Rapport
Déploiement et orchestration

Team A

(comme argocd)

kubernetes

Antoine BUQUET

-

Benoit GAUDET

-

Ayoub IMAMI

-

Mourad KARRAKCHOU

Table des matières :

I. TD1

A) Une version locale en docker-compose

1) Application

Nous avons choisi de développer notre application en Python à l'aide du framework Flask, car il nous permettait de lancer un serveur et d'exposer des endpoints facilement. L'application était alors déployée en local à l'aide d'un Dockerfile et d'un docker compose. Tous les besoins du sprint 1 étaient correctement fonctionnels :

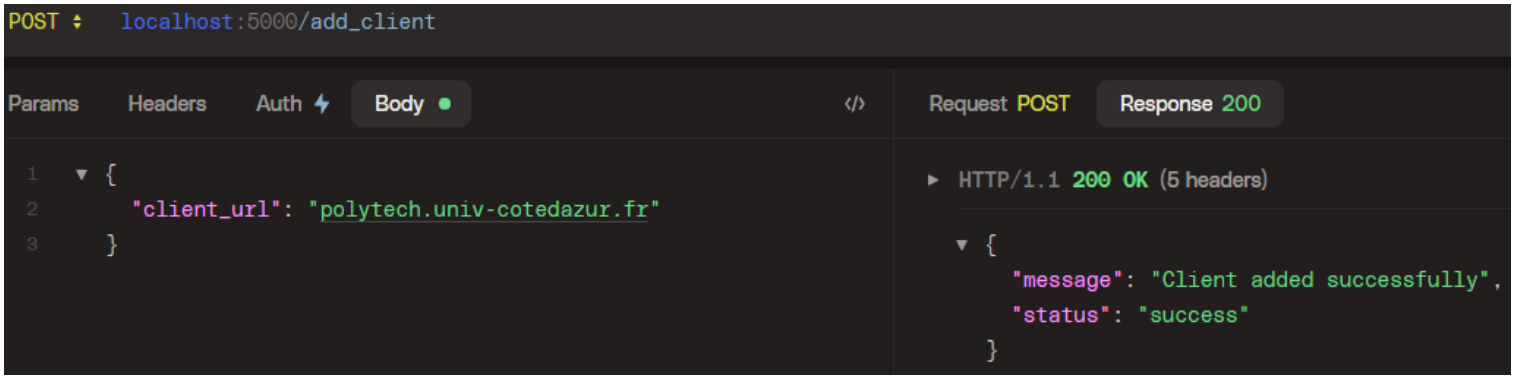

Figure I.A.1.1 : Ajout d'un client dans notre application

Figure I.A.1.2 : Consulter les clients de notre application

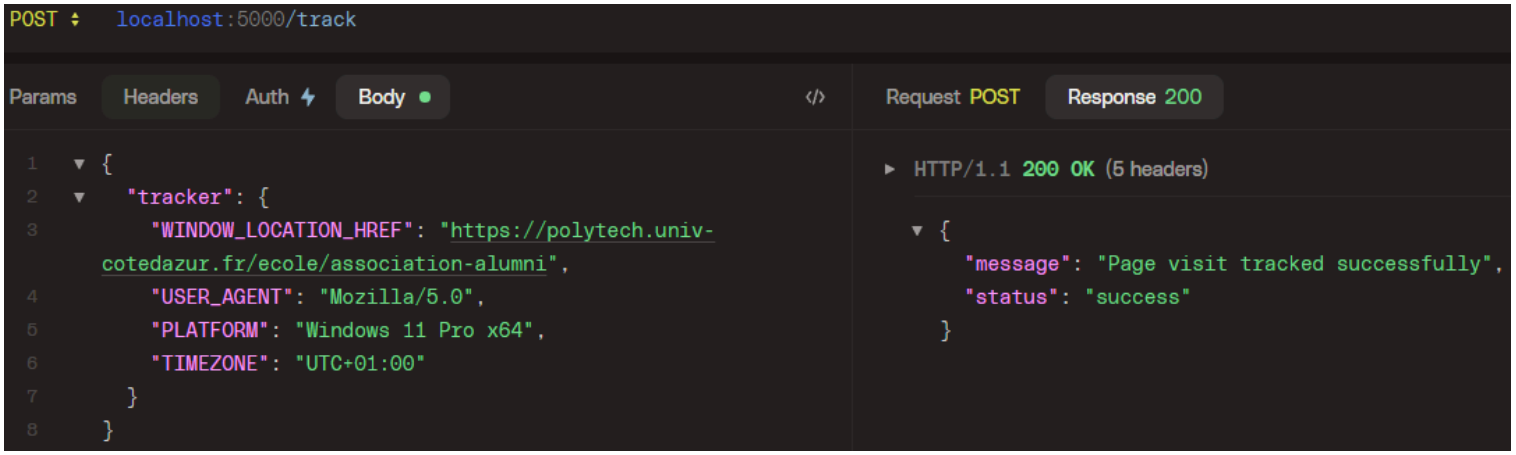

Figure I.A.1.3 : Tracker une page

Dans le cas où le client n'est pas connu, sa requête n'est pas prise en compte et il obtient une erreur.

Figure I.A.1.4 : Tracker une page d'un client non enregistré dans l'application

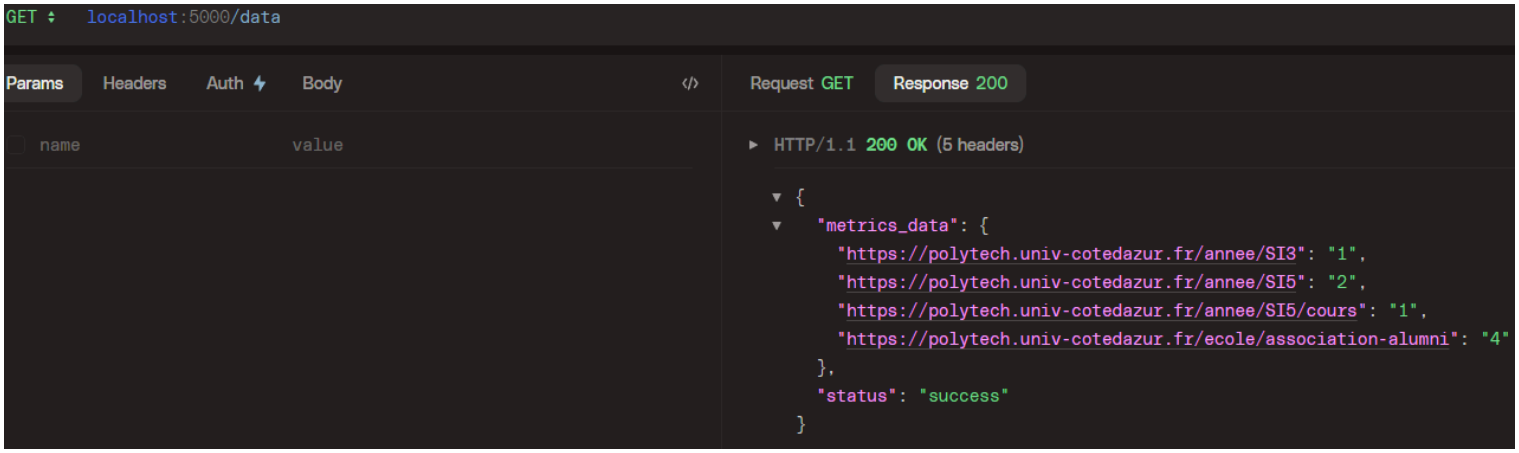

Figure I.A.1.5 : Consulter les données de tracking

2) Exposition d'un point /metrics

L'objectif était d'exposer un endpoint "/metrics" pouvant exposer des métriques intéressantes pour la suite. Notre projet étant en Flask, nous avons importé un Prometheus Exporter permettant d'avoir plusieurs métriques qui permettent de donner des indicateurs pour monitorer notre application. Nous avons implémenté des métriques plus système comme sur le CPU, la RAM et le temps de réponse, et des métriques permettant d'observer notre application pour un client donné. L'idée était de pouvoir avoir, à terme sur Grafana, l'information du nombre de requêtes qu'un client a eu et sur quel page de son site. Pour cela, on a mis en place un Gauger qui avait 2 champs comme on peut le voir sur la *Figure I.A.2.1* où le subscriber représente le client et l'url_page la page visitée. Cette métrique a été affinée lorsque nous avons travaillé sur Grafana, car le but était de pouvoir créer des statistiques sur les pages et les clients et donc cette façon de faire nous permettait d'avoir les 2.

```
website_visits{subscriber="test-k6-3.fr",url_page="https://test-k6-3.fr/tracking-test-v2"} 327621.0
website_visits{subscriber="polytech.univ-cotedazur.fr",url_page="https://polytech.univ-cotedazur.fr/antoine"} 5.0
website_visits{subscriber="test-k6-2.fr",url_page="https://test-k6-2.fr/tracking-test-v2"} 327644.0
website_visits{subscriber="polytech.univ-cotedazur.fr",url_page="https://polytech.univ-cotedazur.fr/mourad"} 22.0
```

Figure I.A.2.1 : Metric website_visits exposé sur le /metrics du polymetrie-service

B) En route vers Kube !

1) Configuration d'un environnement de travail nous permettant d'interagir avec Kubernetes

Lors de la première configuration de notre environnement de travail, nous constatons que nous avons déjà plusieurs ressources. Avec la commande "kubectl cluster-info", nous avons les adresses du plan de contrôle de Kubernetes, du CoreDNS et du serveur de métriques, sur OVH, comme illustré sur la *Figure I.B.1.1*. Le cluster livré est composé de 3 workers, comme le montre la *Figure I.B.1.2*. Chaque nodes utilisent différentes quantités de ressources CPU et RAM, comme l'indiquent les *Figures I.B.1.3, I.B.1.4 et I.B.1.5*. Enfin, des namespaces et des pods existants sont présents dans notre cluster, comme vous pouvez le voir sur les dernières *Figures I.B.1.6 et I.B.1.7*.

```
ntoine@antoine-3200:~$ kubectl cluster-info
Kubernetes control plane is running at https://we0ymc.c1.gra7.k8s.ovh.net
CoreDNS is running at https://we0ymc.c1.gra7.k8s.ovh.net/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://we0ymc.c1.gra7.k8s.ovh.net/api/v1/namespaces/kube-system/services/https:metrics-se
rver:/proxy

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

Figure I.B.1.1 : Résultat de la commande "kubectl cluster-info"

```
ntoine@antoine-3200:~$ kubectl get nodes
NAME                                          STATUS   ROLES    AGE   VERSION
nodepool-4b095459-9701-4946-ae-node-ac5ebf    Ready    <none>   8d    v1.27.4
nodepool-4b095459-9701-4946-ae-node-f67635    Ready    <none>   8d    v1.27.4
nodepool-4b095459-9701-4946-ae-node-f70186    Ready    <none>   8d    v1.27.4
```

Figure I.B.1.2 : Résultat de la commande “kubectl get nodes”

```
  Namespace                  Name                                       CPU Requests  CPU Limits  Memory Requests  Mem
ory Limits  Age
  ---------                  ----                                       ------------  ----------  ---------------  ---
----------  ---
  kube-system                calico-kube-controllers-654868b4c-6j92c    0 (0%)        0 (0%)      0 (0%)           0 (
0%)         8d
  kube-system                canal-2jbz6                                250m (13%)    0 (0%)      0 (0%)           0 (
0%)         8d
  kube-system                coredns-9dc5d84b6-xs2xp                    100m (5%)     0 (0%)      70Mi (1%)        170
Mi (3%)     8d
  kube-system                kube-dns-autoscaler-7944bcf59d-977lm       20m (1%)      0 (0%)      10Mi (0%)        0 (
0%)         8d
  kube-system                kube-proxy-mt6gl                           100m (5%)     0 (0%)      200Mi (3%)       200
Mi (3%)     8d
  kube-system                metrics-server-68fdc444fc-znh8x            100m (5%)     0 (0%)      200Mi (3%)       0 (
0%)         8d
  kube-system                wormhole-66hkz                             0 (0%)        0 (0%)      0 (0%)           0 (
0%)         8d
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                570m (30%)  0 (0%)
  memory             480Mi (9%)  370Mi (7%)
  ephemeral-storage  0 (0%)      0 (0%)
  hugepages-2Mi      0 (0%)      0 (0%)
Events:              <none>
```

Figure I.B.1.3 : Ressources du premier node

```
  Namespace                  Name                CPU Requests  CPU Limits  Memory Requests  Memory Limits  Age
  ---------                  ----                ------------  ----------  ---------------  -------------  ---
  kube-system                canal-n28pb         250m (13%)    0 (0%)      0 (0%)           0 (0%)         8d
  kube-system                kube-proxy-8jzjn    100m (5%)     0 (0%)      200Mi (3%)       200Mi (3%)     8d
  kube-system                wormhole-qsbwh      0 (0%)        0 (0%)      0 (0%)           0 (0%)         8d
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                350m (19%)  0 (0%)
  memory             200Mi (3%)  200Mi (3%)
  ephemeral-storage  0 (0%)      0 (0%)
  hugepages-2Mi      0 (0%)      0 (0%)
```

Figure I.B.1.4 : Ressources du deuxième node

```
Non-terminated Pods:          (4 in total)
  Namespace                   Name                      CPU Requests  CPU Limits  Memory Requests  Memory Limits  Age
  ---------                   ----                      ------------  ----------  ---------------  -------------  ---
  kube-system                 canal-x4nx8               250m (13%)    0 (0%)      0 (0%)           0 (0%)         8d
  kube-system                 coredns-9dc5d84b6-mxql6   100m (5%)     0 (0%)      70Mi (1%)        170Mi (3%)     8d
  kube-system                 kube-proxy-q6bpl          100m (5%)     0 (0%)      200Mi (3%)       200Mi (3%)     8d
  kube-system                 wormhole-fh4r2            0 (0%)        0 (0%)      0 (0%)           0 (0%)         8d
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                450m (24%)  0 (0%)
  memory             270Mi (5%)  370Mi (7%)
  ephemeral-storage  0 (0%)      0 (0%)
  hugepages-2Mi      0 (0%)      0 (0%)
```

Figure I.B.1.5 : Ressources du troisième node

```
ntoine@antoine-3200:~$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   8d
kube-node-lease   Active   8d
kube-public       Active   8d
kube-system       Active   8d
```

Figure I.B.1.6 : Namespaces déjà créés

```
ntoine@antoine-3200:~$ kubectl get pods -n kube-public
No resources found in kube-public namespace.
ntoine@antoine-3200:~$ kubectl get pods -n kube-node-lease
No resources found in kube-node-lease namespace.
ntoine@antoine-3200:~$ kubectl get pods -n default
No resources found in default namespace.
ntoine@antoine-3200:~$ kubectl get pods -n kube-system
NAME                                       READY   STATUS    RESTARTS   AGE
calico-kube-controllers-654868b4c-6j92c    1/1     Running   0          8d
canal-2jbz6                                2/2     Running   0          8d
canal-n28pb                                2/2     Running   0          8d
canal-x4nx8                                2/2     Running   0          8d
coredns-9dc5d84b6-mxql6                    1/1     Running   0          8d
coredns-9dc5d84b6-xs2xp                    1/1     Running   0          8d
kube-dns-autoscaler-7944bcf59d-977lm       1/1     Running   0          8d
kube-proxy-8jzjn                           1/1     Running   0          8d
kube-proxy-mt6gl                           1/1     Running   0          8d
kube-proxy-q6bpl                           1/1     Running   0          8d
metrics-server-68fdc444fc-znh8x            1/1     Running   0          8d
wormhole-66hkz                             1/1     Running   0          8d
wormhole-fh4r2                             1/1     Running   0          8d
wormhole-qsbwh                             1/1     Running   0          8d
```

Figure I.B.1.7 : Pods existants

2) Déploiement et accès à Kubernetes Dashboard

La commande kubectl proxy et kubectl port-forward sont toutes deux utilisées pour fournir un accès réseau à des services dans un cluster Kubernetes. Cependant, elles ont des fonctionnalités et des cas d'utilisation légèrement différents :

- kubectl proxy crée un serveur proxy HTTP local entre la machine locale et le cluster Kubernetes. Elle permet d'accéder aux API du cluster Kubernetes via ce proxy. Elle est notamment utilisée pour accéder aux interfaces web des services Kubernetes, notamment le dashboard Kubernetes.

- kubectl port-forward permet de rediriger les connexions réseau depuis un port local de la machine locale vers un port spécifique d'un pod dans le cluster Kubernetes. Elle est surtout utilisée pour rediriger le traffic entre le service d'un pod et la machine locale.

Après avoir exécuté la commande kubectl proxy, il est possible d'accéder au dashboard avec le lien suivant :

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

Pour s'identifier cependant, il faut avoir créé au préalable un ServiceAccount, un ClisterroleBinding et un Secret. Puis, il faut récupérer un token d'authentification avec la commande : kubectl -n kubernetes-dashboard create token admin-user

Lors de l'installation du dashboard Kubernetes, plusieurs ressources sont créées pour permettre le déploiement et le fonctionnement de l'application. On peut voir dans la Figure *I.B.2.1* deux services. Le service dashboard-metrics-scraper permettant de récupérer les informations du cluster et le service kubernetes-dashboard permettant de proposer une interface graphique pour l'utilisateur affichant les données du cluster.

```
benoit@benoit-VM:~$ kubectl get all -n kubernetes-dashboard
NAME                                            READY   STATUS    RESTARTS   AGE
pod/dashboard-metrics-scraper-5cb4f4bb9c-925wt   1/1     Running   0          8d
pod/kubernetes-dashboard-6967859bff-nxg4t        1/1     Running   0          8d

NAME                                TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/dashboard-metrics-scraper   ClusterIP   10.3.88.202    <none>        8000/TCP   8d
service/kubernetes-dashboard        ClusterIP   10.3.24.182    <none>        443/TCP    8d

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/dashboard-metrics-scraper   1/1     1            1           8d
deployment.apps/kubernetes-dashboard        1/1     1            1           8d

NAME                                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/dashboard-metrics-scraper-5cb4f4bb9c   1         1         1       8d
replicaset.apps/kubernetes-dashboard-6967859bff        1         1         1       8d
```

Figure I.B.2.1 : Ressources créées lors du déploiement du dashboard Kubernetes

3) Déploiement de Polymétrie sur Kubernetes à l'aide d'un ensemble de fichiers YAML "à plat".

Dans un premier temps, nous installons Redis et PostgreSQL en utilisant Helm des charts Helm du dépôt Bitnami. Puis, nous avons build l'image Docker avec le nom "polymetrie" et tag cette image en *"leolebossducloud/polymetrie:latest"* avant de la push sur le registre d'image DockerHub.

Une fois l'image sur DockerHub, il faut déployer notre application grâce au fichier polymetrie-deployment.yaml. En accédant au dashboard Kubernetes, nous pouvons observer l'avancement du déploiement de Polymétrie.

De plus, nous avons le fichier polymetrie-service.yaml qui nous permet d'accéder à nos pods en interne et un polymetrie-ingress.yaml qui nous permet d'exposer notre service Polymétrie à l'extérieur.

Nos fichiers YAML sont utilisés pour la gestion des ressources au sein de notre cluster Kubernetes. Nous utilisons cette approche déclarative, où nous spécifions l'état désiré du système plutôt que de donner des instructions impératives sur la manière d'atteindre cet état. Nous décrivons simplement l'état souhaité de cette ressource dans un fichier YAML. Cela offre plusieurs avantages, dont la clarté, la reproductibilité et la facilité d'automatisation. Cela permet aussi une meilleure compréhension et une représentation visuelle de l'infrastructure et des applications déployées.

L'approche déclarative offre plusieurs avantages :

- Reproductibilité : En déclarant l'état souhaité, nous pouvons reproduire facilement le même environnement ou la même configuration sur différents clusters ou dans différentes phases de développement.
- Versioning : Les fichiers YAML peuvent être versionnés dans un système de contrôle de source (qu'on a mit en place avec GitHub), ce qui permet de suivre les changements au fil du temps et de revenir à des versions antérieures si nécessaire.
- Collaboration : Les fichiers YAML peuvent être partagés et compris naturellement entre nous.
- Automatisation : L'approche déclarative facilite l'automatisation des déploiements, des mises à jour et de la gestion des ressources, ce qui contribue à l'efficacité opérationnelle.

C) Industrialisation avec Ansible

Jusqu'à maintenant, nous avions déployé tous les composants de notre application à la main. Cependant, le déploiement manuel peut entraîner des erreurs humaines, des configurations incohérentes et des processus non reproductibles. Il est plus sujet aux oublis, ce qui peut causer des problèmes de stabilité et de sécurité. L'approche manuelle prend également plus de temps. Des solutions à ces problèmes existent et Ansible en fait partie. Il s'agit d'un outil d'automatisation open-source permettant la gestion de configuration, le déploiement d'applications et l'orchestration de tâches sur des infrastructures informatiques. Ansible automatise les tâches de déploiement, garantissant la reproductibilité et la cohérence. Il facilite par ailleurs la maintenance grâce à des scripts déclaratifs. L'utiliser fait économiser du temps et réduit les erreurs humaines comparé au déploiement manuel.

L'automatisation d'Ansible se fait à travers un *ansible-playbook*. Notre playbook est composé de plusieurs tâches qui permettent de déployer séquentiellement tout ce qui est nécessaire à l'application. Dans notre cas, il a fallu que le playbook ajoute le référentiel de charts Helm de Bitnami, puis déploie grâce à ce référentiel les bases de données PostgreSQL et Redis. Enfin, il ne restait plus qu'à appliquer au cluster Kubernetes nos différents fichiers yaml précédemment écrits pour les ressources de notre application.

```
- name: Deploy Polymetrie Application
  hosts: localhost

  tasks:
    - name: Add the Bitnami Chart Repository
      kubernetes.core.helm_repository:
        name: bitnami
        repo_url: "https://charts.bitnami.com/bitnami"
```

```yaml
- name: Apply PostgreSQL Database
  kubernetes.core.helm:
    name: postgresql
    chart_ref: bitnami/postgresql
    release_namespace: default
    values_files:
      - ../kubernetes/postgresql-values.yaml
    state: present

- name: Apply Redis Database
  kubernetes.core.helm:
    name: redis
    chart_ref: bitnami/redis
    release_namespace: default
    values_files:
      - ../kubernetes/redis-values.yaml
    state: present

- name: Wait for PostgreSQL to be ready
  pause:
    seconds: 20

- name: Apply Polymetrie Deployment
  kubernetes.core.k8s:
    state: present
    src: ../kubernetes/polymetrie-deployment.yaml

- name: Apply Polymetrie Service
  kubernetes.core.k8s:
    state: present
    src: ../kubernetes/polymetrie-service.yaml

- name: Apply Polymetrie Ingress
  kubernetes.core.k8s:
    state: present
    src: ../kubernetes/polymetrie-ingress.yaml
```

Figure I.C.1 : Playbook Ansible pour déployer l'application Polymétrie

Comme on peut le remarquer sur la *Figure I.C.1*, une tâche de pause a été ajoutée pour attendre que la base de données PostgreSQL soit prête. Cela est dû au fait que Ansible exécute les tâches à la suite, tel un script sh par exemple, sans se soucier de l'état dans lequel se trouve la tâche actuelle pour passer à la tâche suivante. Nous avons donc été contraints d'ajouter cette étape afin de retarder le déploiement de notre application sinon la connexion à la base de données PostgreSQL ne pouvait pas s'effectuer.

Pour finir, nous avons choisi d'écrire plusieurs autres playbook Ansible pour déployer nos différents outils tels que le dashboard Kubernetes, K6 et la stack Prometheus/Grafana.

II. TD2

A) Un peu de terraforming

Jusqu'ici, nous avons été amenés à travailler avec Ansible, Terraform et Helm. Ces technologies ont chacune leur cas d'utilisation dans lesquelles elles seront meilleures qu'une autre ou complémentaires.

1) Terraform

Terraform est utile pour provisionner et gérer des ressources d'infrastructure sur le cloud. Sa particularité vient de la gestion des états des machines, permettant de pouvoir effectuer des actions en fonction des derniers états connus.
En effet, il a les avantages de connaître les états des anciens déploiements afin de ne pas faire d'étapes inutiles et d'assurer une certaine cohérence entre les déploiements dans des clouds différents grâce à la gestion des états. Il permet aussi de gérer des dépendances entre certaines ressources (possibilité de créer des ressources dans un ordre précis).

Bien que l'utilisation de Terraform automatise efficacement le provisionnement d'infrastructure, il faut gérer le partage des fichiers d'états lors d'un projet collaboratif. De plus, comme cet outil est spécialisé dans le déploiement d'infrastructures, il ne peut pas forcément effectuer d'autres tâches spécifiques.

2) Ansible

Ansible est un outil de scripting, souvent utilisé pour automatiser de la configuration logicielle et des tâches d'administration système. Par exemple, il peut être utilisé pour effectuer des actions sur des machines en SSH.

Grâce à sa grande flexibilité, il peut être utilisé pour un grand nombre d'applications, car il offre les possibilités d'un script. Les tâches sont exécutées dans l'ordre et il existe une large base de modules pour différents systèmes.

Malheureusement, il peut être moins efficace pour effectuer des tâches liées à l'infrastructure par rapport à Terraform. En effet, comme Ansible ne gère pas l'état des ressources, il effectuera "bêtement" toutes ses actions, même si elles n'ont pas besoin d'être effectuées. Cela peut aussi être gênant dans le cas où il est nécessaire d'attendre qu'une ressource soit déployée avant d'effectuer la suite des actions, ce qui obligera à effectuer des temps de pause explicites dans les étapes du script Ansible.

3) Helm

Helm permet d'installer et de mettre à jour des applications Kubernetes en automatisant les opérations de déploiements au niveau du cluster. C'est un gestionnaire de paquets pour Kubernetes.

Il propose des modèles de déploiement réutilisables avec l'utilisation et le partage de Chart, gestion des mises à jour et des rollbacks. Cela entraîne une simplification du déploiement d'applications, mais aussi son partage et sa réutilisation. Par contre, la gestion d'erreurs peut être un peu difficile et Helm ne permet pas toujours une personnalisation élevée.

Terraform est meilleur dans le provisionnement d'infrastructure grâce à sa gestion des états et Ansible est meilleur pour effectuer des actions plus spécifiques grâce à la liberté d'utilisation offerte. Ansible et Terraform implémentent souvent Helm afin de déployer des applications sur Kubernetes. Il peut être tout à fait justifiable de composer avec les trois technologies. Helm, Terraform et Ansible peuvent être utilisés dans un flux GitOps en utilisant des pipelines CI/CD dans le but de pouvoir automatiser encore plus les déploiements. Finalement, le choix entre Terraform, Ansible, Helm dépend du contexte et des exigences spécifiques du projet.

B) Un peu d'économie sur les LoadBalancers

La ressource Ingress nous permet d'accéder à nos services depuis l'extérieur du cluster. L'objectif est d'éviter la création d'un load balancer par service, mais seulement un par Ingress. Cela permet d'économiser les coûts nécessaires au déploiement d'un grand nombre de load balancer.

D'après la documentation proposée par OVHCloud, l'Ingress Controller de Kubernetes utilise un serveur web Nginx en tant que reverse proxy et load balancer.

Plus précisément, lors du déploiement et de l'exécution de l'Ingress Controller Nginx, nous avons un pod qui exécute Nginx et surveille le plan de contrôle Kubernetes afin de voir s'il y a de nouvelles ressources Ingress et des mises à jour.

De plus, nous disposons également d'un load balancer OVHcloud. Grâce à ce dernier, le trafic externe sera dirigé vers le pod de l'Ingress Controller exécutant Nginx, qui redirigera ensuite le trafic vers les services en backend appropriés que nous configurons dans les ressources Ingress.

Une fois le load balancer installé, nous disposons d'une EXTERNAL-IP que nous pouvons convertir en un nom de domaine pour obtenir un lien compréhensible plutôt qu'une simple adresse IP.

Avec cela et les différents Ingress et règles de routage, nous avons pu accéder aux services suivants depuis l'extérieur :

- Polymetrie
- ArgoCD
- Prometheus
- Grafana
- Kibana

De ce fait, nous n'avions pas besoin d'avoir accès à notre environnement Kubernetes pour travailler sur certaines tâches, telles que surveiller et agir sur l'état de notre travail via l'application d'ArgoCD. Et cela était plus pratique que de faire du "port forwarding" dès que nous voulions accéder à une UI.

C) Implémentation de GitOps avec ArgoCD

Nous avons voulu tester deux approches pour la mise en place d'une chaîne de construction automatique de nos images Docker : Jenkins et GitHub Actions. La principale différence, dans notre cas, est que nous devons déployer Jenkins sur notre cluster tandis que GitHub Actions peut se faire simplement depuis notre répertoire GitHub. L'installation et la configuration de Jenkins était plus coûteuse en temps et en ressources, nous avons donc décidé de continuer avec GitHub Actions.

Notre GitHub Action vérifie s'il y a des changements sur la branche "main" car c'est cette branche qui doit toujours avoir une version stable, nous travaillons sur d'autres branches pour conserver l'état stable de "main". S'il y a des changements, alors la GitHub Action va build notre image docker et la push sur DockerHub à l'aide d'un token pour s'identifier.

Figure II.C.1 : Étapes de la GitHub Action qui build et push notre image sur DockerHub

De plus, nous avons décidé de réunir tous nos fichiers de configurations dans un même sous-répertoire "/kubernetes" de notre répertoire principal, de cette façon, nous centralisons nos configurations et il est donc plus simple de les retrouver que de naviguer dans tout le répertoire, mais cela nous permet surtout de configurer

ArgoCD en lui indiquant le sous-répertoire à partir duquel il peut déployer nos ressources, comme vous pouvez le constater sur la *Figure II.C.2*.

Nous avons configuré le déploiement de l'instance ArgoCD de sorte à utiliser une ressource "Ingress". Et nous avons rencontré un problème, car la ressource était configurée pour y être accédé en HTTP, mais on accède à l'application d'ArgoCD en HTTPS par défaut. Il a donc fallu ajouter la configuration "server.insecure: "true"" dans le fichier configmap.yaml d'ArgoCD afin de pouvoir y accéder en HTTP.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: myapp-argo-application
  namespace: argocd
spec:
  project: default

  source:
    repoURL: https://github.com/pns-si5-cloud/orchestration-at-scale-23-24-polymetrie-a.git
    targetRevision: HEAD
    path: kubernetes
  destination:
    server: https://kubernetes.default.svc
    namespace: default

  syncPolicy:
    automated: {}
    syncOptions:
    - CreateNamespace=true
```

Figure II.C.2 : Configuration de l'application ArgoCD automatisant le déploiement

De plus, nous avons ajouté deux configurations dans "syncPolicy", qui permet de synchroniser automatiquement le déploiement en fonction du répertoire et la création automatique de namespaces si ces derniers n'existent pas déjà.

Depuis l'interface utilisateur, nous avons ajouté un token pour donner l'accès de notre répertoire Git à ArgoCD.

Nous avions également des configurations “autoHeal” et “autoPrune” qui permettaient de protéger notre cluster contre les mauvaises manipulations à la main afin de toujours conserver l’état stable du répertoire. Cependant, nous avons dû retirer ces configurations, car elles nous empêchaient de faire les tests de charge. En effet, lorsque nous modifions les configurations à la main et que nous faisons des tests de charge, ArgoCD détectait le changement et réappliquait l’état du répertoire. Nous avons retiré ces configurations assez tôt, et nous avions donc simplement des “outOfSync” indiquant que l’état du déploiement n’était pas le même que celui du répertoire.

Avec du recul, l’utilisation d’ArgoCD nous a facilité le développement, nous n’avions en effet plus besoin de tout déployer à la main, il fallait uniquement push sur “main”. En revanche, déployer à chaque push peut s’avérer redondant et pas forcément nécessaire, nous pouvons donc améliorer notre automatisation en faisant en sorte de déployer à l’aide de tags ou de pull requests.

III. TD3

A) Kube-Prom-Stack

La prochaine étape fut l’utilisation de la stack Prometheus. Le but de la stack Prometheus/Grafana était de pouvoir monitorer notre application. En effet, il est important de pouvoir avoir des informations et des métriques sur notre application afin de pouvoir monitorer les montées de charges, de pouvoir comprendre pourquoi un crash arrive à un certain moment et potentiellement pouvoir prévenir l’arrivée d’un tel crash en identifiant de potentiels symptômes.

L’aspect qui a été problématique dans l’implémentation de Prometheus était de faire en sorte qu’il trouve notre application. Tout d’abord, il fallait ajouter une nouvelle ressource Service Monitor qui était bien reliée avec le bon label à notre Deployment Polymetrie App, puis relier notre Prometheus à notre Service Monitor. Il faut bien faire attention à faire bien coïncider les différents labels pour que Prometheus trouve l’application.

Pour vérifier que tout a fonctionné, il est possible de retrouver “polymetrie-app” dans les targets trouvés par Prometheus, comme l’illustre la *Figure III.A.1.* On voit alors que le scraping se fait bien sur l’endpoint “/metrics” de polymetrie-app.

serviceMonitor/prometheus/polymetrie-app-monitor/0 (2/2 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.2.0.22:5000/metrics	UP	endpoint="polymetrie-port" instance="10.2.0.22:5000" job="polymetrie-service" namespace="default" pod="polymetrie-app-84f65c9bbf-2hjrp" service="polymetrie-service"	15.201s ago	12.424ms	
http://10.2.2.71:5000/metrics	UP	endpoint="polymetrie-port" instance="10.2.2.71:5000" job="polymetrie-service" namespace="default" pod="polymetrie-app-84f65c9bbf-gvwq5" service="polymetrie-service"	2.721s ago	7.854ms	

Figure III.A.1 : Onglet “target” de Prometheus montrant le scrapping de polymetrie-app

Il est ensuite possible d’avoir accès aux différentes métriques qu’expose notre application. On peut voir un exemple dans la *Figure III.A.2*. Cette commande nous permet d’avoir l’ensemble des routes d’un client et le nombre d'appels sur celles-ci. On y voit alors tout l’intérêt de Prometheus, car on peut maintenant observer les différentes métriques exposées dans l’application.

Figure III.A.2 : Requête sur Prometheus permettant de voir le nombre d'appels par page pour un client.

B) Un peu de Graphes

L'objectif ensuite était de pouvoir implémenter Grafana pour pouvoir organiser nos métriques dans un dashboard clair, ce qui nous permettra d'avoir une meilleure clarté et la capacité à monitorer notre application plus facilement.

Pour la création de notre dashboard, nous avons tout d'abord créé plusieurs rows comme dans la *Figure III.B.1*. L'objectif était de permettre une lecture simple de nos rows.

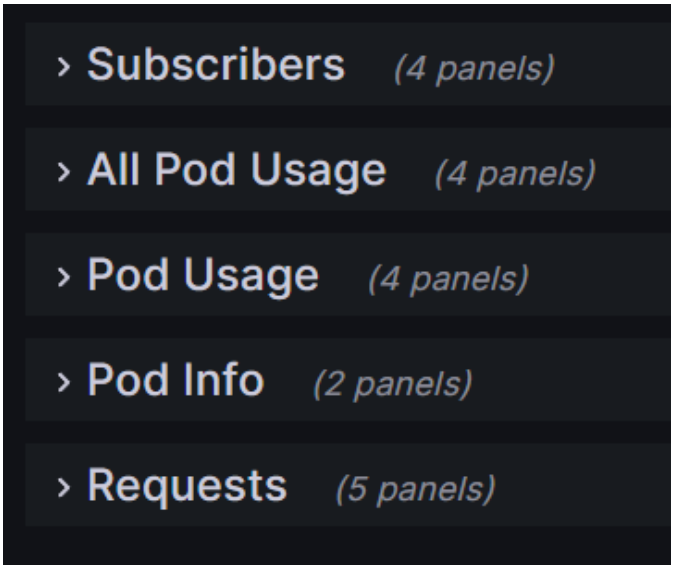

Figure III.B.1 : Rows du dashboard Grafana

Pour notre première row dans la *Figure III.B.2*, on a décidé de regrouper les métriques liées au client. Nous pouvons ainsi monitorer le nombre d'appels d'une page particulière ou d'un client en particulier ou encore avoir son nombre d'appels total. Cela permet en cas de problème sur un client de pouvoir plus simplement le monitorer et cela nous a servi notamment pendant nos tests de charge pour pouvoir observer les pics d'appel sur les routes trackées.

Figure III.B.2 : Row subscribers

En parallèle, notre 2eme row sur la *Figure III.B.3*, nous permet de voir l'utilisation de la mémoire et du CPU pour l'ensemble de nos pods qu'on retrouve à l'aide du container name. Nous avons décidé d'agencer nos row et de mettre celle-ci juste après la row métier pour qu'il soit possible de plus simplement observer que l'utilisation de nos containers est cohérente et proportionnelle aux nombres d'appels que reçoit l'application. On remarque alors que pendant le pic de charge, un nouveau pod a été créé grâce au HPA (en jaune) et ensuite, il disparait lorsque la charge est finie. Notre Dashboard est justement pensé pour pouvoir observer ces mécanismes qu'on a mis en place.

Figure III.B.2 : Row All Pod Usage

Notre 3eme row sur la *Figure III.B.3*, permet de décrire un pod en particulier. Cela est utile pour pouvoir choisir d'observer un pod particulier. On a alors des mesures particulières comme le pourcentage d'utilisation du pod en question. Le Pod CPU Usage Limit n'a par contre pas de données, cela s'explique le fait que nos pods n'ont pas de limite de ressources au niveau du CPU. On observe bien la charge au niveau du pod et on peut voir qu'on atteint pratiquement 45% de l'utilisation de la mémoire du pod.

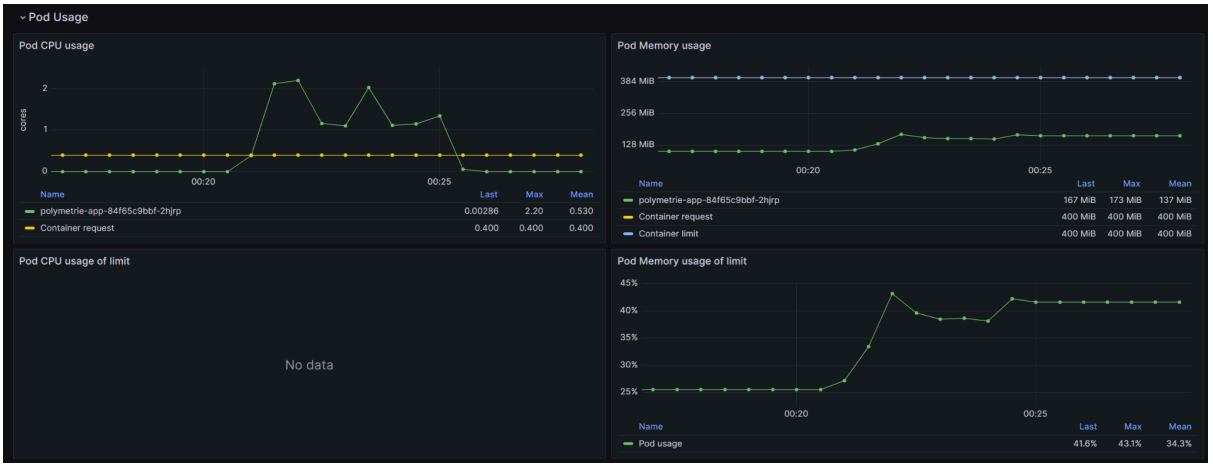

Figure III.B.3 : Row Pod Usage

Enfin la row Requests sur la *Figure III.B.4*; nous permet de voir le nombre d'appels par pod et au total. Cela permet de voir le nombre d'appels qui ont été faits sur notre application et donc de pouvoir bien vérifier que cela concorde avec le reste de nos indicateurs. On voit bien le pic correspondant au test de charges ici aussi, ce qui correspond bien aux résultats trouvés précédemment.

Figure III.B.4 : Row Requests

La création du dashboard grafana nous a pris du temps, car notre but était de créer un dashboard qui pouvait réellement permettre de monitorer un grand nombre d'indicateurs pour comprendre et décrire exactement l'état de notre application a un instant T. On a rencontré plusieurs difficultés. La plus récurrente était la manière de trouver et de traiter nos métriques (quelle unité prendre, quel call faire, quoi afficher dans la légende). Il fallait aussi réfléchir à la disposition de nos row, que ce soit le contenu de chaque row ou encore l'agencement des rows entre elles. En effet, chaque row devait nous permettre d'avoir les informations les plus pertinentes et qui peuvent potentiellement matcher avec les row adjacentes pour que lors de la lecture de nos graphes, on retrouve une cohérence. Avec du recul, nous pensons avoir réussi à mener à bien la création de dashboard. Nous l'avons surtout senti lors des tests de charges, car même en n'ayant aucune information sur un test de charge, à l'aide de nos dashboard, il était possible de retrouver l'ensemble des informations : le nombre d'appels total fait pendant le pic de charge, sur quel client, sur quelles routes, l'impacte que cela a eu sur l'utilisation du cpu et de la mémoire, si de nouveaux pods de polymetrie ont été déployés.

IV. TD4

A) HPA

La mise en place d'un Horizontal Pod Autoscaler va permettre à notre application d'absorber plus facilement des pics de charge, en déployant automatiquement, selon des règles configurables, de nouveau pod de notre application.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: polymetrie-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: polymetrie-app
  minReplicas: 2
  maxReplicas: 3
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 40
```

Figure IV.A.1 : Définition de notre ressource HPA

Comme visible sur la *Figure IV.A.1*, notre HPA surveille notre déploiement "polymetrie-app", et plus précisément la ressource CPU du pod. Son objectif est ici de garder l'utilisation moyenne des pods à 40%. Nous avons choisi 40% pour s'assurer que le pod ne va pas être trop surchargé et crash le temps qu'un autre pod soit déployé par le HPA. Enfin, notre HPA est configuré avec un nombre de réplica minimum de 2 et maximum de 3, mais il est possible d'augmenter le nombre maximal de réplicas dans la limite des ressources disponibles sur le cluster Kubernetes.

Un problème rencontré lors de cette phase était que le HPA, une fois déployé, indiquait toujours "<unknown>/40%" dans sa colonne "TARGET". Cela était dû au fait que nous n'avions pas spécifié initialement de ressources demandées pour le CPU.

B) On soulève un peu de fonte

Pour faire une campagne de benchmark sur notre application, nous avons dans un premier temps installer un opérateur K6 dans notre cluster Kubernetes afin que les nombreux appels qui vont être faits proviennent de l'intérieur du cluster et non une adresse IP externe qui pourrait alors s'apparenter à une attaque DDoS par exemple.

Ensuite, nous avons élaboré le scénario de test qui devait être le même pour chacun des benchmarks afin d'être cohérent. La route qui sera sans aucun doute la plus utilisée dans notre API est celle permettant de traquer une page. Voici ainsi le scénario utilisé pour notre campagne de benchmark :

Test de charge K6	
Chemin simulé	/track (avec payload)
Montée	0 à 1000 VUs en 30 secondes
Plafond	1000 VUs pendant 3 minutes
Descente	1000 à 0 VUs en 30 secondes

Figure IV.B.1 : Scénario utilisé pour les tests de charge.

Le but de cette campagne de benchmark était d'identifier les quantités de ressources idéales pour un seul pod de notre application. Il a fallu donc faire varier les demandes et limites en ressources CPU et RAM de notre application. Pour répertorier les résultats de chaque benchmark, nous avons créé un tableur[1] contenant les informations sur les ressources du pod, les résultats du test de charge K6, et enfin des métriques telles que l'utilisation du CPU et de la RAM ainsi que le nombre de requêtes traitées par secondes, obtenues à l'aide de notre dashboard Grafana.

Nous avons appliqué les conseils vu en cours, à savoir ne pas mettre de limite de ressource CPU et mettre la même quantité en demande et en limite pour la RAM. Enfin, pour déterminer quel benchmark était mieux qu'un autre, nous nous sommes basés sur le nombre de requêtes par secondes qu'avait pu traiter notre pod lors du test de charge.

[1] Résultats des benchmarks : https://docs.google.com/spreadsheets/d/1Wpcj3Y1A2L9_bM5e-8-swOU5prp_8114aTamhdgf7og/edit?usp=sharing

Au total, c'est 25 benchmarks différents qui ont été réalisés. La campagne s'est décomposée en deux parties. Les 12 premiers benchmarks augmentaient petit à petit les quantités de CPU et RAM alloué au pod. Partant de 0,1 CPU et 256 MiB de RAM et allant jusqu'à 1,1 CPU et 2500 MiB en demande. Parmi ces 12 benchmarks, nous avons sélectionné celui qui avait démontré le plus de performance en termes de requêtes par secondes. Puis, nous avons effectué 13 nouveaux benchmarks en faisant varier les demandes et limites en CPU et RAM de manière beaucoup plus fine que lors de la première partie de benchmarks, dans le but d'obtenir encore de meilleurs résultats. Voici ainsi le résultat du benchmark le plus performant que nous avons réussi à réaliser :

Benchmark	Request		Limit		Iterations	Success *(in %)*	Pod CPU Usage *(in cores)*	Pod Memory Usage *(in MiB)*	Pod Memory Usage of limit *(in %)*	Requests per seconds
	CPU *(in cores)*	RAM *(in MiB)*	CPU *(in cores)*	RAM *(in MiB)*						
20	0,4	400	Any	400	119545	99,34%	4,17	171	42,80%	655

La quantité de ressources idéale pour un pod de notre application est donc 0,4 core et 400 MiB en demande, ainsi que 400 MiB en limite pour la RAM. Le nombre d'appels traités par seconde est alors aux alentours de 650. Ce nombre peut être multiplié en fonction du nombre de réplicas ajoutés par notre HPA.

V. TD5

A) Écris-moi un log

1) Acronyme ELK

Elasticsearch : outil de stockage et centralisation des données des logs. Il est notamment utilisé pour sa capacité à réaliser l'indexation et la recherche des données. Il fournit un moteur de recherche distribué à travers une interface REST.

Logstash : outil open source d'ingestion de données. Il permet de collecter des données provenant de diverses sources, de les transformer et de les envoyer à la destination souhaitée (ici Elasticsearch).

Kibana : outil de visualisation et d'exploration des données. Il permet d'analyser des données de journalisation en proposant des filtres afin d'effectuer de la surveillance.

2) Alternative à Logstash

Dans notre implémentation de la stack ELK, nous avons remplacé Logstash par Filebeat. Dans notre cas d'utilisation, l'objectif de ces deux briques est le même. Filebeat doit récupérer les données de journalisation de notre cluster Kubernetes pour les envoyées dans l'Elasticsearch.

3) Configuration spécifique de Elasticsearch

Elasticsearch, créé de nombreux segments de mémoire virtuelle pour gérer l'indexation et la recherche de données. Si le paramètre vm.max_map_count est trop bas, Elasticsearch peut rencontrer des problèmes de performance ou même échouer à démarrer.

4) Configuration spécifique de Kibana

Il est impératif que le paramètre "elasticsearchRef.name" dans le fichier de déploiement Kibana ait le même nom que l'instance Elasticsearch, car cela permet à Kibana de connaître l'instance Elasticsearch avec laquelle il doit communiquer pour récupérer les données.

5) Configuration spécifique de Filebeat

La variable d'environnement ELASTICSEARCH_HOST est défini comme "elastic-cluster-es-http.elastic.SVC". Cela indique que Filebeat doit envoyer les données de journalisation qu'il récupère au service d'Elasticsearch appelé elastic-cluster-es-http dans le namespace elastic.

6) Utilisation de Kibana

Pour accéder à Kibana depuis l'extérieur du cluster, nous avons mis en place un ingress permettant d'accéder au service **kibana-kb-http** depuis l'extérieur du cluster. Il est important de préciser l'annotation **nginx.ingress.kubernetes.io/backend-protocol** permettant d'indiquer que le ingress doit communiquer en HTTPS avec le service. Dans le cas où cette indentation viendrait à manquer, le ingress essayerait de communiquer en HTTP avec le service Elastisearch, ce qui ne fonctionnerait pas, car par défaut, il n'accepte que les connexions en HTTPS.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kibana
  namespace: elastic
  annotations:
    nginx.ingress.kubernetes.io/backend-protocol: HTTPS
spec:
  ingressClassName: nginx
```

```yaml
rules:
- host: kibana.orch-team-a.pns-projects.fr.eu.org
  http:
    paths:
    - path: /
      pathType: Prefix
      backend:
        service:
          name: kibana-kb-http
          port:
            number: 5601
```

Figure V.A.1 : Fichier de déploiement du ingress Kibana

Lorsqu'on accède à Kibana depuis le navigateur, il est possible de trier les informations selon les critères des noms de champs et de période.

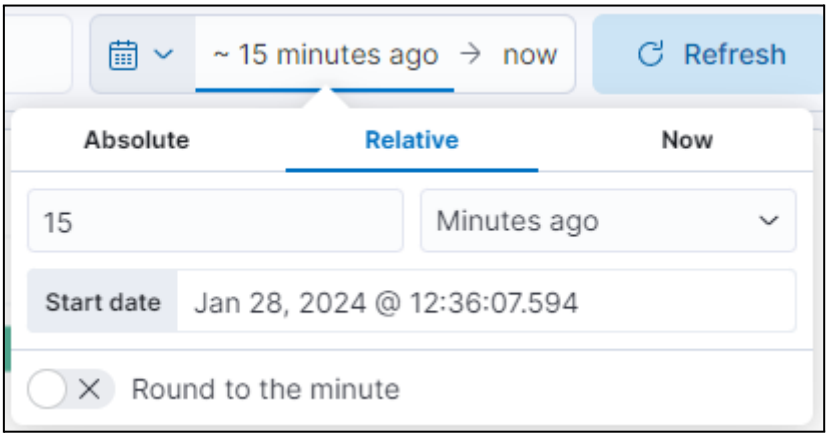

Figure V.A.2 : Paramétrage de la période à observer

Afin de récupérer la région dans laquelle est déployé notre cluster, il est nécessaire de trier sur les champs contenant des informations sur la région. Dans la *Figure V.A.3*, on peut observer que notre cluster est déployé dans la région GRA7 (la même région est retrouvée sur tous les logs de notre application). Ça signifie que notre cluster se situe à Gravelines en France. Avec Kibana, il est possible de faire des tris plus spécifiques, par exemple, comme trier sur les noms de pods spécifiques.

kubernetes.pod.na...	message	kubernetes.node.labels.topology_kubernetes_io/region
dashboard-metrics-scraper-5cb4f4bb9c-925wt	51.91.140.221 - - [28/Jan/2024:11:58:25 +0000] "GET /healthz HTTP/1.1" 200 13 "" "dashboard/v2.7.0"	GRA7
canal-n28pb	2024-01-28 11:58:25.767 [INFO][45] felix/summary.go 100: Summarising 14 dataplane reconciliation loops over 1m2.7s: avg=10ms longest=36ms ()	GRA7
canal-2jbz6	2024-01-28 11:58:25.575 [INFO][4010558] monitor-addresses/startup.go 432: Early log level set to info	GRA7

Figure V.A.3 : Affichage des logs et de leur région