Rapport de Conception : Projet AutoScaling et IaC

Professeurs : **Binh-Minh Bui-Xuan**
Alfred Deivassagayame
Arthur Escriou

Étudiants : **Amir AIT HABIB - 21414628**
Taqiyeddine DJOUANI - 21406863

SCIENCES SORBONNE UNIVERSITÉ

I. Introduction

Ce projet s'inscrit dans une démarche DevOps visant à construire une infrastructure cloud-native, modulaire et entièrement automatisée. L'architecture cible repose sur un cluster Kubernetes, dans lequel sont orchestrés les composants suivants :

- Une base de données Redis configurée en master/replica,
- Une application web stateless développée en Node.js,
- Un outil de monitoring basé sur Prometheus, Grafana et Redis Exporter.

L'objectif n'était pas seulement de faire fonctionner ces éléments ensemble, mais de les intégrer dans une architecture **scalable dynamiquement**, **observables à travers des métriques précises**, et **entièrement reproductibles via Infrastructure as Code**. L'ensemble est déployé par script, sans intervention manuelle, conformément aux bonnes pratiques du déploiement continu.

II. Contexte et Objectifs

Le projet vise à déployer une infrastructure de production sur Kubernetes avec les points suivants :

- **Automatisation complète du déploiement** : Utilisation de scripts bash pour déployer tous les composants sans intervention manuelle.
- **Scalabilité et montée en charge dynamique** : Mise en place d'une architecture capable de monter en charge automatiquement, en séparant les opérations de lecture (réalisées par les replicas) et d'écriture (gérées par le master).
- **Reproductibilité** : Assurer que l'ensemble du déploiement soit facilement reproductible sur tout cluster compatible, garantissant ainsi la portabilité et la maintenabilité de l'architecture.

III. Démarche et Choix Techniques

1. Déploiement Automatisé

Scripts Bash : Un script d'automatisation (deploy-all.sh) a été conçu pour déployer l'ensemble des composants (Redis, Node.js, Monitoring) d'un simple clic. Ce script permet d'orchestrer le déploiement et de simplifier la réplication de l'infrastructure. Celui-ci applique trois répertoires de manière séquentielle :

```
#!/bin/bash

kubectl apply -f Ops/NodeJsApp/
kubectl apply -f Ops/Redis/
kubectl apply -f Monitoring/
```

- Deployment, Service, ServiceMonitor,
- Respect des labels et sélecteurs pour la découverte automatique,
- Découplage logique (Dev / Monitoring / Ops) pour plus de lisibilité et de maintenabilité.

Cette organisation permet un **déploiement déclaratif, traçable et modulaire**, qui facilite

aussi bien le debug que l'évolution du projet.

2. Architecture Redis

Cluster Master/Replica : La décision de mettre en place un cluster Redis avec un master pour les écritures et des replicas pour les lectures a été motivée par le besoin de dissocier les charges et d'optimiser la performance. Ce modèle assure la consistance des données tout en améliorant la réactivité des lectures. Les fichiers redis-master.yaml et redis-slaves.yaml définissent ces rôles. Les réplicas sont configurés avec :

```
- name: redis
  image: redis:6.2-alpine
  command: ["redis-server", "--slaveof", "redis-master", "6379"]
```

Ce choix garantit une **cohérence forte** des données tout en permettant une **scalabilité horizontale contrôlée** (seuls les replicas sont réplicables à l'infini sans compromettre l'intégrité).

- Résultat de la commande kubectl get pods

```
ubuntu in 7assoub in ~ via v3.8.10
> k get pod
NAME                                                READY   STATUS    RESTARTS        AGE
redis-master-869647f547-j5kfk                       1/1     Running   0               17h
redis-slave-56b5dd8994-npvw5                        1/1     Running   0               16h
redis-slave-56b5dd8994-t6czm                        1/1     Running   0               16h
redis-exporter-5c96c859df-fxh9j                     1/1     Running   0               15h
prometheus-kube-prometheus-operator-745d7d9ffb-hj8sr   1/1     Running   0               19m
prometheus-prometheus-node-exporter-9q4jv           1/1     Running   0               19m
prometheus-kube-state-metrics-86cd66b745-r596n      1/1     Running   0               19m
alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running   0               18m
prometheus-prometheus-kube-prometheus-prometheus-0   2/2     Running   0               18m
prometheus-grafana-777fc58fbd-2zxwj                 3/3     Running   0               19m
nodejs-app-56ddd7db6-ghg6h                          1/1     Running   4 (5m20s ago)   16h

ubuntu in 7assoub in ~ via v3.8.10
> kubectl exec -it redis-slave-56b5dd8994-npvw5 -n iacproject -- redis-cli
127.0.0.1:6379> KEYS *
1) "key"
2) "user123"
127.0.0.1:6379>
```

Figure : Résultat de la commande `kubectl get pods` montrant l'état de tous les composants du projet. On y observe la présence des pods Redis (master et replicas), Prometheus, Grafana, Redis Exporter et l'application Node.js, tous en cours d'exécution.

- Résultat de commandes redis-cli sur master et replicas :

```
ubuntu in 7assoub in ~ via v3.8.10
> kubectl exec -it redis-slave-56b5dd8994-npvw5 -n iacproject -- redis-cli
127.0.0.1:6379> KEYS *
1) "key"
2) "user123"
127.0.0.1:6379> exit

ubuntu in 7assoub in ~ via v3.8.10 took 32s
> kubectl exec -it redis-slave-56b5dd8994-t6czm -n iacproject -- redis-cli
127.0.0.1:6379> KEYS *
1) "key"
2) "user123"
127.0.0.1:6379> exit

ubuntu in 7assoub in ~ via v3.8.10 took 4s
> kubectl exec -it redis-master-869647f547-j5kfk -n iacproject -- redis-cli
127.0.0.1:6379> KEYS *
1) "user123"
2) "key"
127.0.0.1:6379>
```

Figure : Résultat des commandes redis-cli montrant la présence synchronisée des clés key et

user123 sur les deux replicas Redis ainsi que sur le master. Cela confirme la bonne propagation des écritures du master vers les replicas, conformément au fonctionnement attendu du cluster Redis.

- **Service Discovery** : Des services Kubernetes exposent le master et les replicas, facilitant ainsi la communication entre l'application Node.js et le cluster Redis.

3. Application Node.js

L'application, développée en Node.js, est **stateless** par conception. Cela signifie :
Aucun état n'est stocké localement dans le conteneur, et elle peut être dupliquée à volonté pour absorber des pics de charge, sans risque de collision de session ou de perte de données.
L'application web est dotée de deux clients Redis distincts, chacun se connectant respectivement au master (pour les opérations d'écriture) et aux replicas (pour les opérations de lecture)

```
const client = createClient({
  url: REDIS_URL,
})

const readClient = createClient({
  url: REDIS_REPLICAS_URL,
})
```

Cette séparation permet un contrôle très fin de la répartition de la charge et garantit une **résilience du backend** en cas de surcharge de lecture.

Instrumentation Prometheus : L'intégration de l'outil [express-prometheus-middleware] permet d'exposer un endpoint `/metrics`, essentiel pour la collecte de métriques de performance et la surveillance de l'application.

- Voici une image montrant l'accès HTTP à l'endpoint `/items`

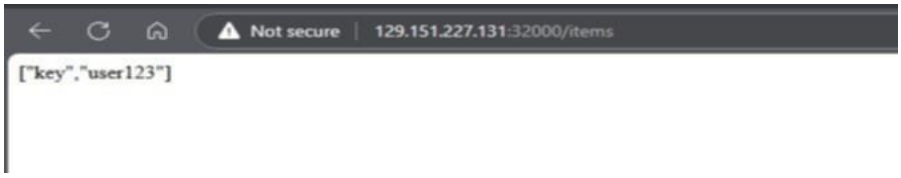

Figure : Résultat de l'appel HTTP à l'endpoint `/items`, montrant la récupération réussie des clés stockées dans Redis. Cette réponse prouve que le serveur Node.js communique correctement avec les replicas Redis et que l'API est accessible publiquement via NodePort.

- Voici une image montrant un test d'écriture + lecture via `curl` :

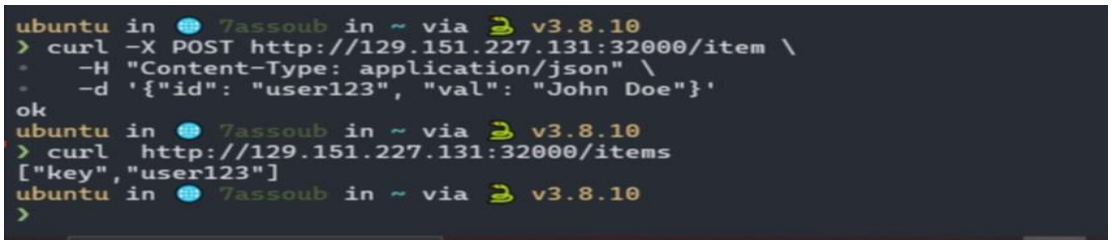

Figure : Écriture d'une donnée dans Redis via une requête HTTP POST sur `/item`, suivie d'une lecture via `/items`. Ces tests réalisés avec `curl` démontrent le bon fonctionnement de l'API Node.js et la persistance des données dans Redis.

- Voici une image montrant les Logs de démarrage du serveur Node.js

```
Sun, 16 Mar 2025 22:15:32 GMT:  redis connected to redis://redis-master:6379
Sun, 16 Mar 2025 22:15:32 GMT:  redis replicas connected to redis://redis-slave:6379
Sun, 16 Mar 2025 22:15:32 GMT:  Set "key" value to "redis connected to redis://redis-master:6379"
Sun, 16 Mar 2025 22:15:32 GMT:  listening at http://localhost:3000 server b8fe4964-ff9b-4ae1-abc4-f971d58cbc46
Sun, 16 Mar 2025 22:16:06 GMT:  It is working, good job  b8fe4964-ff9b-4ae1-abc4-f971d58cbc46 1742163366163
Sun, 16 Mar 2025 22:16:07 GMT:  It is working, good job  b8fe4964-ff9b-4ae1-abc4-f971d58cbc46 1742163367401
Sun, 16 Mar 2025 22:16:14 GMT:  It is working, good job  b8fe4964-ff9b-4ae1-abc4-f971d58cbc46 1742163374572
Sun, 16 Mar 2025 22:16:14 GMT:  It is working, good job  b8fe4964-ff9b-4ae1-abc4-f971d58cbc46 1742163374766
Sun, 16 Mar 2025 22:16:15 GMT:  It is working, good job  b8fe4964-ff9b-4ae1-abc4-f971d58cbc46 1742163375399
Sun, 16 Mar 2025 22:16:15 GMT:  It is working, good job  b8fe4964-ff9b-4ae1-abc4-f971d58cbc46 1742163375579
```

Figure : Logs de démarrage de l'application Node.js, confirmant la connexion réussie à Redis Master et Replica, l'insertion d'une valeur de test, et l'écoute effective du serveur sur le port 3000. Ces messages valident le bon fonctionnement initial de l'API.

4. Stack de Monitoring

Afin d'assurer la **visibilité** sur les performances et le comportement de chaque composant, un système de monitoring complet a été intégré :

- **Prometheus & Grafana via Helm** : Le déploiement de Prometheus et Grafana a été réalisé via le chart Helm `kube-prometheus-stack`, simplifiant l'installation et la configuration de ces outils. Ce choix garantit une intégration étroite avec Kubernetes et une découverte automatique des métriques grâce aux ServiceMonitors.

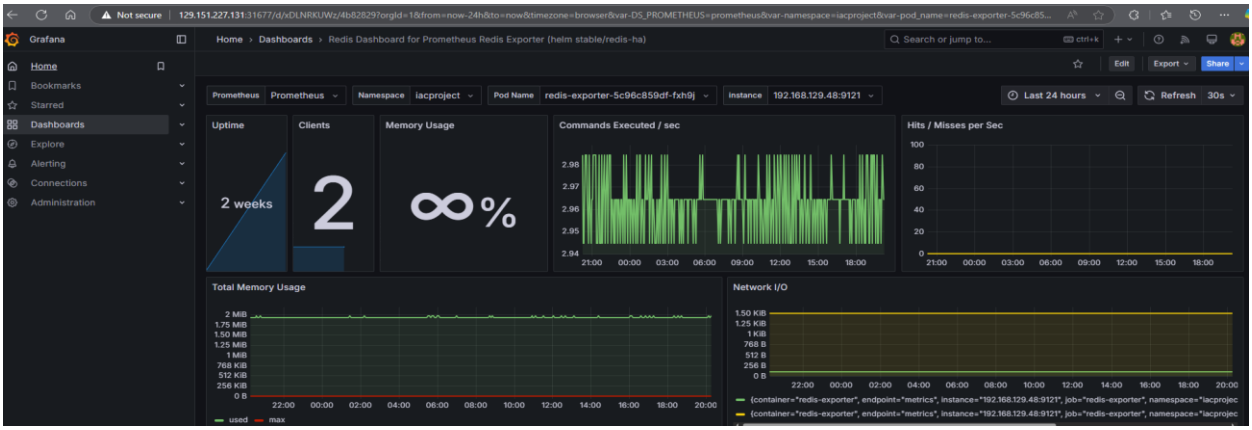

Figure : Tableau de bord Grafana illustrant les métriques collectées depuis Redis Exporter, dont le nombre de commandes exécutées, l'uptime, l'usage mémoire, le trafic réseau et le nombre de clients Redis actifs. Ces données permettent une supervision fine du comportement du service Redis.

- **Redis Exporter** : Pour combler l'absence native d'un endpoint Prometheus dans Redis, l'utilisation d'un Redis Exporter (basé sur l'image oliver006/redis_exporter) permet de collecter et d'exposer les métriques Redis sous un format compatible avec Prometheus.

```
containers:
- name: redis-exporter
  image: oliver006/redis_exporter:latest
```

IV. Montée en Charge Automatique et Dynamique

L'architecture a été pensée pour répondre à une montée en charge dynamique :

- **Séparation des Charges** : En dirigeant les écritures vers le master et les lectures vers les replicas, l'architecture répartit intelligemment la charge, améliorant ainsi la réactivité du système.
- **Monitoring et Alertes** : L'intégration de Grafana et la configuration d'alertes via Prometheus permettent de suivre en temps réel les performances et la santé du système. Cela facilite l'ajustement automatique des ressources en cas de pic de charge.
- **Scalabilité Horizontale** : Le déploiement sur Kubernetes, couplé aux mécanismes de scaling automatique (Horizontal Pod Autoscaler par exemple), garantit que l'infrastructure puisse s'ajuster en fonction de la demande.

V. Reproductibilité du Déploiement

La reproductibilité a été un aspect central du projet :

- **Infrastructure as Code** : L'ensemble des configurations (YAML, scripts bash, Dockerfile) permet de recréer l'architecture sur n'importe quel cluster Kubernetes.
- **Documentation Complète** : Le README.md et ce rapport détaillé fournissent une description exhaustive de la démarche, facilitant la prise en main et la reproduction dudéploiement par d'autres équipes.
- Les images Docker sont disponibles publiquement (ex : taqiyeddinedj/redis-node:1.0.0).
- **Utilisation de Helm** : Le recours à Helm pour le déploiement des composants de monitoring garantit une installation cohérente et simplifiée, tout en facilitant les mises à jour et les rollbacks.
- Le script deploy-all.sh permet un déploiement sans intervention humaine.

VI. Décisions et Justifications

❖ Automatisation et Simplification

- **Pourquoi l'automatisation ?**
  L'objectif était de réduire les erreurs humaines et de garantir une installation rapide et fiable de l'ensemble des composants. Le script `deploy-all.sh` centralise toutes les étapes du déploiement, ce qui permet une exécution sans intervention manuelle.

❖ Séparation des Rôles dans Redis

- **Avantages du modèle Master/Replica**
  Ce modèle permet une gestion optimale des charges en dissociant les opérations de

lecture et d'écriture. Cela améliore la scalabilité et assure une meilleure résilience du système en cas de défaillance d'un replica.

❖ **Choix des Outils de Monitoring**

- **Helm et kube-prometheus-stack**
  L'utilisation de Helm simplifie grandement le déploiement et la configuration des outils de monitoring. Le choix du chart `kube-prometheus-stack` permet une intégration transparente avec Kubernetes et assure une collecte exhaustive des métriques.

VII. Conclusion :

Le projet a permis de mettre en place une infrastructure automatisée, scalable et hautement reproductible sur Kubernetes. Chaque décision – qu'il s'agisse du choix de l'architecture Redis, de l'instrumentation de l'application Node.js ou de l'intégration d'un stack de monitoring robuste – a été motivée par le besoin de garantir performance, fiabilité et simplicité d'utilisation. Cette démarche d'automatisation et de documentation complète assure non seulement une mise en production efficace mais facilite également la maintenance et les évolutions futures de l'infrastructure.