l'état actuel du système entre les exécutions successives.

Ansible a généralement tendance à créer une infrastructure **mutable**.

Helm : Helm permet de simplifier le déploiement des applications sur K8s, en exécutant quelques commandes nous avons toutes les ressources déployées. Cependant, il ne permet pas d'avoir une automatisation de plusieurs déploiement à la fois comme les 2 dernières technologies. Mais avec ansible et terraform on utilise les librairies de helm pour déployer les applications sous forme de releases.

- **Comparaison Ansible et Terraform**

Terraform	Ansible
Outil de provisionnement	Outil de gestion de configuration
il suit une infrastructure déclarative comme approche de code	Il suit plutôt une approche procédurale
Il meilleur pour orchestrer les services cloud et configurer l'infrastructure cloud	Il est principalement utilisé pour configurer les serveurs avec le bon logiciel et mettre à jour les ressources déjà configurées
il ne prend pas en charge le provisionnement sans système d'exploitation par défaut	Il prend en charge le provisionnement des serveurs bare metal.
il n'offre pas un meilleur support en termes d'empaquetage et de modèles	Il fournit un support complet pour l'empaquetage et la création de modèles.
Il dépend fortement du cycle de vie ou de la gestion de l'état	Il n'a pas du tout de gestion de cycle de vie

Remarque : Dans de nombreux cas, Ansible et Terraform peuvent être utilisés ensemble pour atteindre le résultat souhaité. Par exemple, Terraform peut être utilisé pour provisionner l'infrastructure cloud et Ansible peut être utilisé pour configurer les logiciels et les systèmes s'exécutant sur cette infrastructure.

Ansible et Terraform partagent des similarités significatives dans leur capacité à configurer l'infrastructure cloud, adoptent une architecture sans maître, permettent l'exécution à distance sur des machines virtuelles récemment provisionnées, et offrent une intégration aisée avec les fournisseurs de services cloud via des API.

Toutefois, leurs approches et spécialisations diffèrent. Ansible se concentre sur la gestion de la configuration et l'orchestration à travers un langage YAML, fonctionnant de manière sans agent. D'un autre côté, Terraform se spécialise dans le provisionnement de l'infrastructure via des fichiers de configuration déclaratifs, adoptant une approche agnostique de la plateforme.

Dans le domaine du réseau, Terraform excelle dans le provisionnement, tandis qu'Ansible se distingue davantage dans la gestion de la configuration. Bien que Terraform soit davantage orienté vers le provisionnement de manière programmatique, Ansible, tout en mettant l'accent sur la gestion de la configuration, demeure efficace également pour le provisionnement.

En ce qui concerne la gestion d'état, Ansible opère de manière implicite, tandis que Terraform utilise des fichiers d'état. Dans leur philosophie d'infrastructure, Terraform adopte une approche immuable en provisionnant à partir de zéro, tandis qu'Ansible est plus orienté vers la mutabilité, se concentrant sur la configuration en temps réel.

En matière d'automatisation, Terraform adopte une approche déclarative en définissant l'état souhaité, tandis qu'Ansible suit une approche procédurale en définissant des séquences d'actions.

- **Pensez-vous que toutes ces technologies sont compatibles avec la méthodologie GitOps et ses implémentations (ArgoCD, FluxCD...) ?**

Helm est parfaitement compatible avec la méthodologie GitOps et notamment les outils ArgoCD et FluxCD. En général, la source de vérité est le repository Github qui contient la Helm Chart, s'il y a un changement dans le repository, ArgoCD est capable de le détecter et d'appliquer la nouvelle configuration en exécutant les commandes Helm.

Terraform : Les fichiers Terraform peuvent être stockés dans un dépôt Git. Une pipeline CI/CD (Jenkins, Gitlab CI/CD, Github Action...) peut être configurée afin déclencher le déploiement de l'infrastructure Terraform lorsqu'il y a des modifications dans le répertoire Git. Cependant, il est important de noter que ArgoCD ne prend pas en charge directement les fichiers Terraform. ArgoCD est principalement conçu pour travailler avec des ressources Kubernetes et des configurations Helm. Cela signifie que l'utilisation directe de fichiers Terraform avec ArgoCD peut être moins intuitive.

Pour intégrer Terraform avec ArgoCD, une approche courante consiste à utiliser Terraform pour provisionner l'infrastructure sous-jacente, puis à utiliser des outils spécifiques, tels que Helm, pour déployer des applications sur cette infrastructure provisionnée dans un format compréhensible par ArgoCD.

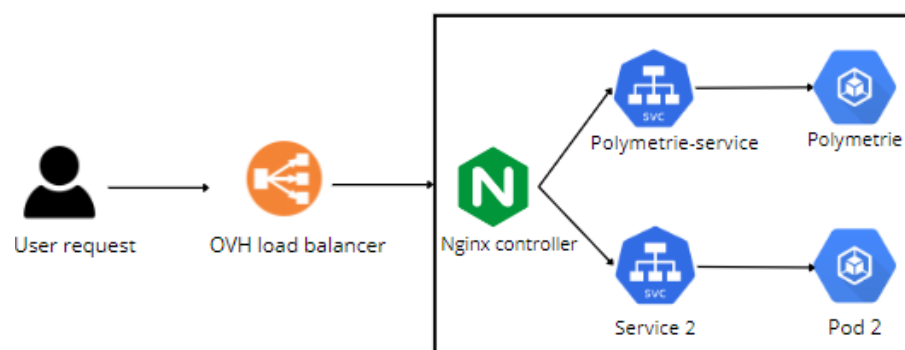
Ansible : Les playbooks Ansible peuvent être intégrés dans un flux GitOps en les stockant dans un dépôt Git. Les outils GitOps peuvent alors déclencher le déploiement d'Ansible lorsqu'il y a des modifications dans le référentiel. De plus, le fait que les fichiers de configuration d'Ansible sont au format YAML facilitent leur utilisation avec des outils compatibles YAML comme ArgoCD.

Les playbook Ansible peuvent effectuer diverses tâches, comme l'application de configurations ou le déploiement d'applications, en réaction aux changements détectés dans le dépôt Git. Ainsi, Ansible s'intègre efficacement dans un environnement GitOps, simplifiant la gestion des configurations et des déploiements.

B. Ingress controller

Pour accéder à nos services depuis l'extérieur, nous utilisons Nginx Ingress Controller en tant que Load Balancer dans le cluster Kubernetes. Il permet l'accès à toutes les applications déployées via une seule adresse IP(attribuée par le Load Balancer OVH au Nginx Ingress Controller). Voici comment il fonctionne :

- Surveille les objets Ingress pour les règles de routage.
- Ajuste dynamiquement la configuration de Nginx en cas de modification dans les objets Ingress.
- Gère les certificats TLS pour le chiffrement HTTPS.
- Route le trafic en fonction des règles définies (nom d'hôte, chemin URL, annotations).
- Redirige le trafic vers les services Kubernetes correspondants.



Nous avons utilisé le déploiement à travers Helm en exécutant les deux commandes suivantes :

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

```
helm -n ingress-nginx install ingress-nginx ingress-nginx/ingress-nginx --create-namespace
```

Ensuite, dans le namespace ingress-nginx, on retrouve les ressources du contrôleur Ingress de Nginx.

Dans la capture d'écran fournie ci-dessous, on note que l'EXTERNAL-IP associé au contrôleur ingress-nginx, configuré en tant que LoadBalancer, affiche une adresse IP de 162.19.109.96. Ce LoadBalancer est responsable de la redirection du trafic des hôtes vers les services appropriés au sein du cluster.

```
souro@superfastboiiii MINGW64 ~/Desktop/AL/orchestration-at-scale-23-24-polymetrie-g/deployment/poly-increment (main)
$ kubectl get svc -n ingress-nginx
```

NAME	PORT(S)	AGE	TYPE	CLUSTER-IP	EXTERNAL-IP
ingress-nginx-controller	80:31939/TCP,443:32209/TCP	18d	LoadBalancer	10.3.119.16	162.19.109.96
ingress-nginx-controller-admission	443/TCP	18d	ClusterIP	10.3.88.205	<none>

Pour tester cette technologie, nous avons créé un ingress pour notre application polymétrie. On a choisi comme nom d'hôte **polymetrie.orch-team-g.pns-projects.fr.eu.org**.

```
souro@superfastboiiii MINGW64 ~/Desktop/AL/orchestration-at-scale-23-24-polymetrie-g/deployment/poly-increment (main)
$ kubectl get ingress -n default
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
polymetrie	<none>	polymetrie.teamg.com	162.19.109.96	80	18d

Dans la ressource ingress il faut bien préciser l'attribut ingress class pour qu'il soit récupéré par le loadbalancer nginx ingress controller.

```
$ kubectl describe ingress polymetrie -n default
```

```
Name:          polymetrie-polymetrie-chart
Labels:        app.kubernetes.io/instance=polymetrie
               app.kubernetes.io/managed-by=Helm
               app.kubernetes.io/name=polymetrie-chart
               app.kubernetes.io/version=1.22.0
               helm.sh/chart=polymetrie-chart-0.1.0
Namespace:     default
Address:       162.19.109.96
Ingress Class: nginx
Default backend: <default>
```

Rules:	Host	Path	Backends
	polymetrie.orch-team-g.pns-projects.fr.eu.org	/	polymetrie-polymetrie-chart:5000 (10.2.1.169:5000)

```
Annotations:
Events:
```

Cette adresse est bien l'adresse du service LoadBalancer nginx ingress controller. On constate bien que tous le trafic vers notre application polymétrie est géré maintenant par le ingress controller.

```
162.19.53.180 - - [23/Dec/2023:11:41:58 +0000] "GET /fetch-redis HTTP/1.1" 404 207 "-" Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36" 464 0.004 [default-polymetrie-polymetrie-chart-5000] [] 10.2.2.160:5000 207 0.003 404 01088d7d3444fd607ecb83ce3e7cf556
162.19.53.180 - - [23/Dec/2023:11:42:04 +0000] "GET /metrics HTTP/1.1" 200 3226 "-" Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36" 460 0.008 [default-polymetrie-polymetrie-chart-5000] [] 10.2.2.160:5000 3226 0.007 200 d386a36a2d89d6800f0874c8d1256cc9
```

C. Argo CD

Avant d'installer Argo CD, explorons son fonctionnement. Il repose sur deux référentiels : l'un pour notre application et l'autre pour l'état désiré du cluster. Argo CD, en tant que contrôleur Kubernetes, peut être déployé dans le même cluster que notre application ou dans un cluster différent. Si l'état en cours ne correspond pas à l'état souhaité, Argo CD tente de le créer et de synchroniser le tout. L'interface utilisateur affiche "OutOfSync" si les états ne sont pas synchronisés, et "Synced" s'ils correspondent.

Nous avons installé l'opérateur Argo CD en suivant la documentation du TD, ce qui nous permet de gérer les instances Argo CD dans notre cluster, par exemple, avec des versions de développement et de production. Cette approche facilite la maintenance de l'application Argo CD grâce aux Custom Resource Definitions (CRD).

Ensuite, pour installer l'application Argo CD, nous avons utilisé la CRD ArgoCD.

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  name: argocd
  namespace: argocd
  labels:
    app: argocd
spec:
  server:
    host: argocd.orch-team-g.pns-projects.fr.eu.org
    ingress:
      ingressClassName: nginx
      enabled: true
    insecure: true
```

Analysons les différentes parties du fichier YAML :

- **apiVersion** : Indique la version de l'API ArgoCD utilisée dans ce manifeste YAML.
- **metadata** : Cette section contient des informations sur la ressource, telles que son nom, son espace de noms (namespace), et des étiquettes (labels) pour organiser et classer la ressource.
- **spec** : C'est la section où sont configurés les détails spécifiques à ArgoCD.
- **host** : Indique l'URL à laquelle le serveur ArgoCD sera accessible.
- **ingress** : Signale l'utilisation d'un Ingress pour exposer le serveur ArgoCD à l'extérieur du cluster Kubernetes.
- **insecure: true** : Autorise les connexions non sécurisées (HTTP) au serveur ArgoCD

La figure dessus représente le fichier yaml de la ressource CRD. En appliquant ce fichier dans le cluster, il sera détecté par Argo CD Operator qui va installer une nouvelle instance Argo CD dans le namespace argocd. De plus, on peut spécifier dans ce fichier la création d'une ressource ingress de type nginx pour accéder à notre Argo CD depuis l'extérieur.

```
2023-12-21T16:02:03Z INFO Starting Controller {"controller": "argocdexport", "controllerGroup": "argoproj.io", "controllerKind": "ArgoCDExport"}
2023-12-21T16:02:03Z INFO Starting workers {"controller": "argocdexport", "controllerGroup": "argoproj.io", "controllerKind": "ArgoCDExport", "worker count": 1}
2023-12-21T16:02:03Z INFO Starting workers {"controller": "argocd", "controllerGroup": "argoproj.io", "controllerKind": "ArgoCD", "worker count": 1}
2023-12-21T16:02:03Z INFO Reconciling ArgoCD {"controller": "argocd", "controllerGroup": "argoproj.io", "controllerKind": "ArgoCD", "ArgoCD": {"name": "argocd", "namespace": "argocd"}, "namespace": "argocd", "name": "argocd", "reconcileID": "b9f65e55-38e2-4224-8f87-9b24fe9da5d6", "namespace": "argocd", "name": "argocd"}
2023-12-21T16:02:03Z INFO controller_argocd reconciling SSO
2023-12-21T16:02:03Z INFO controller_argocd reconciling status
2023-12-21T16:02:03Z INFO conversion-webhook v1beta1 to v1alpha1 conversion requested.
```

On voit bien dans cette capture que l'instance est gérée par l'opérateur qui va créer toutes les ressources nécessaires pour déployer Argo CD en incluant les secrets, configmap, et bien sûr les pods et services.

- **Chaîne de construction des images docker :**

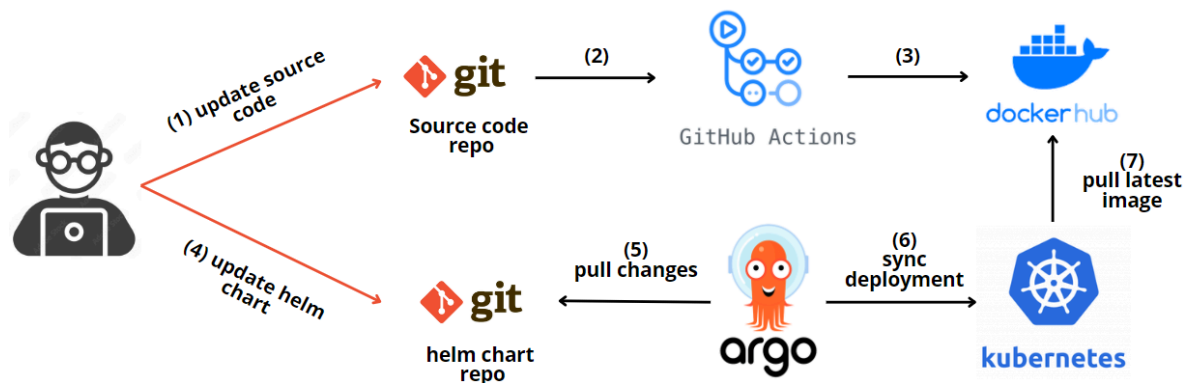
Pour réaliser cette tâche, nous avons utilisé l'outil Github Actions qui va exécuter une pipeline pour construire l'image python de notre application et la publier ensuite dans notre répertoire publique docker **hamza125/polymetrie-increment**.

En ce qui concerne le déploiement de notre stack dans le cluster Kubernetes, nous avons décidé de construire une Chart avec l'outil Helm. Nous avons [un répertoire Github](#) qui contient la configuration de l'application polymetrie :

- **Configuration de l'application sur l'environnement Kubernetes :**

Pour créer une Helm Chart, nous avons utilisé la commande **helm create**. Cette commande crée un projet avec un dossier templates qui contient les ressources Kubernetes qui peuvent être utilisées dans le déploiement (Deployment, Service, Ingress, ServiceAccount, ...). Elle contient également un fichier Values.yml qu'on utilise pour passer les valeurs de notre application aux ressources Kubernetes (par exemple le nom de l'image utilisé dans le déploiement, ou bien le type du service ClusterIP/ LoadBalancer, Et même la configuration de l'ingress).

Argo CD et GitOps :

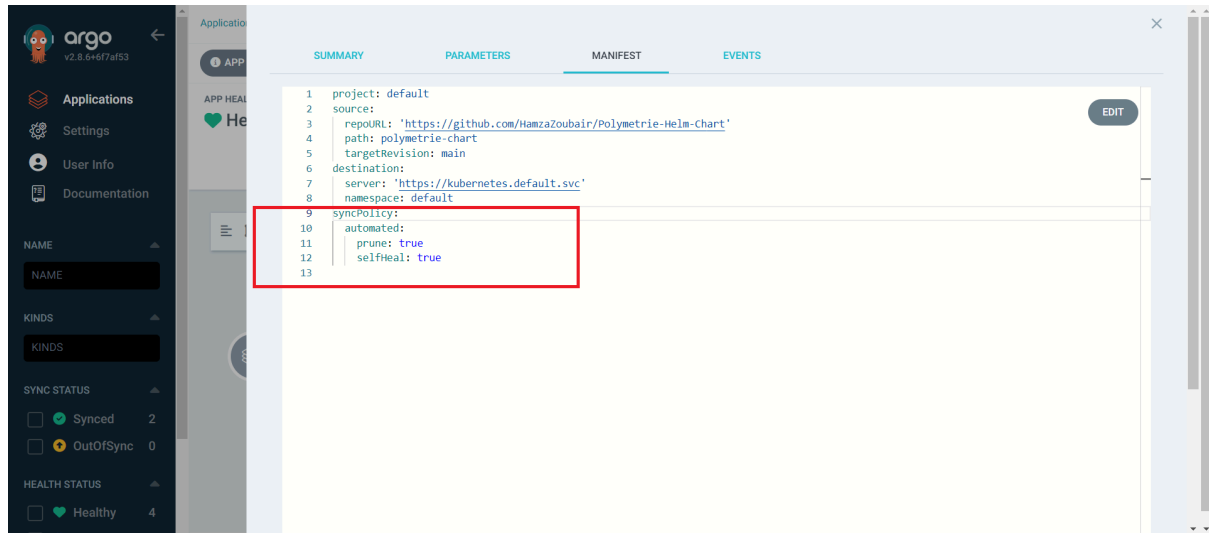


La figure ci- dessus résume le fonctionnement de notre chaîne GitOps.

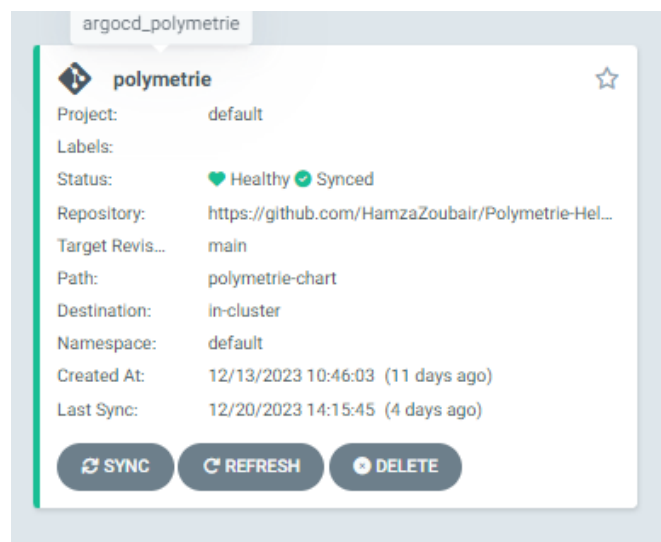
Nous avons ajouté notre application "polymetrie" sur argoCD à l'aide de l'interface utilisateur dédiée. Cette étape a requis les actions suivantes :

- **Fourniture du lien du répertoire** : Nous avons fourni le lien du référentiel Git où se trouve le code source de l'application "polymetrie". Cette information permet à Argo CD de récupérer les configurations nécessaires.
- **Spécification du cluster Kubernetes** : Le lien vers le cluster Kubernetes a été spécifié pour indiquer où le déploiement de l'application doit avoir lieu. Dans notre cas, nous avons choisi de déployer dans le même cluster qu'Argo CD, utilisant le paramètre `https://kubernetes.default.svc`.
- **Configuration des détails spécifiques** : Quelques détails supplémentaires ont été spécifiés, incluant la branche spécifique du code source à utiliser (dans notre cas, la branche "main") et le namespace de déploiement dans Kubernetes.
- **Mise en Place d'une Politique de Synchronisation Automatique** : Pour automatiser la synchronisation entre le code source et le cluster Kubernetes, une politique a été établie avec deux éléments clés :

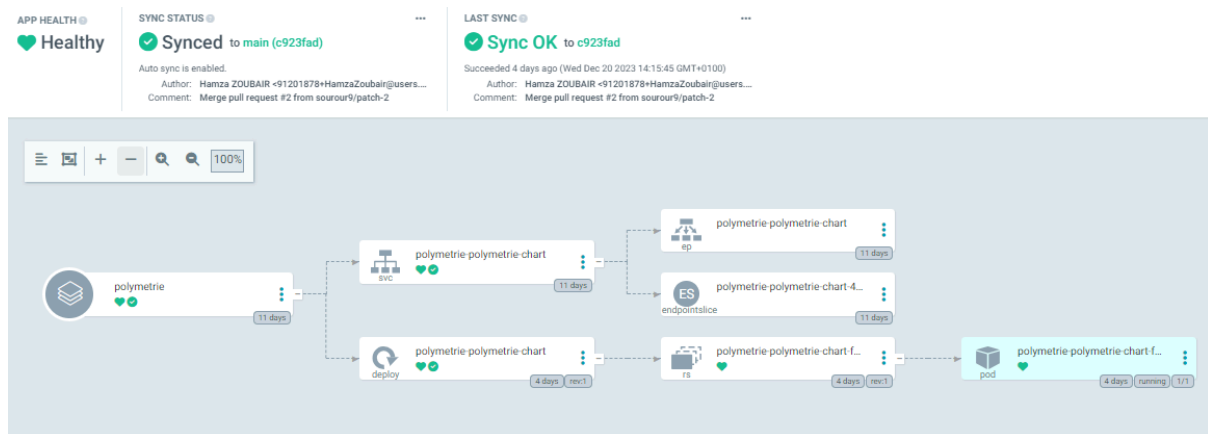
- **prune: true** : Autorisation pour Argo CD de supprimer automatiquement les ressources Kubernetes non déclarées dans le référentiel Git.
- **selfHeal: true** : Argo CD synchronise la configuration du référentiel Git avec l'état réel dans le cluster Kubernetes, annulant toute mise à jour manuelle.



Après l'ajout de l'application, nous avons validé son existence et son état en consultant [l'interface utilisateur d'Argo CD](#). Celle-ci nous a confirmé que les ressources nécessaires ont été créées avec succès dans le cluster Kubernetes.



Nous avons également vérifié le statut de l'application qui a été indiqué comme "Healthy". Ce statut confirme que toutes les ressources ont été déployées sans erreur, attestant du succès du processus d'intégration.



Pour y accéder, veuillez utiliser les identifiants suivants : username: **admin** et password: **bCStMcxRHezrhWZ1A2q0Uy3BPm9QXLGs**

Bien que nous ayons opté pour l'utilisation de l'interface graphique d'Argo CD pour intégrer notre application, il convient de souligner que ces actions peuvent également être réalisées efficacement via la commande `argocd app create`.

Voici un exemple concret de cette procédure avec la commande `argocd app create`, intégrant l'application "polymetrie" dans Argo CD. Les paramètres essentiels, tels que le référentiel Git, le chemin du code source, le serveur de destination (cluster Kubernetes), le namespace, et la révision du code source, sont spécifiés :

```
argocd app create polymetrie \
--repo https://github.com/HamzaZoubair/Polymetrie-Helm-Chart \
--path polymetrie-chart \
--dest-server https://kubernetes.default.svc \
--dest-namespace default \
--revision main
```

Une fois l'application ajoutée, la commande `argocd app set` définit des options de synchronisation automatique, comme `--auto-prune` et `--self-heal` pour la suppression automatique des ressources non déclarées dans le Git et l'activation de l'auto-réparation :

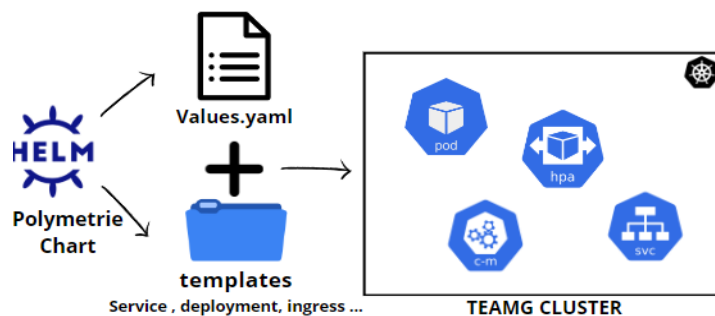
```
argocd app set polymetrie \
--sync-policy automated \
--auto-prune \
--self-heal
```

Pour confirmer le succès du processus d'intégration, `argocd app get polymetrie` vérifie l'état de l'application, fournissant des informations détaillées sur sa santé.

- **Avis d'utilisation :**

Après le déploiement de l'application polymétrie, ArgoCD s'est révélé essentiel. Avant, les modifications manuelles dans plusieurs fichiers de déploiement étaient laborieuses, mais ArgoCD simplifie le processus en centralisant les changements dans un référentiel GitHub. Grâce à Helm, les ajustements se font dans un fichier unique `values.yaml`, et ArgoCD

applique ces changements sur les ressources de la Chart. De plus, Argo CD fournit un suivi précis de l'état de l'application, des logs, et signale les erreurs, facilitant ainsi l'automatisation des déploiements avec GitOps.



- **Problèmes rencontrés :**

- 1) Après avoir créé notre Chart, notre objectif était d'intégrer notre application à ArgoCD. Cependant, nous avons rencontré un problème : l'instance d'ArgoCD ne parvenait pas à détecter les autres espaces de noms. Nous avons résolu ce problème en découvrant la possibilité de configurer notre espace de noms de manière à ce qu'il soit repérable par ArgoCD. Ainsi, nous avons ajouté le libellé suivant à la configuration de notre espace de noms **default**, où nous avons l'intention de déployer l'application : "argocd.argoproj.io/managed-by: argocd". Grâce à l'utilisation de ce libellé, Argo CD est désormais capable de gérer des ressources non seulement dans l'espace de noms où il est installé, mais aussi dans le namespace default.

Remarque : En tant qu'utilisateur administratif, ce label confère des privilèges de **namespace-admin** à ArgoCD, ce qui n'est généralement pas recommandé. L'utilisation des [custom roles](#) résout ce problème : Il est possible de configurer un rôle cluster commun pour l'ensemble des espaces de noms gérés en utilisant les liens de rôle pour le contrôleur d'application Argo CD (avec la variable `CONTROLLER_CLUSTER_ROLE`) et le serveur Argo CD (avec la variable `SERVER_CLUSTER_ROLE`). Si ces variables contiennent des rôles personnalisés, l'opérateur utilise ces rôles au lieu de créer le rôle d'administrateur par défaut, étendant ainsi l'utilisation du rôle personnalisé à tous les espaces de noms gérés.

- 2) En restant, dans le même contexte, nous avons trouvé que ArgoCD permet de créer des utilisateurs (comptes) avec des rôles limités. Nous pouvons créer un utilisateur qui ne peut pas ajouter ou supprimer des applications, cela est pertinent dans le cas d'une entreprise. Nous pouvons associer chaque groupe d'utilisateurs à un rôle spécifique. Ces rôles peuvent être déclarés dans une [ConfigMap](#).

TP3:

A. Kube-Prom-Stack

Le premier pas consiste à ajouter le référentiel Helm de la communauté Prometheus pour accéder aux charts nécessaires. Nous avons utilisé la commande suivante :

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
```

Ensuite nous avons exécuté la commande suivante pour s'assurer que nos référentiels locaux Helm sont à jour pour obtenir les dernières versions des charts:

```
helm repo update
```

Pour déployer la stack technologique (grafana, prometheus et alermanager) nous avons utilisé cette commande :

```
helm install prometheus-stack prometheus-community/kube-prometheus-stack  
--namespace monitoring --values=values-kube-prometheus-stack.yaml  
--create-namespace
```

```
souro@superfastboiii MINGW64 ~/Desktop/AL/orchestration-at-scale-23-24-polymetrie-g/deployment/charts/values/prometheus (main)  
$ helm install prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring --values=values-kube-prometheus-stack.yaml --create-namespace  
NAME: prometheus-stack  
LAST DEPLOYED: Sun Dec 24 14:56:29 2023  
NAMESPACE: monitoring  
STATUS: deployed  
REVISION: 1  
NOTES:  
kube-prometheus-stack has been installed. Check its status by running:  
  kubectl --namespace monitoring get pods -l "release=prometheus-stack"  
  
Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to create & configure Alertmanager and Prometheus instances using the Operator.  
  
souro@superfastboiii MINGW64 ~/Desktop/AL/orchestration-at-scale-23-24-polymetrie-g/deployment/charts/values/prometheus (main)  
$ kubectl --namespace monitoring get pods -l "release=prometheus-stack"  
NAME                                READY   STATUS    RESTARTS   AGE  
prometheus-stack-kube-prom-operator-fcb64cc5-ttvwf  1/1     Running   0           76s  
prometheus-stack-kube-state-metrics-b67cd45df-m5qfr  1/1     Running   0           76s  
prometheus-stack-prometheus-node-exporter-728qq     1/1     Running   0           76s  
prometheus-stack-prometheus-node-exporter-clhr5     1/1     Running   0           76s  
prometheus-stack-prometheus-node-exporter-lpnnl     1/1     Running   0           76s
```

Cette commande permet de déployer la stack technologique dans le namespace monitoring, en utilisant la chart kube-prometheus-stack du référentiel prometheus-community, avec des paramètres personnalisés définis dans le fichier **values-kube-prometheus-stack.yaml**. L'option **--create-namespace** garantit que le namespace est créé s'il n'existe pas déjà.

Le fichier values-kube-prometheus-stack.yaml inclut principalement les valeurs par défaut proposées par prometheus-community. Cependant, certaines de ces valeurs ont été modifiées pour répondre aux exigences spécifiques définies dans le TD :

1. La première exigence consistait à utiliser un stockage persistant de 16 GiB pour la TSDB de Prometheus, avec une rétention des données limitée à 7 jours.

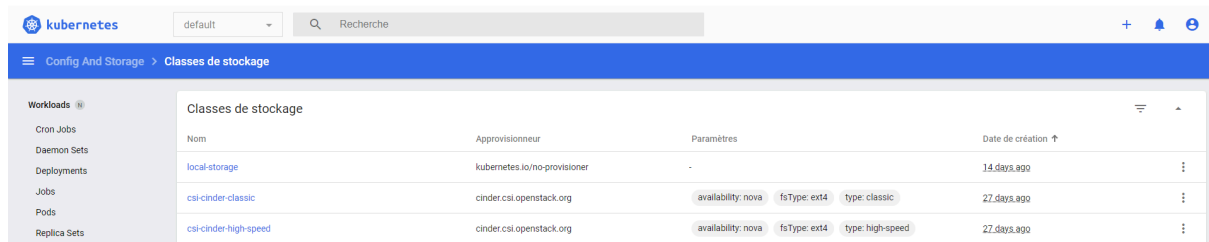
```
prometheus:  
  prometheusSpec:  
    storageSpec:  
      volumeClaimTemplate:  
        spec:  
          storageClassName: "csi-cinder-classic"  
          accessModes: ["ReadWriteOnce"]  
          resources:  
            requests:  
              storage: 16Gi  
      retention: 7d
```


Analysons ces paramètres :

- **storageSpec**: Cette section spécifie la configuration du stockage persistant pour la base de données temporelle (TSDB) de Prometheus. On utilise un modèle de demande de volume (volumeClaimTemplate) pour définir les caractéristiques du stockage. Les Persistent Volume Claims (PVC) dans Kubernetes permettent aux développeurs de demander dynamiquement des espaces de stockage persistant, en spécifiant la quantité et les caractéristiques nécessaires. Ces demandes sont ensuite automatiquement liées aux Persistent Volumes (PV), qui représentent des ressources de stockage physiques dans le cluster,
- **storageClassName**: Indique la classe de stockage utilisée, dans notre cas, "csi-cinder-classic".

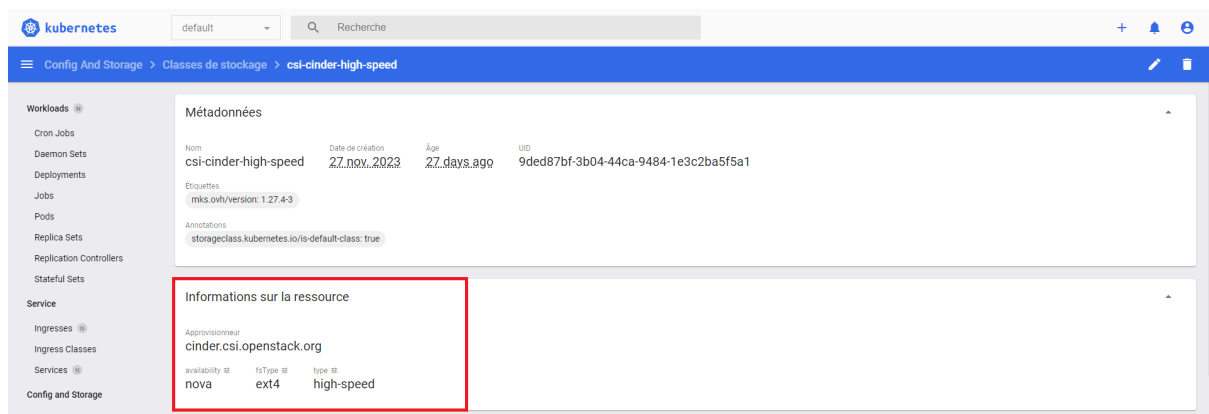
À ce point, nous avons approfondi notre compréhension de ce qu'est une classe de stockage dans un PVC. Nous avons découvert que Kubernetes utilise l'objet Storage Class pour décrire le stockage avec des caractéristiques spécifiques. Les PVC utilisent ces Storage Classes pour provisionner le stockage. Une Storage Class peut avoir un provisionner associé, déclenchant l'attente par Kubernetes de la provision du volume par le fournisseur spécifié.

Nous avons examiné les classes de stockage déjà installées dans notre cluster, cherchant à mieux comprendre les options disponibles et les caractéristiques associées à chaque classe.



Nom	Provisionneur	Paramètres	Date de création ↑
local-storage	kubernetes.io/no-provisioner	-	14 days ago
csi-cinder-classic	cinder.csi.openstack.org	availability: nova fsType: ext4 type: classic	27 days ago
csi-cinder-high-speed	cinder.csi.openstack.org	availability: nova fsType: ext4 type: high-speed	27 days ago

Cet exemple de classe de stockage Kubernetes, nommé "csi-cinder-high-speed", utilise le provisionneur "cinder.csi.openstack.org" avec des paramètres spécifiques tels que l'availability "nova" et un type de stockage "high-speed".



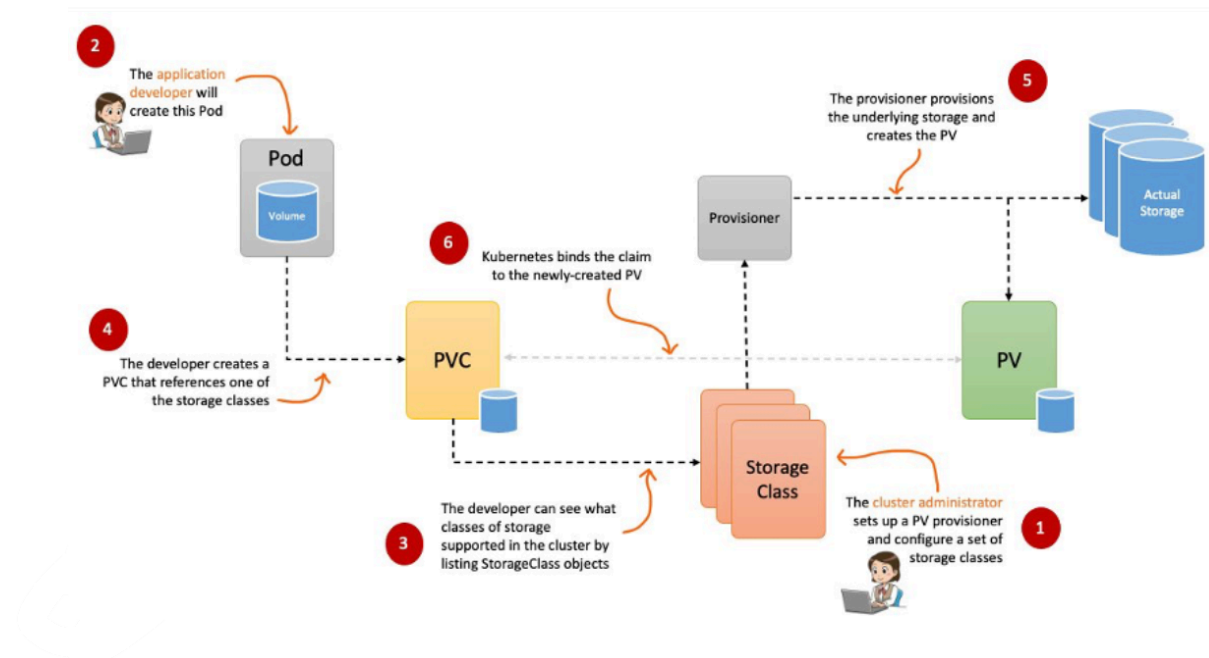
Nom	Date de création	Âge	UID
csi-cinder-high-speed	27 nov. 2023	27 days ago	9ded87bf-3b04-44ca-9484-1e3c2ba5f5a1

Informations sur la ressource			
Provisionneur	cinder.csi.openstack.org		
availability	fsType	type	
nova	ext4	high-speed	

La différence entre csi-cinder-high-speed et csi-cinder-classic réside dans le type de dispositif de stockage physique utilisé. Le premier (csi-cinder-high-speed) fait usage de SSD, tandis que le second (csi-cinder-classic) utilise des disques rotatifs traditionnels.

Pour résumer notre recherche dans la compréhension du processus de gestion du stockage dans Kubernetes, l'administrateur du cluster initie le cycle en créant une classe de stockage (StorageClass), définissant ainsi les propriétés du stockage. Les développeurs ensuite

formulent des Persistent Volume Claims (PVC) pour demander dynamiquement des espaces de stockage, en spécifiant leurs besoins. Ces PVC sont automatiquement associés à des Persistent Volumes (PV), concrétisant des ressources de stockage physiques provisionnées conformément aux spécifications de la storage class définie.



accessModes: Définit les modes d'accès au stockage. Ici, "ReadWriteOnce" signifie qu'un seul nœud peut monter ce stockage en lecture-écriture.

resources: Spécifie les ressources allouées au stockage, avec une demande de 16 GiB.

retention: Cette option définit la durée de rétention des données dans la base de données Prometheus. Dans ce cas, la rétention est configurée à 7 jours. Cela signifie que les données seront conservées pendant cette période avant d'être automatiquement purgées.

2. Pour la deuxième exigence, la taille du stockage persistant de Grafana a été spécifiée dans le fichier de valeurs, conformément à la demande.

```
grafana:
  enabled: true
  persistence:
    enabled: true
    storageClassName: "csi-cinder-classic"
    accessModes: ["ReadWriteOnce"]
    size: 1Gi
```

Analysons ces paramètres spécifiques :

persistence: Cette section spécifie les paramètres de stockage persistant pour Grafana.

enabled: Indique si le stockage persistant est activé pour Grafana. Dans ce cas, il est activé (true).

storageClassName: Définit la classe de stockage utilisée, ici, "csi-cinder-classic".

accessModes: Définit les modes d'accès au stockage. "ReadWriteOnce" signifie qu'un seul nœud peut monter ce stockage en lecture-écriture.

size: Spécifie la taille du stockage persistant alloué à Grafana, dans ce cas, 1 GiB.

3. La troisième exigence du TD consiste à mettre en place un "Ingress" pour permettre un accès facile à Grafana et Prometheus via des URLs spécifiques

```
grafana:
  ingress:
    enabled: true
    ingressClassName: nginx
    annotations: {}
    labels: {}
    hosts:
      - "grafana.orch-team-g.pns-projects.fr.eu.org"
    paths:
      - /

prometheus:
  ingress:
    enabled: true
    ingressClassName: nginx
    annotations: {}
    labels: {}
    hosts:
      - "prometheus.orch-team-g.pns-projects.fr.eu.org"
    paths:
      - /
```

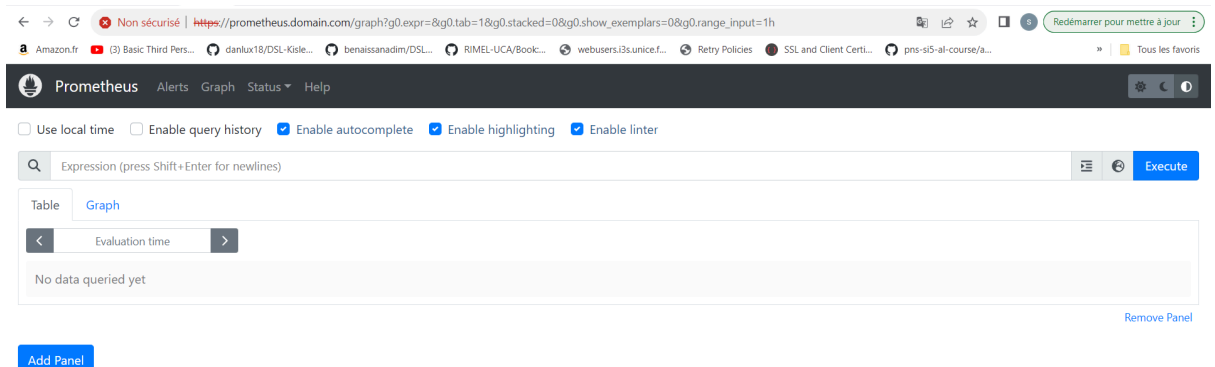
- **Explications détaillées de l'Ingress :**

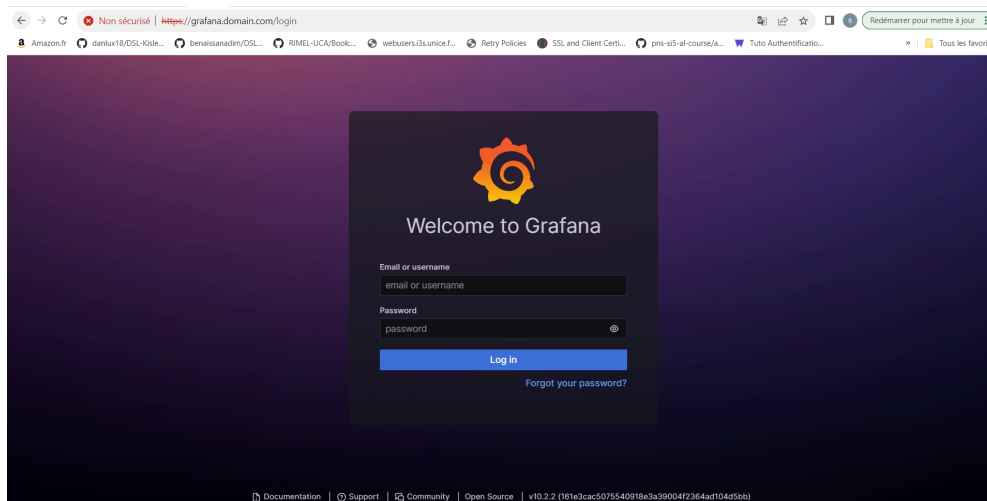
enabled: true : Active la configuration Ingress, indiquant que les règles spécifiées seront utilisées lors du déploiement.

ingressClassName: nginx : Spécifie la classe de l'Ingress à utiliser, indiquant que le contrôleur NGINX Ingress sera responsable de la gestion de cet Ingress.

hosts: - "prometheus.orch-team-g.pns-projects.fr.eu.org" : Spécifie le(s) nom(s) de domaine associé(s) à l'Ingress pour Prometheus. Nous avons décidé de rendre Prometheus accessible via l'URL prometheus.orch-team-g.pns-projects.fr.eu.org. Pour Grafana, le domaine est grafana.orch-team-g.pns-projects.fr.eu.org.

paths: - / : Indique le chemin d'accès de l'URL pour lequel cette règle Ingress est applicable. Nous avons décidé de rendre Prometheus et Grafana accessibles à la racine de l'URL.





- **Intérêt d'utiliser Prometheus dans un contexte de déploiement conteneurisé et de passage à l'échelle :**

Lors du passage à l'échelle, nous commençons à remarquer des comportements anormaux de nos applications. Cela est dû aux charges des utilisateurs, les problèmes réseaux, problèmes de stockage, etc. C'est dans ce contexte où la notion d'observabilité s'avère importante. En effet, nous avons besoin d'un outil qui sera capable de récupérer l'état de nos applications en temps réel. Prometheus est parmi les technologies les plus utilisées dans ce domaine, il fonctionne avec un système de scraping périodique. Il va questionner les services observés pour extraire les métriques pertinentes. Nous avons deux cas :

- 1) Les métriques système : Ils englobent toutes les informations sur la consommation du CPU, mémoire, disque et réseau.
- 2) Les métriques métier : Ce sont les informations métier qu'on veut vérifier périodiquement. Par exemple : la durée de vie moyenne d'une session utilisateur, le nombre de téléchargements, ou bien le nombre de requêtes avec succès.

Prometheus va nous permettre de surveiller nos applications et nos infrastructures pour assurer le bon fonctionnement métier et système.

- **Alternatives à cette technologie :**

- 1) InfluxDB : c'est un outil de monitoring qui stocke les time series comme Prometheus. Il permet de d'utiliser des requêtes SQL pour l'interrogation des données.
- 2) Elasticsearch : C'est pas un outil de monitoring, mais il peut être utilisé pour analyser les logs et détecter des comportements sur nos systèmes.
- 3) DataDog : c'est l'outil de monitoring le plus facile à utiliser, il est en mode SAAS. Donc l'utilisateur ne gère pas l'infrastructure du logiciel. Il intègre de plus des techniques de Machine learning pour détecter les anomalies.

- **Prometheus et Kubernetes Metrics Server :**

Kubernetes Metrics Server est un composant de base de Kubernetes qui permet de collecter les métriques du cluster telles que la consommation des ressources par les pods, nodes, etc. Il ne permet pas de stocker les données à long terme. Ce composant n'est pas capable de récupérer les métriques métier. Il est conçu juste pour communiquer les métriques systèmes des ressources déployées.

Par contre Prometheus est un outil avancé de Monitoring et Alerting. Il permet de collecter les métriques métier et système, il permet de définir ces propres règles de monitoring à l'aide du langage promql. De plus, nous pouvons créer des règles d'alerting qui seront évaluées par notre prometheus. S'il détecte que la condition d'une règle d'alerting est réalisée, il envoie ces alertes au composant AlertManager. Ce composant permet de rediriger les alertes vers les différentes destinations (slack, email, ...).

B. Un peu de Graphes

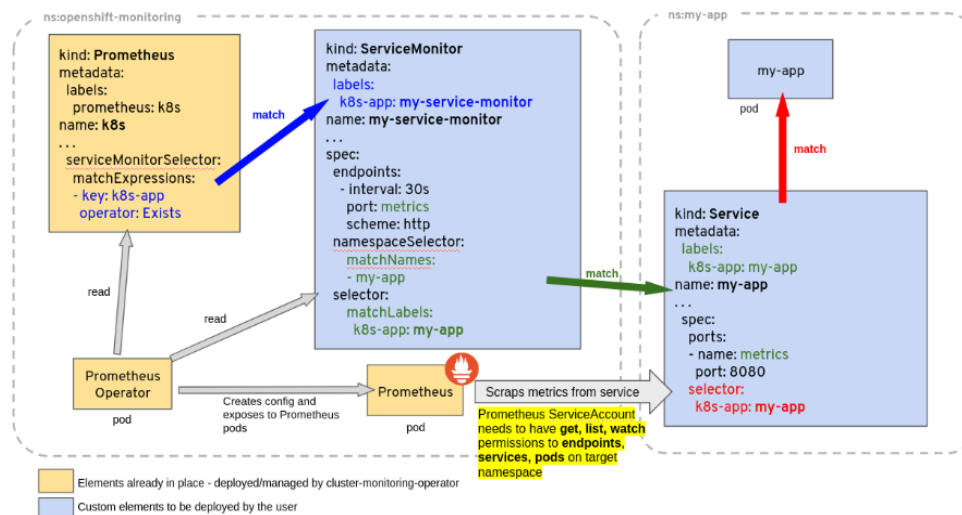
- **En réfléchissant à notre projet, une question clé est apparue : qu'est-ce qu'un Service Monitor ?**

Le Service Monitor, une Définition de Ressource Personnalisée (CRD) de l'opérateur Prometheus, permet la configuration de la surveillance des services et la spécification de la collecte de métriques pour divers services. Utilisé en interne par l'opérateur Prometheus, le Service Monitor configure ses paramètres en lien avec Prometheus, les ServiceMonitors et les Services. Prometheus détecte les ServiceMonitors en interrogeant périodiquement l'API Kubernetes et en filtrant les objets de type ServiceMonitor.

Les ServiceMonitors décrivent les règles de découverte de cibles au sein des Services, permettant à Prometheus de collecter des métriques de manière dynamique. Ces règles spécifient comment identifier les endpoints au sein des Services, en utilisant des critères tels que les labels et les ports.

Une fois les règles établies, Prometheus scrute activement les endpoints, collecte les métriques exposées par les Services, et les utilise pour la surveillance et les alertes. L'image illustre clairement la connexion entre Prometheus, les ServiceMonitors et les Services.

À l'état initial, Prometheus identifie les PodMonitors et les ServiceMonitors uniquement au sein de son propre namespace, étiquetés avec la même balise de version que la sortie de la version du prometheus-operator. Cette approche restreint la découverte des métriques aux seuls PodMonitors dans le même namespace. Pour surmonter cette limitation, nous avons donc entrepris de reconfigurer notre installation Helm kube-prometheus-stack en ajustant le paramètre `serviceMonitorSelectorNilUsesHelmValues` pour le définir sur `false`. De cette manière, Prometheus peut désormais détecter tous les ServiceMonitors dans son espace de noms, supprimant ainsi les contraintes imposées par les sélecteurs préconfigurés. Il est à noter que la fonctionnalité par défaut de la découverte des PodMonitors a également été désactivée en configurant `podMonitorSelectorNilUsesHelmValues` sur `false`. Cette démarche élargit la portée de la surveillance des métriques au-delà du seul namespace de Prometheus.



Lors de la mise à jour de notre installation, nous avons exécuté la commande suivante pour utiliser les nouveaux paramètres de configuration :

```
helm upgrade --install prometheus-stack
prometheus-community/kube-prometheus-stack --namespace monitoring
--values=values-kube-prometheus-stack.yaml
```

```
sour0@superfastboiii MINGW64 ~/Desktop/AL/orchestration-at-scale-23-24-polymetrie-g/deployment/charts/values/prometheus (main)
$ helm upgrade --install prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring --values=values-kube-prometheus-stack.yaml
Release "prometheus-stack" has been upgraded. Happy Helming!
NAME: prometheus-stack
LAST DEPLOYED: Sun Dec 24 15:04:10 2023
NAMESPACE: monitoring
STATUS: deployed
REVISION: 2
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
kubectl --namespace monitoring get pods -l "release=prometheus-stack"
```

Visit <https://github.com/prometheus-operator/kube-prometheus> for instructions on how to create & configure Alertmanager and Prometheus instances using the Operator.

Voici notre Service monitor, permettant à Prometheus de collecter les métriques de notre application :

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app: polymetrie
  name: polymetrie-service-monitor
  namespace: monitoring
spec:
  endpoints:
    - interval: 30s
      port: http
      path: /metrics
  namespaceSelector:
    matchNames:
      - default
  selector:
    matchLabels:
      app: polymetrie
```

Analysons ces paramètres :

- **Métadonnées :** Nous avons attribué des métadonnées à notre objet ServiceMonitor. Le nom choisi est "polymetrie-service-monitor" avec l'étiquette "app: polymetrie". De plus, cet objet a été créé dans le namespace "monitoring".
- **Spécifications du ServiceMonitor :** Dans la section spec, nous avons défini les spécifications pour la surveillance du service.

- Endpoints : Nous avons configuré un point de terminaison qui interroge le service toutes les 30 secondes via le protocole HTTP sur le chemin "/metrics".
- Namespace Selector : La surveillance est limitée au namespace "default".
- Selector : La sélection des pods à surveiller est basée sur l'étiquette "app: polymetrie".

Nous avons exécuté la commande suivante pour installer le service monitor :

```
souro@superfastboiiii MINGW64 ~/Desktop/AL/orchestration-at-scale-23-24-polymetrie-g/deployment/prometheus (main)
$ kubectl apply -f servicemonitor.yaml -n monitoring
servicemonitor.monitoring.coreos.com/polymetrie-service-monitor created

souro@superfastboiiii MINGW64 ~/Desktop/AL/orchestration-at-scale-23-24-polymetrie-g/deployment/prometheus (main)
$ kubectl get serviceMonitor -n monitoring
```

NAME	AGE
polymetrie-service-monitor	101s
prometheus-stack-grafana	3m42s
prometheus-stack-kube-prom-alertmanager	3m42s
prometheus-stack-kube-prom-apiserver	3m42s
prometheus-stack-kube-prom-coredns	3m42s
prometheus-stack-kube-prom-kube-controller-manager	3m42s
prometheus-stack-kube-prom-kube-etcd	3m42s
prometheus-stack-kube-prom-kube-proxy	3m42s
prometheus-stack-kube-prom-kube-scheduler	3m42s
prometheus-stack-kube-prom-kubelet	3m42s
prometheus-stack-kube-prom-operator	3m42s
prometheus-stack-kube-prom-prometheus	3m42s
prometheus-stack-kube-state-metrics	3m42s
prometheus-stack-prometheus-node-exporter	3m42s

Nous avons vérifié sur Prometheus que notre ServiceMonitor est opérationnel et qu'aucune erreur n'a été détectée. Nous avons confirmé que Grafana est associé à data source Prometheus, en l'occurrence celle que nous avons déployée dans notre cluster Kubernetes.

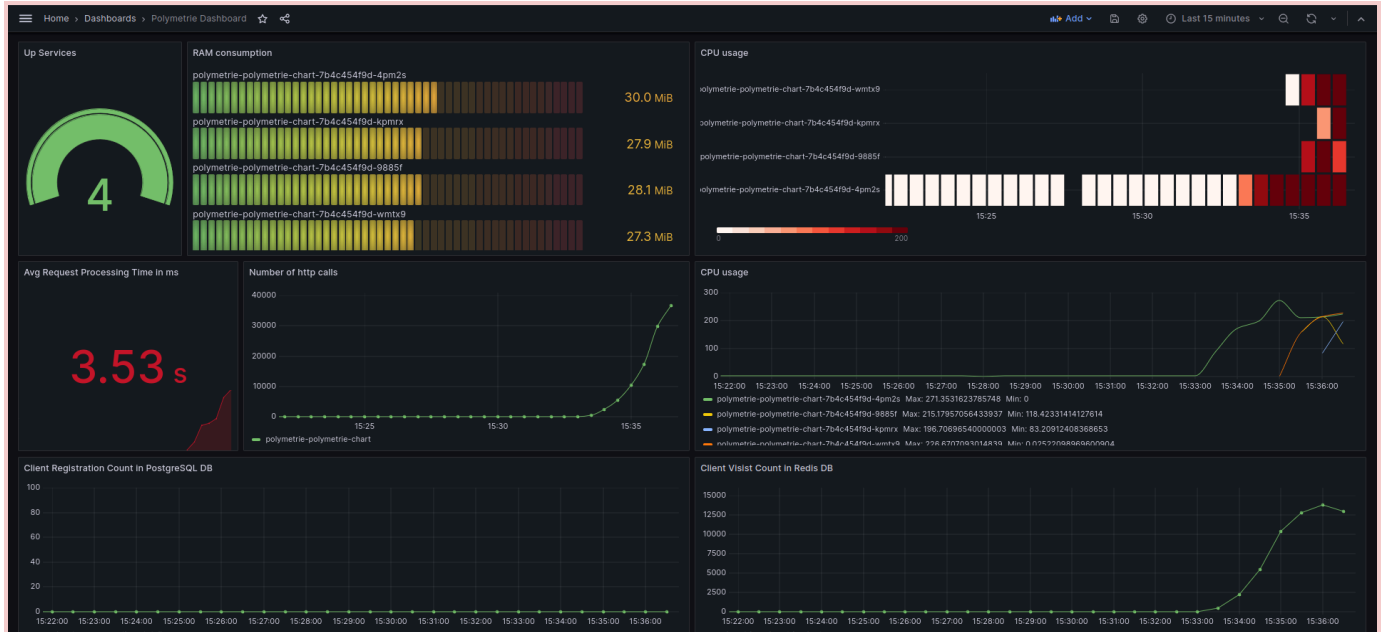
The top screenshot shows the Prometheus web interface at <https://prometheus.team-g.com/targets?search=#pool-serviceMonitor/monitoring/polymetrie-service-monitor/0>. It displays a table of scrape targets. The first target is 'http://10.2.2.160:5000/metrics', which is in a 'UP' state. The table also shows the last scrape time (33.696s ago) and the scrape duration (8.984ms).

The bottom screenshot shows the Grafana web interface at <https://grafana.team-g.com/connections/datasources>. It displays a list of data sources. Two data sources are listed: 'Alertmanager' and 'Prometheus'. The 'Prometheus' data source is highlighted with a red box. It shows the URL 'http://prometheus-stack-kube-prom-prometheus.monitoring:9090/' and the namespace 'default'.

Métriques et Visualisation : Dashboard Grafana de Polymetrie

Pour visualiser les métriques, nous avons créé [un tableau de bord grafana](#), basé sur notre compréhension de ce qui est l'information la plus "importante à transmettre" en premier lieu. Pour y accéder, veuillez utiliser les identifiants suivants pour vous connecter :

- username : **admin** et password : **teamg**



Métriques système :

1. **Up Services** : indique le nombre de services opérationnels de polymétrie.

Type de graphique : Jauge. La jauge est utilisée ici pour une référence visuelle rapide du nombre de services opérationnels, ce qui est un indicateur immédiat du scale du système.

2. **RAM Consumption** : ce graphique illustre l'utilisation de la mémoire par les différents services.

Type de graphique : Diagramme à barres horizontales. Ce choix permet de comparer la consommation de mémoire vive des différents services de manière à ce qu'elle soit lisible d'un seul coup d'œil et donne l'impression d'une "capacité qui se remplit lentement".

Choix de Couleurs : Dégradé de vert à rouge. Ce schéma de couleurs indique les niveaux d'utilisation : le vert pour une faible utilisation et le rouge pour une surutilisation potentielle ou des fuites de mémoire.

3. **CPU Usage** : représente la charge du CPU dans le temps pour des services individuels, l'intensité de la couleur indiquant le niveau d'utilisation.

Type de graphique : Carte thermique. Ce type de visualisation permet de montrer l'intensité de l'utilisation de l'unité centrale pour différents services et intervalles de temps, ce qui permet d'identifier des modèles ou des points chauds de forte activité.

Choix de Couleurs : Spectre allant du blanc au rouge. Dans ce contexte, le blanc représente probablement une utilisation faible ou nulle du CPU, et le spectre passe au rouge lorsque l'utilisation s'intensifie.

4. Avg Request Processing Time in ms

Mesure le temps moyen de traitement des requêtes par le système, un indicateur critique de la réactivité du système.

Type de graphique : Il s'agit d'un panneau de statistiques qui affiche directement un chiffre clé dans un format facile à digérer, vu qu'on a une seule donnée d'importance capitale.

Métriques métier :

5. **Number of HTTP Calls** Représente la charge de trafic sur le système, mesurée par les requêtes HTTP.

Type de graphique : Graphique linéaire. Un choix évident pour afficher les tendances du volume dans le temps et détecter les périodes de forte demande ou d'augmentation de la demande.

6. **Client Registration Count in PostgreSQL DB** : trace le nombre de nouveaux clients créés dans la base de données PostgreSQL du système.

Type de graphique : Graphique linéaire. Efficace pour montrer les actions cumulées dans le temps, telles que les enregistrements, et pour mettre en évidence les tendances de croissance.

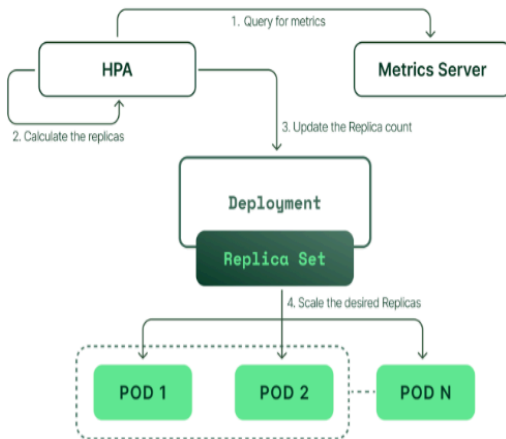
7. **Client Visit Count in Redis DB** : surveille le nombre de visites des sites clients, telles qu'elles sont enregistrées dans la base de données Redis.

Type de graphique : Graphique linéaire. Il est intéressant de montrer les tendances des visites des sites web des clients au fil du temps.

TP 4

A. HPA

1. Explication du mécanisme HPA :



L'Autoscaler Horizontal de Pods (HPA) de Kubernetes ajuste automatiquement le nombre de pods en fonction de la charge de travail, en particulier en réponse aux modèles d'utilisation du CPU. Il réalise une mise à l'échelle horizontale en ajoutant ou supprimant des instances de pods dans des Déploiements, Contrôleurs de Réplication, StatefulSets ou ReplicaSets, basé sur des métriques comme l'utilisation moyenne du CPU. Fonctionnant en continu, l'HPA surveille les métriques des pods, calcule le nombre optimal de réplicas, et ajuste automatiquement entre un nombre minimum et maximum prédéfini.

Ceci assure une adaptation dynamique aux fluctuations prévisibles des besoins de charge de travail, optimisant l'utilisation des ressources.

2. Implémentation du HPA :

```
{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "polymetrie-chart.fullname" . }}
  labels:
    {{- include "polymetrie-chart.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "polymetrie-chart.fullname" . }}
  minReplicas: {{ .Values.autoscaling.minReplicas }}
  maxReplicas: {{ .Values.autoscaling.maxReplicas }}
  metrics:
    {{- if .Values.autoscaling.targetCPUUtilizationPercentage }}
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: {{ .Values.autoscaling.targetCPUUtilizationPercentage }}
    {{- end }}
    {{- if .Values.autoscaling.targetMemoryUtilizationPercentage }}
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: {{ .Values.autoscaling.targetMemoryUtilizationPercentage }}
    {{- end }}
  {{- end }}
```

- **Explication :**

- **Activation de l'Autoscaling :** La condition ``if .Values.autoscaling.enabled`` sert de commutateur pour activer ou désactiver l'autoscaling. Si ``autoscaling.enabled`` est ``true``, le code de l'HorizontalPodAutoscaler (HPA) est pris en compte lors du déploiement, activant ainsi l'autoscaling. En revanche, si la valeur est ``false``, l'autoscaling est désactivé.
- **Référence à la cible de mise à l'échelle :** La section `scaleTargetRef` spécifie le type de ressource à mettre à l'échelle. Dans ce cas, il s'agit d'un déploiement (`apiVersion: apps/v1, kind: Deployment`) portant le même nom que l'application.
- **Définition des bornes de mise à l'échelle :** Les valeurs minimales et maximales du nombre de réplicas sont définies par les variables `minReplicas` et `maxReplicas` dans les valeurs du déploiement.
- **Métriques utilisées pour la mise à l'échelle :** Les métriques utilisées pour la mise à l'échelle sont définies en fonction des pourcentages d'utilisation cible du CPU et de la mémoire. Ces valeurs sont extraites des valeurs du déploiement
- **Métrique d'utilisation du CPU :** La première condition `if .Values.autoscaling.targetCPUUtilizationPercentage` vérifie si un pourcentage d'utilisation cible du CPU est spécifié dans les valeurs du déploiement. Si c'est le cas, une métrique de type Ressource est définie pour le CPU. L'objectif est d'atteindre une utilisation moyenne du CPU équivalente à la valeur spécifiée dans `targetCPUUtilizationPercentage`. L'HPA ajustera automatiquement le nombre de réplicas pour maintenir l'utilisation du CPU à ce pourcentage cible.
- **Métrique d'utilisation de la mémoire :** La deuxième condition `if .Values.autoscaling.targetMemoryUtilizationPercentage` vérifie si un pourcentage d'utilisation cible de la mémoire est spécifié dans les valeurs du déploiement. Si c'est le cas, une métrique de type Resource est définie pour la mémoire. L'objectif est d'atteindre une utilisation moyenne de la mémoire équivalente à la valeur spécifiée dans `targetMemoryUtilizationPercentage`. L'HPA ajustera automatiquement le nombre de réplicas pour maintenir l'utilisation de la mémoire à ce pourcentage cible.

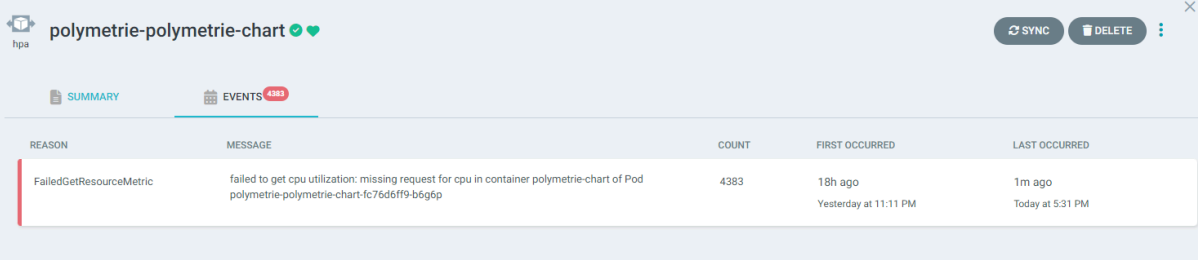
En définissant les valeurs dans un fichier de configuration externe (`values.yaml`), la gestion de l'autoscaling est simplifiée, permettant d'ajuster le comportement sans toucher au code, améliorant ainsi la maintenabilité et la collaboration entre équipes

```
autoscaling:
  enabled: true
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 80
```

Dans notre configuration spécifique, nous avons défini une politique de HPA avec une configuration minimale d'une réplica, une configuration maximale de 10 réplicas, et une cible d'utilisation du CPU fixée à 80%. Nous n'avons pas spécifié de pourcentage d'utilisation cible de la mémoire, car notre application ne consomme pas beaucoup de mémoire, même en cas de pic de charge.

3. Les problèmes rencontrés

Lorsque nous avons déployé notre HPA, nous avons rencontré une erreur qui a mis en lumière notre oubli de spécifier les ressources dans la configuration de notre pod (deployment). Il est essentiel de définir les demandes de ressources, telles que le CPU et la mémoire, car elles sont cruciales pour évaluer l'utilisation des ressources. Le contrôleur HPA exploite ces informations pour ajuster dynamiquement la mise à l'échelle de la cible vers le haut ou vers le bas en fonction des besoins de la charge de travail.



REASON	MESSAGE	COUNT	FIRST OCCURRED	LAST OCCURRED
FailedGetResourceMetric	failed to get cpu utilization: missing request for cpu in container polymetrie-chart of Pod polymetrie-polymetrie-chart-fc76d6ff9-b6g6p	4383	18h ago Yesterday at 11:11 PM	1m ago Today at 5:31 PM

Actuellement, les demandes de ressources sont configurées à 100 milliCPU et 128 MiB de mémoire, tandis que la limite de mémoire est fixée à 128MiB.

```
resources:
  limits:
    memory: 128Mi
  requests:
    cpu: 100m
    memory: 128Mi
```

Nous avons décidé de ne pas mettre une limite sur le CPU pour éviter des comportements inattendus lors de pics d'activité.

En ce qui concerne la mémoire, nous avons spécifié des limites de mémoire pour éviter que les conteneurs ne consomment des quantités excessives de mémoire, ce qui pourrait entraîner des problèmes de performances ou d'épuisement des ressources.

Remarque: Kubernetes définit les limites (limits) comme la quantité maximale de ressources qu'un conteneur peut utiliser. D'autre part, les demandes (requests) représentent la quantité minimale garantie de ressources réservée pour un conteneur.

B. Évaluation des ressources

Nous avons choisi k6 en tant qu'outil de test de charge en raison de sa performance, de sa simplicité d'utilisation avec JavaScript, de sa polyvalence pour différents protocoles, et de son intégration transparente avec Kubernetes via le k6-operator. Notre préférence pour k6 s'est également basée sur notre expérience préalable avec cet outil, facilitant ainsi son intégration dans notre environnement.

Nous avons installé le k6-operator dans le namespace “k6-operator-system” en suivant les étapes ci-dessous :

```
helm repo add grafana https://grafana.github.io/helm-charts  
helm repo update  
helm install k6-operator grafana/k6-operator
```

Pour procéder au test de charge, nous avons créé notre scénario en typescript, nous l'avons passé en tant que configmap, et nous avons démarré notre déploiement.

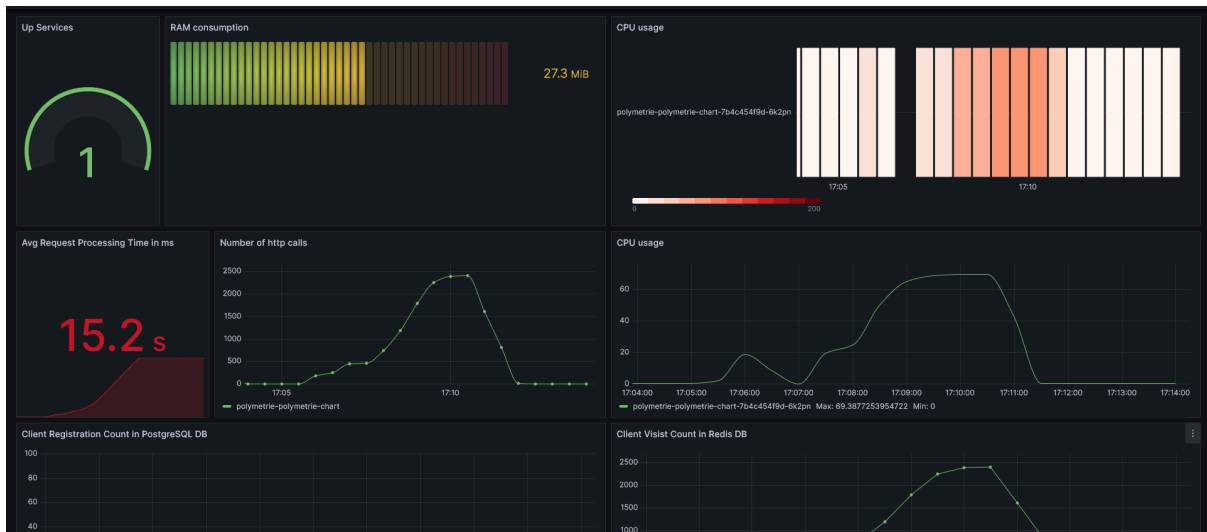
```
yimeng@131:~/Documents/orchestration-at-scale-23-24/polymetrie-g (main)$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
k6-sample-1-cjps7	1/1	Running	0	3m41s
k6-sample-2-zfhd4	1/1	Running	0	3m41s
k6-sample-3-zst7d	1/1	Running	0	3m41s
k6-sample-4-828vl	1/1	Running	0	3m41s
k6-sample-initializer-n2twm	0/1	Completed	0	3m46s
k6-sample-starter-mlng4	0/1	Completed	0	3m5s
my-postgresql-0	1/1	Running	1 (56d ago)	56d
my-redis-master-0	1/1	Running	0	57d
my-redis-replicas-0	1/1	Running	0	57d
my-redis-replicas-1	1/1	Running	0	57d
my-redis-replicas-2	1/1	Running	0	57d
polymetrie-polymetrie-chart-7b4c454f9d-fz5cc	1/1	Running	0	4m50s

L'opérateur k6 assume un rôle de gestionnaire essentiel lorsqu'il détecte un déploiement de test de charge k6. En réaction, il coordonne la création d'un initialisateur k6, un composant dédié à l'orchestration des pods de test. Ces derniers sont minutieusement configurés pour exécuter les scénarios de test spécifiés, induisant une charge significative sur l'application cible.

Pour évaluer les performances de notre application, nous avons initié le processus en exécutant des tests de charge avec k6, simulant ainsi une charge normale de requêtes par seconde.

Au début du processus, l'application présente une utilisation initiale minimale du CPU, se situant autour de 1 millième, tandis que la consommation de mémoire est d'environ 26 mégaoctets. Dans des conditions générales, au cours d'un scénario typique, nous avons constaté que l'application requiert environ 80 millièmes de CPU et 57mb de mémoire. Ces chiffres fournissent une indication précise de la configuration nécessaire en termes de demande de ressources CPU et mémoire pour un pod.



En effectuant des tests de référence sur les limites de notre application, nous avons remarqué que la mémoire n'augmente jamais de manière significative, tandis que le processeur augmente rapidement. Selon nos tests, nous avons mis en place un HPA (Horizontal Pod Autoscaler) pour scheduler de nouveaux pods à mesure que la charge augmente, et nous l'avons configuré pour réagir à l'utilisation du processeur. Lorsque celle-ci dépasse 70 % en moyenne sur le déploiement, un nouveau pod est planifié.

Nous avons décidé de définir notre demande CPU à 100mi, non seulement pour fournir une quantité minimale de CPU pour un fonctionnement optimal, mais aussi pour avoir une marge permettant de gérer les demandes avant la création d'un nouveau pod par notre HPA. Nous avons programmé nos pods pour s'exécuter sur le nœud avec le plus de ressources disponibles (affinity), et nous avons simulé une charge de 1600 req/s, qui est notre maximum avant que l'HPA ne puisse plus planifier de nouveaux pods faute de ressources. Nous avons donc configuré le nombre maximum de pods autorisés par l'HPA à 7 en fonction de cette charge maximale.

TD5

A. Exploration du stack ELK

1. Installation

Nous avons installé l'opérateur ECK à l'aide de l'outil Helm. Ensuite, nous devons déployer le cluster ECK en utilisant le CRD Elasticsearch. En raison de ressources limitées, nous avons déployé un seul nœud qui remplit les rôles de master, data et data-warm.

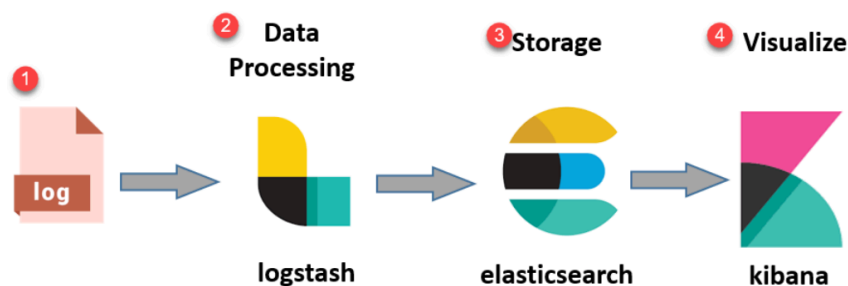
```
$ kubectl get elasticsearch -n elastic-cluster
NAME          HEALTH    NODES   VERSION   PHASE    AGE
elastic-cluster green     1       8.6.2     Ready    5m59s
```

Ensuite nous avons installé kibana dashboard dans le même namespace

```
$ kubectl get kibana -n elastic-cluster
NAME    HEALTH    NODES   VERSION   AGE
kibana  green     1       8.6.2     2m55s
```

2. Explication

- **À quoi correspond chaque brique de l'acronyme ELK ?**
Quelle est la fonctionnalité de chaque brique ?
- **Logstash (L):** un outil de collecte, de traitement et d'expédition de logs. Il prend en charge la collecte de logs à partir de différentes sources, les traite en fonction des filtres spécifiés, et les envoie vers des destinations spécifiques. Logstash permet de normaliser les logs, de les enrichir et de les rendre compatibles avec Elasticsearch.
- **Elasticsearch (E):** un moteur de recherche et d'analyse de données distribué. Il est conçu pour indexer, stocker et rechercher des données, qu'elles soient structurées ou non. Dans le contexte de la stack ELK, Elasticsearch est utilisé pour stocker les logs de manière scalable et permettre des recherches rapides sur ces logs.
- **Kibana (K) :** une interface web de visualisation et d'analyse de données. Elle permet aux utilisateurs de créer des tableaux de bord interactifs, des graphiques et des visualisations basés sur les données stockées dans Elasticsearch. Kibana facilite la compréhension et l'exploration des logs, offrant une interface conviviale pour l'analyse des tendances, la recherche de patterns, et la création de rapports.



- Dans l'article, une de ces briques est remplacée par une alternative. Laquelle ? Ces deux briques servent-elles totalement le même but ?

Dans l'article, Logstash est remplacé par Filebeat.

Filebeat et Logstash sont deux outils de gestion des logs dans l'écosystème ELK. Filebeat est léger et se concentre sur la collecte efficace des logs, les envoyant directement à Elasticsearch ou Logstash. Il a des capacités de traitement limitées mais offre des métriques intégrées pour le suivi de la performance.

Logstash, plus polyvalent, collecte des données de diverses sources et les traite avec des filtres complexes avant de les envoyer vers Elasticsearch ou d'autres destinations. Il propose une gamme étendue de filtres et des capacités avancées de traitement, mais nécessite plus de ressources.

Pour résumer, Filebeat convient à une collecte légère et efficace de logs ainsi qu'à une analyse simple, notamment depuis des sources standard comme syslog. Logstash est préférable pour des transformations complexes sur des logs non structurés, la gestion de sources diverses comme les bases de données, et des opérations avancées de traitement des données.

- Lors de la configuration du cluster elasticsearch, l'auteur indique qu'il est nécessaire de "modifier le paramètre noyau `vm.max_map_count`". Expliquez, simplement, le fonctionnement d'Elasticsearch nécessitant ce réglage

La valeur par défaut de `vm.max_map_count` sur de nombreux systèmes est de 65536. Cependant, il est recommandé de la définir à au moins **262144** pour éviter les exceptions de mémoire insuffisante. Cela s'explique par le fait que Elasticsearch utilise un grand nombre de descripteurs de fichiers et de zones mappées en mémoire, surtout dans le cas de clusters plus importants. Si le nombre de zones mappées en mémoire est insuffisant, Elasticsearch peut échouer au démarrage ou crasher pendant son fonctionnement.

La commande suivante, définie dans notre le CRD Elasticsearch, permet de fixer la valeur de `vm.max_map_count` à 262144 :

```
command: ['sh', '-c', 'sysctl -w vm.max_map_count=262144']
```

- Lors de la configuration de kibana, il indique qu'il est impératif que "elasticsearchRef.name" ait le même nom que l'instance elasticsearch. À quoi peut bien servir cette clé ?

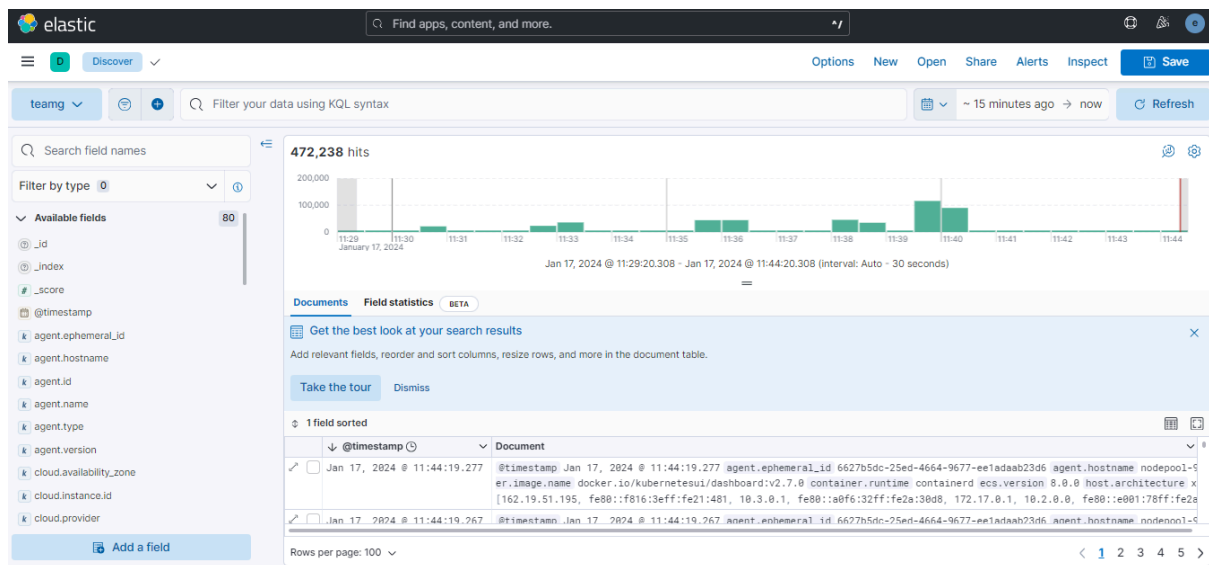
La clé "elasticsearchRef.name" dans la configuration de Kibana sert à spécifier le nom de l'instance Elasticsearch à laquelle Kibana doit se connecter. Il assure l'association correcte entre les deux composants, permettant à Kibana de localiser et de se connecter à la bonne instance Elasticsearch. La synchronisation des noms est donc essentielle pour garantir une connexion appropriée.

- Expliquez le contenu de la variable d'environnement "ELASTICSEARCH_HOST" dans l'étape de configuration de filebeat.

La variable d'environnement "ELASTICSEARCH_HOST" dans la configuration de Filebeat spécifie l'adresse du service http du cluster Elasticsearch, le mapping se fait à l'aide du nom du service `elastic-cluster-es-http` et le namespace `elastic-cluster`. D'où le host `elastic-cluster-es-http.elastic-cluster.SVC`. Les logs sont envoyés au cluster en communiquant avec ce service.

- **Après configuration de la stack, avez-vous identifié dans les logs collectés la région sur laquelle votre cluster est déployé ? Fournir un screenshot complet du log en question.**

Nous avons mis en place un ingress pour accéder à notre UI Kibana. En suivant le tutoriel, nous avons mis en place une dataview pour récupérer les logs de tout notre cluster. Dans la figure suivante, nous visualisons tous les logs du cluster.



En cherchant le label `kubernetes.node.labels.beta_kubernetes_io/os`, nous trouvons bien que la région sur laquelle le cluster est déployé est la région GRA7. Cette région correspond à la ville Gravelines.

<code>kubernetes.node.labels.beta_kubernetes_io/os</code>	<code>linux</code>
<code>kubernetes.node.labels.failure-domain_beta_kubernetes_io/region</code>	<code>GRA7</code>
<code>kubernetes.node.labels.failure-domain_beta_kubernetes_io/zone</code>	<code>nova</code>
<code>kubernetes.node.labels.kubernetes_io/arch</code>	<code>amd64</code>

```

5af5"
},
"kubernetes.node.labels.failure-domain_beta_kubernetes_io/region": [
  "GRA7"
],
"kubernetes.node.hostname": [
  "nodepool1-9b179ca9-c791-4b38-ad-node-ffc-ff"
]

```

Copy to clipboard

3. Les problèmes rencontrés

- 1) Pendant le déploiement des tests de charge, Elasticsearch a rencontré un dysfonctionnement. L'analyse des journaux a révélé un dépassement de la capacité de stockage.

```
elasticsearch {"@timestamp":"2024-01-23T21:00:08.173Z", "log.level": "INFO", "message":"using [1] data paths, mounts [[/usr/share/elasticsearch/data (/dev/sdc)]], net usable_space [0b], net total_space [9.7gb], types [ext4]", "ecs.v
```

Bien que nous ayons initialement attribué 10 Gi à ElasticSearch, les tests de charge ont généré une quantité importante de journaux ce qui a conduit kubelet à mettre fin au pod. En réponse, nous avons augmenté la capacité de stockage à 15 Gi.

- 2) Un autre problème identifié c'est lors du redémarrage du cluster elasticSearch. Dans ce cas, l'opérateur génère un nouveau secret avec un nouveau mot de passe pour s'authentifier au cluster. Puisque nous avons installé filebeat avec des fichiers yml statiques, nous devrions récupérer le nouveau mot de passe et le changer dans le déploiement de filebeat. Cette opération est nécessaire parce que Filebeat a besoin des credentials pour récupérer les logs depuis le cluster elasticSearch.

Cette instruction permet d'extraire le mot de passe contenu dans le secret "**elastic-cluster-es-elastic-user**" à l'aide de la commande suivante :

```
kubectl get secret elastic-cluster-es-elastic-user -o go-template='{{.data.elastic | base64decode}}' -n elastic-cluster
```

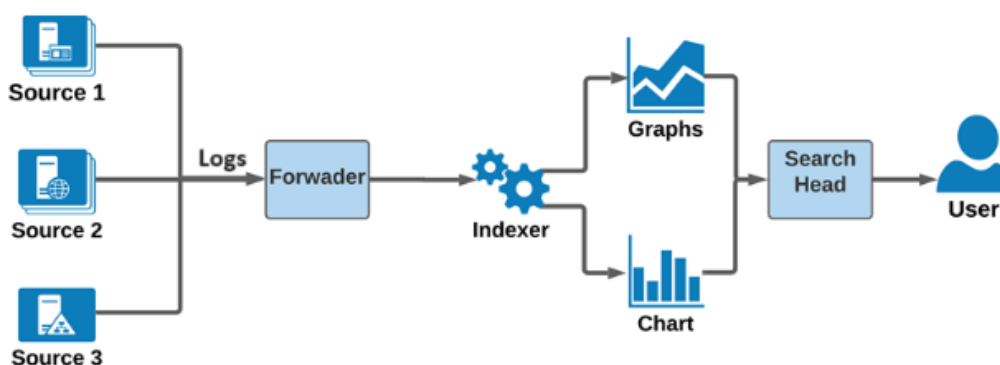
Le username correspondant est **elastic**. Vous pouvez accéder à l'interface en utilisant l'URL suivante :

<https://kibana.orch-team-g.pns-projects.fr.eu.org/login?next=%2Fapp%2Fhome>.

B. Comparaison Splunk VS Kibana

Ce TD nous a encouragé à explorer d'autres technologies de gestion des logs, et nous avons identifié **Splunk** comme une option intéressante.

Splunk se concentre sur l'analyse approfondie des journaux, facilitant l'exploration, la recherche et la compréhension des données de journal pour le dépannage et la détection d'anomalies.



Les trois composants essentiels de Splunk sont :

- **Forwarder** : Injecte les données et les transporte vers l'Indexeur.
- **Indexer** : Stocke et indexe les données, et répond aux demandes de recherche.
- **Search Head** : L'interface où ces composants sont regroupés ou répartis sur plusieurs serveurs.

Nous avons relevé quelques différences significatives entre ELK (Elasticsearch-Logstash-Kibana) et Splunk :

- **Configuration et maintenance** : Elasticsearch nécessite une installation sur chaque serveur du cluster, suivi de la configuration des paramètres réseau, de la sécurité et de la définition des rôles des serveurs. Une fois le cluster établi, Logstash et Kibana peuvent être utilisés pour ingérer et visualiser les données.

Splunk, quant à lui, s'installe plus simplement sur un seul serveur avec un assistant guidant la spécification des répertoires et la création de comptes utilisateurs. L'interface web de Splunk permet ensuite de commencer l'ingestion et l'analyse des données.

Les deux prennent en charge le déploiement sur site et en tant que service (SaaS),

- **Stockage** : Splunk utilise des index de fichiers pour stocker les données, tandis qu'Elasticsearch stocke des documents JSON non structurés, nécessitant plus d'espace. Ce choix de stockage peut entraîner des problèmes, notamment une consommation importante d'espace disque, surtout dans des environnements ayant des ressources limitées.
- **Langage de requête** : Kibana utilise la syntaxe Lucene, tandis que Splunk a son propre langage (SPL), plus complexe.
- **Indexation** : Dans Elasticsearch, les indexes utilisent la structure de données de l'index inversé pour permettre des correspondances rapides, tandis que dans Splunk, l'indexage est effectué par le composant d'indexation, analysant et organisant les journaux.

Un avantage clé de l'indexeur Splunk réside dans le stockage de plusieurs copies des données, minimisant ainsi le risque de perte d'informations.

- **Réplication des données** : Splunk assure la réplication des données avec plusieurs copies, éliminant le risque de perte.
- **Débogage** : Kibana ne propose pas de fonction de débogage, tandis que Splunk offre un support de débogage et de dépannage en cas d'échec pour trouver la cause fondamentale du problème.
- **Sécurité** : En matière de sécurité, Splunk oriente son avenir autour de sa plateforme de sécurité unifiée basée sur le cloud. En revanche, Kibana ne propose pas le même niveau de sécurité intégré.

Bibliographie

[K8s commands](#)

[K8s StoraKubectI Reference Docsge class, PV, and PV](#)

[Stack overflow - Service monitor](#)

[Splunk](#)

[CoreDNS docs](#)