h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

Mise en place d'une architecture clusterisée à haute disponibilité pour Botalista

Thèse de Bachelor présentée par

Dario GENGA

pour l'obtention du titre Bachelor of Science HES-SO en

Informatique et systèmes de communication avec orientation Informatique logicielle

août, 2024

Professeur HES responsable

Florent GLÜCK

Mandant

Cyril BOILLAT

Légende et source de l'illustration de couverture : Un arbre, des buissons et des papillons dans un heptagone. Réalisé par Dario GENGA.

TABLE DES MATIÈRES

Remerciements . . . . . **vi**

Énoncé . . . . . **vii**

Résumé . . . . . **viii**

Liste des acronymes . . . . . **ix**

Glossaire . . . . . **x**

Liste des illustrations . . . . . **xi**

Liste des tableaux . . . . . **xii**

Liste des listings . . . . . **xiii**

Liste des annexes . . . . . **xiv**

Introduction **1**

1 Chapitre 1 : Analyse de l'existant **4**
1.1 Analyse générale 4
1.2 Technologies utilisées 5
1.3 Infrastructure réseau et matérielle 6
a Cluster Kubernetes 7
b Serveur *jiangxia* 7
1.4 Les instances et modules de Botalista 8
a Modules métiers et logiciels 9
b Implémentation des modules au sein de Kubernetes 9
c Présentation des instances 11
d Instance clone 12
1.5 Problèmes identifiés 12
a Plan de contrôle à un seul nœud 12
b Accès au service dépendant du nœud *antilia* 12
c Bases de données sans réplications 13
d Volumes partagés dépendant d'un serveur NFS 13
e Non-réplication des modules et applications 14
f Consommations en ressources RAM 14

2 Chapitre 2 : Architecture à haute disponibilité **15**
2.1 Infrastructure proposée 15
2.2 Cluster Kubernetes 17
a Plan de contrôle à haute disponibilité 17
b Workers 19
c Bases de données conteneurisées 20

d Stockage distribué . . . 21
e Longhorn . . . 22
2.3 Bases de données avec réplication . . . 23
a PostgreSQL . . . 24
b MongoDB . . . 25
c Redis . . . 26
2.4 Accès au cluster . . . 27
a HAProxy . . . 28
b Keepalived . . . 30
c Détails de l'implémentation . . . 31
2.5 RabbitMQ . . . 33
a Installation de l'opérateur . . . 33
b Création du cluster RabbitMQ . . . 33
2.6 Registre Docker . . . 34
2.7 Réplication des modules . . . 34

3 Chapitre 3 - Analyse de la robustesse . . . 36
3.1 Robustesse des composants externes . . . 36
a Attribution de l'adresse IP virtuelle par Keepalived . . . 36
b Analyse de la charge . . . 37
3.2 Robustesse du cluster Kubernetes . . . 38
a Panne du plan de contrôle . . . 39
b Panne d'un worker . . . 39
3.3 Tolérances de l'architecture . . . 39
a Tolérance des serveurs de répartition de la charge . . . 40
b Tolérance du plan de contrôle . . . 40
c Tolérance des workers . . . 40
d Tolérance de l'architecture . . . 41
e Tableau récapitulatif . . . 41

4 Chapitre 4 - Étude comparative . . . 42
4.1 Situation actuelle . . . 42
4.2 Ressources requises . . . 43
4.3 OVHcloud . . . 43
a Impact environnemental . . . 44
b Public cloud . . . 44
c Managed Kubernetes Service . . . 45
4.4 Infomaniak . . . 46
a Impact environnemental . . . 46
b Publique cloud . . . 47
4.5 Exoscale . . . 47
a Impact environnemental . . . 48
b Publique cloud . . . 48
4.6 Résumé . . . 49
a Comparaison des impacts environnementaux . . . 49
b Comparaison des frais d'architecture . . . 50
c Prestataire de service sélectionné . . . 50

Conclusion **51**

Annexes **53**

Références documentaires **82**

REMERCIEMENTS

Je souhaite tout d'abord remercier mon professeur responsable M. Florent Glück qui m'a guidé et soutenu durant ce travail de Bachelor.

Je remercie ensuite M. François Tamone pour m'avoir aidé au début du projet concernant l'aspect réseau de Kubernetes, ainsi que M. Andrien Lescourt pour la gestion des machines virtuelles au sein du réseau école.

Je tiens également à remercier M. Cyril Boillat qui a répondu à mes questions concernant Botalista.

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Printemps 2024
Session de bachelor

MISE EN PLACE D'UNE ARCHITECTURE CLUSTERISÉE À HAUTE DISPONIBILITÉ POUR BOTALISTA

ORIENTATION : INFORMATIQUE LOGICIELLE

Descriptif : L'association Botalista a pour objectif la mise en place d'une communauté de jardins et d'instituts botaniques qui partage ses connaissances et expériences dans le domaine de la botanique. Pour cela, l'association a développé une solution informatique pour la gestion de l'ensemble des processus métiers liés aux différentes activités d'un jardin botanique. Cette solution, nommée Botalista.software, est développée sous la forme d'une application web déployée sur un cluster Kubernetes en tant que Software as a Service. Elle est conçue pour être un outil communautaire, accessible et modulaire pour les différents membres de l'association. Mis en place depuis désormais trois ans, le cluster Kubernetes ne dispose cependant pas de redondance pour ses composants critiques. La tolérance aux pannes est faible, ce qui rend le système peu robuste. L'objectif du présent travail est de proposer une nouvelle architecture en cluster qui permet d'assurer la disponibilité du service même en cas de pannes matérielles ou logicielles.

Travail demandé :

- Concevoir et mettre en place une infrastructure sans point de défaillance unique.
- Analyser, puis sélectionner le prestataire de service le plus intéressant en fonction de l'architecture choisie tout en respectant les contraintes géographiques et politiques.
- Mettre en place des mécanismes de mise à l'échelle automatique (scale up et down si le temps le permet) pour l'architecture.
- Tester et valider la robustesses de l'architecture proposée.
- Étudier les aspects permettant de développer la solution la plus verte possible.
- Effectuer une étude comparative (robustesse, coût, emprunte écologique) entre l'existant et la nouvelle architecture.
- Mesurer l'impact (en ressources) lors de l'ajout d'un ou plusieurs instituts botaniques au sein de l'association.

Candidat :

DARIO GENGA

Filière d'études : ISC

Professeur responsable :

FLORENT GLÜCK

En collaboration avec : Association Botalista

Travail de bachelor soumis à une convention de stage en entreprise : non

Travail soumis à un contrat de confidentialité : non

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Printemps 2024
Session de bachelor

RÉSUMÉ

L'association Botalista a pour objectif la mise en place d'une communauté de jardins et d'instituts botaniques qui partage ses connaissances et expériences dans le domaine de la botanique. Pour cela, l'association a développé une solution informatique pour la gestion de l'ensemble des processus métiers liés aux différentes activités d'un jardin botanique. Cette solution, nommée *Botalista.software*, est développée sous la forme d'une application web déployée sur un cluster Kubernetes en tant que Software as a Service. Elle est conçue pour être un outil communautaire, accessible et modulaire pour les différents membres de l'association. Mis en place depuis désormais 3 ans, le cluster Kubernetes ne dispose cependant pas de redondance pour ses composants critiques. La tolérance aux pannes est faible, ce qui rend le système peu robuste. L'objectif du présent travail est de proposer une nouvelle architecture en cluster qui permet d'assurer la disponibilité du service même en cas de pannes matérielles ou logicielles. De manière à proposer cette nouvelle architecture, j'ai tout d'abord étudié la solution existante afin de déterminer les éléments vulnérables à améliorer. Suite à cette étude, j'ai conceptualisé une architecture sans point de défaillance unique afin d'assurer la haute disponibilité de l'ensemble des composants essentiels à Botalista, qu'ils soient matériels ou logiciels. Cette architecture est également conçue pour être facilement extensible, de manière à lui permettre d'évoluer selon les besoins de l'association. Afin de valider sa robustesse, j'ai déployé l'architecture sur un environnement appartenant à l'école pour étudier son comportement en cas de défaillance affectant ses divers composants. Finalement, ce travail se conclut sur une étude comparative entre l'architecture existante et celle proposée, afin de déterminer le prestataire de service le plus adapté selon les aspects financiers et écologiques.

Candidat :

DARIO GENGA

Filière d'études : ISC

Professeur responsable :

FLORENT GLÜCK

En collaboration avec : Association Botalista

Travail de bachelor soumis à une convention de stage en entreprise : non

Travail soumis à un contrat de confidentialité : non

LISTE DES ACRONYMES

ACL Access Control List.

AMQP Advanced Message Queuing Protocol.

API Application Programming Interface.

CNI Container Network Interface.

CRI Container Runtime Interface.

FaaS Function as a Service.

GED Gestion Électronique des Documents.

IoT Internet of Things.

IPAM IP Address Management.

k8s Kubernetes.

LVS Linux Virtual Server.

MKS Managed Kubernetes Service.

NFS Network File System.

RAM Random Access Memory.

SaaS Software as a Service.

UID Unique Identifier.

VIP Virtual IP.

VM Machine virtuelle.

VRRP Virtual Router Redundancy Protocol.

GLOSSAIRE

chart Une collection de fichiers qui décrivent un ensemble de ressources Kubernetes.

cluster Un ensemble de machine physique ou virtuelle.

conteneur Une forme de virtualisation qui permet d'isoler un ensemble de processus du système hôte et des autres conteneurs.

daemon Un processus Linux qui est exécuté en arrière-plan.

etcd Une base de données en clé-valeur utilisée par Kubernetes.

failover Mécanisme de basculement effectué pour rétablir un état souhaité.

haute disponibilité Système fonctionnel en permanence, même en cas d'un arrêt planifié, ou non, d'un composant.

Helm Le gestionnaire de paquets pour Kubernetes.

image Une image Docker représente le système de fichier racine d'un conteneur.

load balancing Répartition de la charge.

module logiciel Un microservice propre à une instance *Botalista.software*.

module métier Un microservice partagé à toutes les instances de *Botalista.software*.

nœud Une machine d'un cluster. Appelé *node* en anglais.

pod Un ensemble d'uns ou plusieurs conteneurs co-localisés et co-planifiés.

service Un objet Kubernetes qui expose le réseau d'une application s'exécutant sur un pod.

volume Un répertoire contenant des données accessibles par les conteneurs d'un pod.

worker Un nœud sur lequel une charge de travail Kubernetes est déployée.

LISTE DES ILLUSTRATIONS

1 Les modules de *Botalista.software* 2

1.1 Les technologies de Botalista 5
1.2 L'infrastructure réseau et matérielle de Botalista 6

2.1 Infrastructure proposée 15
2.2 Cluster à haute disponibilité - topologie empilée 18
2.3 Cluster à haute disponibilité - topologie externe 18
2.4 Architecture de Longhorn 23
2.5 Schéma du cluster PostgreSQL et des services Kubernetes associés 24
2.6 Un Replica Set MongoDB composé d'une instance primaire et deux secondaires 25
2.7 Élection d'une nouvelle instance primaire pour un Replica Set MongoDB . . . 26
2.8 Représentation d'un cluster Redis avec Sentinel 27
2.9 Répartition de charge par *round robin* 29
2.10 Schéma des composants Kubernetes pour accéder à *Botalista.software* 32

Références des URL

— URL01 https://botalista.community/concept/modules
— URL02 https://kubernetes.io/docs/concepts/overview/components/
— URL03 https://medium.com/@seifeddinerajhi/navigating-the-journey-of-a-pod-a-guide-to-the-exciting-world-of-pod-lifecycle-a1fbc2c98c55
— URL04 https://yuminlee2.medium.com/kubernetes-container-network-interface-cni-ee5b21514664
— URL05 https://tamerlan.dev/load-balancing-in-kubernetes-a-step-by-step-guide/
— URL06 https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/
— URL08 https://traefik.io/glossary/round-robin-load-balancing/
— URL09 https://longhorn.io/
— URL10 https://www.mongodb.com/docs/manual/replication/
— URL11 https://redis.io/learn/operate/redis-at-scale/high-availability/understanding-sentinels

LISTE DES TABLEAUX

1.1 Résumé de l'infrastructure matérielle 6
1.2 Liste des modules de *Botalista.Software* 8
1.3 Les modules métiers déployés 9
1.4 Les modules logiciels qui composent une instance de *Botalista.software* 10
1.5 Consommation en RAM des instances de *Botalista.Software* 11

2.1 Tableau de l'infrastructure à haute disponibilité proposé 16
2.2 Majorité et tolérance de pannes selon taille du cluster etcd 17

3.1 Attribution de l'adresse IP virtuelle selon le statut du service HAProxy 37
3.2 Tableau récapitulatif de la tolérance de l'architecture 41

4.1 Tableau des frais annuels pour l'architecture actuelle de Botalista 42
4.2 Tableaux comparatifs des instances OVHcloud 45
4.3 Tarification de l'architecture public cloud chez OVHcloud 45
4.4 Tarification de l'architecture avec service Kubernetes chez OVHcloud 46
4.5 Tableaux comparatifs des instances Infomaniak 47
4.6 Tarification de l'architecture public cloud chez Infomaniak 47
4.7 Tableaux comparatifs des instances Exoscale 48
4.8 Tarification de l'architecture public cloud chez Exoscale 49
4.9 Tarifications annuelles et mensuelles entre OVHcloud, Infomaniak et Exoscale . 50

Références des URL

— URL07 https://etcd.io/docs/v3.6/faq/#what-is-failure-tolerance

LIST OF LISTINGS

2.1 Ajout d'une *taint* pour les nœuds de développement et présentation 19
2.2 Ajout d'une *toleration* au format YAML 20
2.3 Ajout d'un label pour les nœuds qui gèrent les bases de données 20
2.4 L'affinité et anti-affinité pour les instances des bases de données PostgreSQL . 20
2.5 Commande pour récupérer le Redis maître 26
2.6 HAProxy configuration pour plan de contrôle à haute disponibilité 28
2.7 HAProxy configuration HTTP simple 29
2.8 Keepalived configuration pour la machine principale 30
2.9 Exemple d'un objet `IngressRoute` au format YAML 31
2.10 Installation du RabbitMQ Cluster Kubernetes Operator 33
2.11 Un objet RabbitmqCluster avec 3 replicas au format YAML 33
3.1 Arrêt du service HAProxy sur LB_MASTER 36
3.2 Attribution de l'adresse IP virtuelle pour LB_BACKUP 36
3.3 Simulation de 200 utilisateurs concurrent avec l'utilitaire Siege 38

LISTE DES ANNEXES

1 Annexe 1 - Kubernetes 54
1.1 Composants d'un cluster 54
1.2 Charges de travail 56
1.3 Le modèle réseau de Kubernetes 58
1.4 Installation 62

2 Annexe 2 - Liste des composants 65
2.1 Tableau des versions 65
2.2 Kubernetes 66
2.3 Calico 67
2.4 Provisionneur de volume 67
2.5 Traefik 67
2.6 Bases de données 68
2.7 NFS 69
2.8 Surveillance de l'état de santé du cluster 70
2.9 RabbitMQ 70
2.10 Keycloak 70
2.11 Alfresco 70

3 Annexe 3 - Installation et configuration de Longhorn 71
3.1 Installation 71
3.2 Configuration 71

4 Annexe 4 - Installation et configuration des bases de données 73
4.1 PostgreSQL 73
4.2 MongoDB 75
4.3 Redis 78

INTRODUCTION

Les jardins et instituts botaniques sont des acteurs indispensables pour le maintien de la biodiversité au sein de notre société. Ces acteurs assument des rôles clés tels que la conservation des plantes et leurs ressources génétiques, la systématique [1], la taxonomie [2] et l'éducation.

Pour remplir ce rôle, une grande majorité de ces acteurs utilisent des outils informatiques qui permettent une gestion de leurs collections. Néanmoins, ces outils ne répondent pas toujours à l'ensemble des besoins et surtout ne favorisent pas l'échange de savoir entre les différents instituts botaniques au plan national ou international.

Souhaitant répondre à ces besoins, la Ville de Genève développe depuis 2012 le progiciel de gestion *Botalista.software*, une solution informatique qui permet de gérer l'ensemble des processus métiers liés aux différentes activités d'un jardin botanique. Ce progiciel a été conçu pour être un outil communautaire, collaboratif, accessible et modulaire pour les divers instituts botaniques. Pour cela, l'association Botalista a été fondée le 30 septembre 2020 par la Ville de Genève, en collaboration avec les jardins botaniques des villes de Bordeaux, Paris et l'Université de Berne. Botalista est une association à but non-lucratif dont l'objectif est de mettre en place une communauté d'instituts botaniques qui peuvent partager leurs données et expériences au travers de *Botalista.software*. À ce jour, l'association est composée de seize membres situés en Suisse, France et Belgique.

Afin de répondre aux différents besoins des instituts botaniques, *Botalista.software* est une solution web composée de divers modules permettant la gestion des herbiers, flores et collections vivantes. Cette modularité permet à chaque membre de l'association le choix des différents composants et fonctionnalités dont il a besoin.

Ces modules, représentés par l'illustration 1, sont développés sous la forme d'applications indépendantes (microservices) et conteneurisées à partir d'images Docker. Un ensemble de ces applications représentent une instance de *Botalista.software*. Ce progiciel est déployé en tant que Software as a Service (SaaS) pour les membres de l'association Botalista, qui disposent de leur propre instance isolée des autres. Ces instances sont déployées sur un cluster Kubernetes qui est hébergé sur un fournisseur cloud privé mais géré par l'association Botalista.

Ce cluster Kubernetes est en place depuis 2021, mais il ne contient aucun mécanisme permettant d'assurer une haute disponibilité de la solution. Ceci signifie que la moindre panne d'un composant peut impacter la stabilité et le fonctionnement de l'architecture. L'association

1. « Science de l'inventaire et de l'organisation de la diversité des êtres vivants. » (sciencesnaturelles.ch)
2. Science de classification des organismes vivants dans des catégories hiérarchisées appelées taxons.

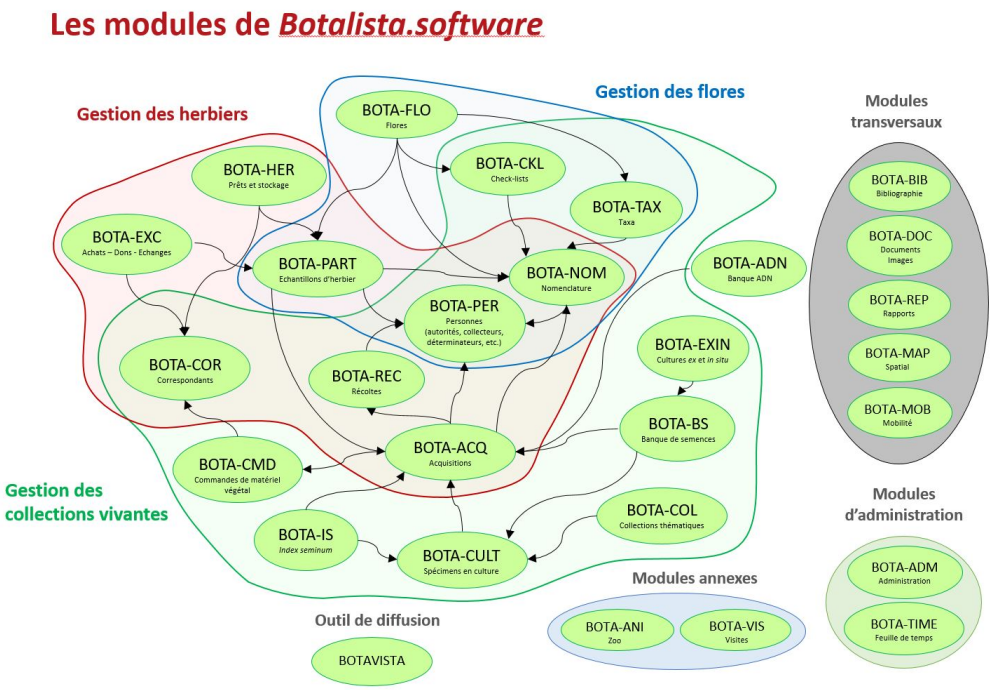

ILLUSTRATION 1 – Les modules de *Botalista.software*. Source : tiré de *botalista.community*, ref. URL01

grandissant d'année en année, chaque nouveau membre est un risque supplémentaire pour la stabilité du service de l'ensemble des utilisateurs. Il devient ainsi primordial d'apporter des modifications pour améliorer la robustesse de l'architecture.

Ce travail de Bachelor a pour objectif de proposer une nouvelle architecture en cluster qui permet d'assurer la disponibilité du service même en cas de pannes matérielles ou logicielles. Il s'agit du principe de *haute disponibilité*, signifiant que le système doit être fonctionnel en permanence même suite à un arrêt (planifié ou non) d'un composant. Ceci est possible en installant des composants de secours, qui prennent le relais afin d'assurer un fonctionnement continu du système.

Pour mener à bien mon travail, j'ai réalisé des réunions hebdomadaire avec mon professeur responsable pour discuter de l'avancement du projet et définir les objectifs des semaines à venir. Le travail réalisé est disponible sur le dépôt Git du projet [3]. Il contient les fichiers de configuration pour la mise en place de l'infrastructure, ma thèse au format PDF, les sources de ma thèse au format LaTeX et finalement mon journal du bord du projet. Ce dernier contient les réflexions et tâches réalisées tout au long du projet.

Cette thèse est découpée en plusieurs chapitres.

3. https://gitedu.hesge.ch/flg_bachelors/tb/botalista_high_availability

Le premier chapitre est consacré à l'analyse de l'architecture actuelle de Botalista, à la fois du point de vue matériel et applicatif. Cette analyse présente également les problèmes actuels de l'architecture.

Le deuxième chapitre présente la nouvelle architecture à haute disponibilité en abordant les modifications apportées, les raisons de ces changements et le fonctionnement des différents composants qui assurent la disponibilité du service.

Le troisième chapitre étudie la tolérance de l'architecture proposée en simulant le dysfonctionnement de ses composants. Ces simulation permettent de valider la robustesse de l'architecture dans son ensemble.

Le quatrième chapitre est une étude comparative de la robustesse, maintenance, prix et empreinte écologique entre le fournisseur actuel et d'autres envisageables pour la nouvelle architecture.

CHAPITRE 1 : ANALYSE DE L'EXISTANT

Ce chapitre est une analyse qui présente la structure actuelle de la solution *Botalista.software* lors de la réalisation du travail. Avant de poursuivre, il est recommandé de consulter l'annexe 1 qui explique les terminologies et composants principaux d'un cluster Kubernetes.

Cette analyse est découpée en cinq sections principales :

1. Une analyse générale de *Botalista.software*, qui présente principalement le fonctionnement de la version *Enterprise Edition* utilisée par les membres de l'association ;
2. Les technologies utilisées pour le développement de *Botalista.software* ;
3. L'infrastructure réseau et matérielle, qui correspond notamment au cluster Kubernetes ;
4. Une présentation des instances et modules de Botalista ;
5. Les problèmes identifiés, tels que les nœuds à risques au sein du cluster.

1.1. ANALYSE GÉNÉRALE

Botalista.software est une solution web composée de modules, ou microservices, étant en charge de la gestion d'un processus métier particulier. Ils assument des responsabilités diverses telles que la gestion des acquisitions, le suivi des cultures, les programmes de conservations et bien d'autres.

Cette solution est un projet open source, développée sous la licence Apache version 2.0. Elle est proposée en deux versions :

— *Community Edition* qui est libre de droit ;
— *Enterprise Edition* qui est réservée aux membres de l'association Botalista et qui est déployée en tant que Software as a Service.

Chaque membre de l'association possède sa propre instance de *Botalista.software* qui est déployée sur le même cluster Kubernetes dans un espace de nom [4] dédié. L'aspect modulaire de la solution permet de n'installer que les modules qui correspondent aux besoins des utilisateurs.

Botalista.software propose actuellement 28 modules, nombre qui ne cesse d'augmenter chaque année.

Le cluster Kubernetes est géré par Botalista et est actuellement déployé avec la version obsolète [5] 1.21.14. Ce dernier héberge actuellement 18 instances de *Botalista.software* : 16 de production, 1 de pré-production et 1 de présentation ouverte à tous.

4. Un espace de nom, ou *namespace*, permet d'isoler un groupe de ressources au sein du cluster.
5. Durant la réalisation de ce travail, la dernière version de Kubernetes est la 1.31.0, sortie le 13 août 2024.

1.2. TECHNOLOGIES UTILISÉES

Cette section présente les technologies principales de *Botalista.software*. La version *community* est représentée par l'illustration 1.1.

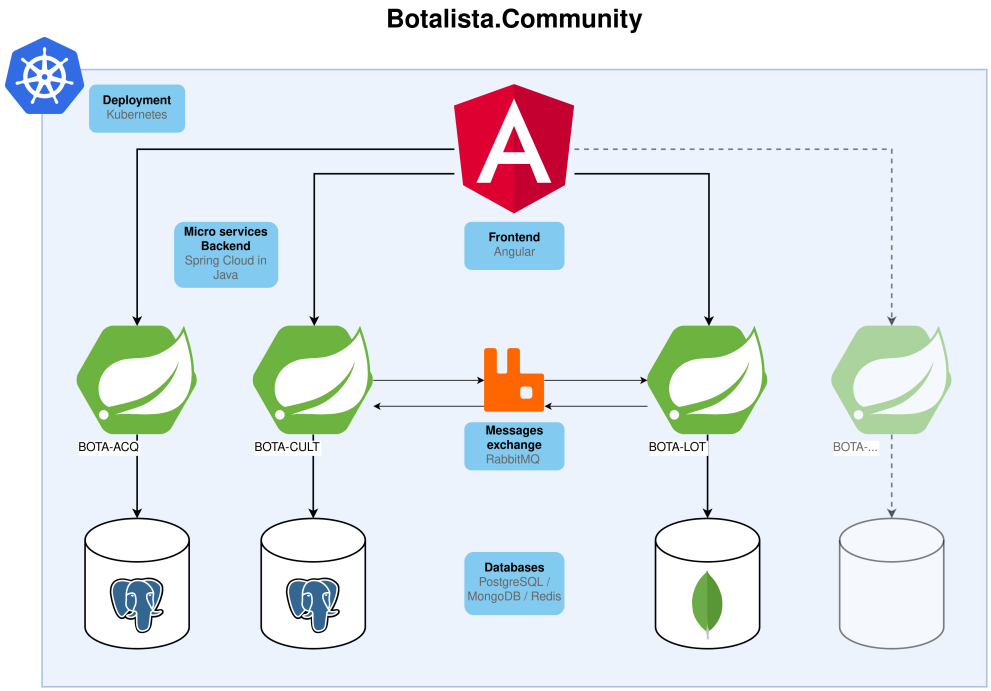

ILLUSTRATION 1.1 – Les technologies de Botalista. Source : Réalisé par Cyril BOILLAT

L'interface web est développée en Angular (avec également une partie de l'interface en AngularJS) et la majorité des modules sont développés en Java Spring Cloud. Ceux restant utilisent du PHP et certaines fonctionnalités sont gérées en tant que Function as a Service (FaaS) développées en Python ou R.

Les différents modules Java communiquent entre eux avec RabbitMQ, un agent de messages open source qui permet des communications asynchrones via différents types de protocoles comme AMQP. RabbitMQ est installé sous la forme d'un conteneur unique au sein du cluster Kubernetes.

Ces modules interagissent également avec trois serveurs de bases de données différents :

— **PostgreSQL** [6] pour la majorité (99%) des données qu'utilisent les différents modules ;

— **MongoDB** est utilisé pour stocker certaines données sous forme de JSON ;

— **Redis** pour gérer des données non-structurées, notamment pour de la traduction.

Ces serveurs de bases de données sont installées directement sur l'une des machines du cluster et ne sont donc pas conteneurisées comme les différents modules. À quelques exceptions

6. L'extension PostGIS est utilisée pour gérer des données géographiques.

près, chaque module d'une instance de *Botalista.software* possède sa propre base de données. Un module d'une instance ne peut pas accéder à la base de données d'une autre instance.

Actuellement, les bases PostgreSQL représentent un total d'environ 120 Go de données pour les 16 instances de production.

1.3. INFRASTRUCTURE RÉSEAU ET MATÉRIELLE

Cette section décrit en détail l'infrastructure actuelle de *Botalista.software*, représentée par l'illustration 1.2 et le tableau 1.1.

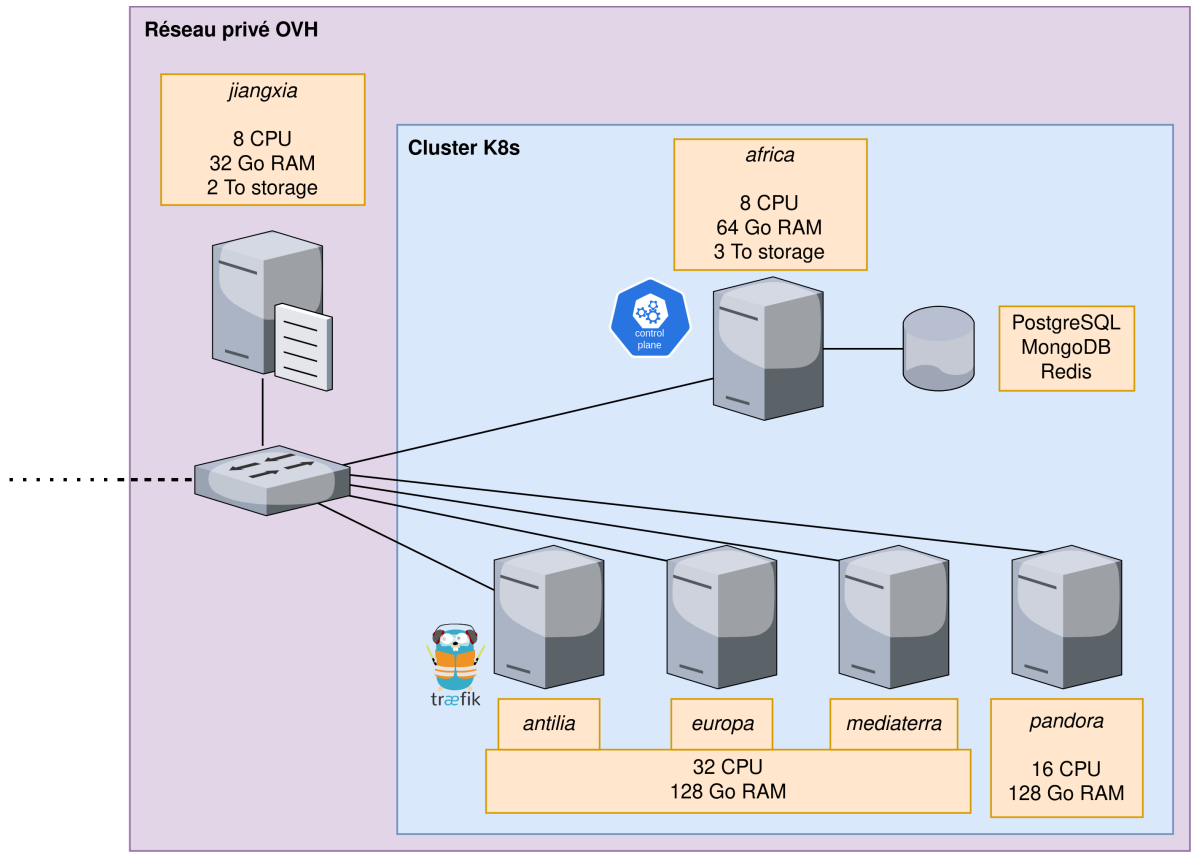

ILLUSTRATION 1.2 – L'infrastructure réseau et matérielle de Botalista. Source : Réalisé par Dario GENGA

Serveur	**CPU**	**RAM** (Go)	**Responsabilités**
africa	8	64	plan de contrôle, bases de données, serveur NFS
antilia	32	128	worker, Traefik, IP publique accessible
europa	32	128	worker
mediaterra	32	128	worker
pandora	16	128	worker
jiangxia	8	32	Registre docker, intégration continue, GED

TABLEAU 1.1 – Résumé de l'infrastructure matérielle. Source : Réalisé par Dario GENGA

La solution *Botalista.software* est déployée sur un réseau privé du fournisseur OVHcloud, composé de 6 serveurs physiques dont 5 qui composent le cluster Kubernetes.

Ces différents serveurs, à l'exception d'*antilia*, sont configurés pour rejeter toutes connexions provenant de l'extérieur du serveur privé OVH.

a. Cluster Kubernetes

Le cluster est composé de 5 serveurs :

— *africa* : nœud responsable du plan de contrôle, du serveur NFS et des bases de données ;

— *antilia* : worker qui gère le reverse proxy Træfik ;

— *europa* : worker ;

— *mediaterra* : worker ;

— *pandora* : worker, ajouté au cours du projet (août 2024).

Le nœud maître, appelé *africa*, gère à lui seul le plan de contrôle du cluster Kubernetes de Botalista et héberge les diverses bases de données. Ce nœud possède 3 To d'espace disque sur lequel sont actuellement stockées :

— 120 Go pour les bases de données (99% PostgreSQL, ~1% MongoDB et <1% Redis) ;

— 1.8 To pour des données spatiales utilisées par Open-Elevation [7] ;

— l'espace restant est mis à disposition pour stocker des images avec Alfresco [8], disponible au travers d'un serveur NFS.

Les nœuds *antilia*, *europa*, *mediaterra* et *pandora* sont les workers du cluster. Ils possèdent des ressources matérielles conséquentes afin d'assumer la charge qu'imposent les différentes instances de Botalista.

Le cluster est accédé par les utilisateurs de *Botalista.software* au travers du nœud *antilia* qui est le seul accessible depuis l'extérieur et sur lequel le reverse proxy Træfik est toujours déployé. Træfik redirige les requêtes des utilisateurs au module correspondant afin d'assurer leur accès peu importe le nœud sur lequel il a été déployé par Kubernetes.

b. Serveur *jiangxia*

Le nœud *jiangxia* est externe au cluster Kubernetes afin d'assumer 3 tâches principales :

— Héberger un registre privé contenant les images Docker des modules de Botalista ;

— Gérer des tâches d'intégration continue avec Jenkins ;

7. Alternative open source à Google Elevation API.
8. Logiciel de Gestion Électronique des Documents (GED). Installé sur le nœud *jiangxia*.

— S'occuper de la GED avec le logiciel Alfresco. Jusqu'à 5 Go de stockage sont offerts à chaque membre de l'association [9].

Le serveur *jiangxia* possède moins de ressources matérielles que les autres nœuds, soit seulement 8 CPU et 32 Go RAM. Il dispose cependant d'un disque avec 2 To d'espace de stockage, dont ~500 Go actuellement utilisé par Jenkins et les différentes images Docker stockées sur le registre privé.

1.4. LES INSTANCES ET MODULES DE BOTALISTA

Le tableau 1.2 présente brièvement les 28 modules qui composent *Botalista.software*.

Module	Description
BOTA-ACQ	Gestion des informations liées au matériel végétal nouvellement acquis
BOTA-ADM	Module d'administration de Botalista
BOTA-ADN	Gestion des informations liées à une banque d'ADN
BOTA-ANI	Gestion des informations liées aux animaux
BOTA-BIB	Gestion de références bibliographiques
BOTA-BS	Gestion des informations liées à une banque de semences
BOTA-CKL	Outil de gestion de checklists
BOTA-CMD	Gestion des commandes de matériel végétal
BOTA-COL	Gestion de collections vivantes
BOTA-COR	Gestion des informations liées aux correspondants
BOTA-CULT	Gestion des spécimens en culture
BOTA-DOC	Gestion des documents (texte, PDF, images, etc.)
BOTA-EXC	Gestion des échanges, dons et achats de parts
BOTA-EXIN	Gestion des informations liées aux programmes de conservation ex et in situ
BOTA-FLO	Gestion des informations liées aux projets de flores
BOTA-HER	Gestion des herbiers (prêts, achats, dons, etc.)
BOTA-IS	Gestion des Index seminum
BOTA-MAP	Gestion des données spatiales
BOTA-MOB	Interfaces de gestion « mobiles »
BOTA-NOM	Gestion de la nomenclature et de la synonymie par projets
BOTA-PART	Gestion des informations liées aux échantillons d'herbier
BOTA-PER	Gestion des informations liées aux personnes
BOTA-REC	Gestion des récoltes
BOTA-REP	Production de rapports
BOTA-TAX	Gestion des taxons
BOTA-TIME	Gestion des heures travaillées par projet, tâche, ...
BOTA-VIS	Outil d'administration pour la planification et la gestion de visites
BOTAVISTA	Portail de diffusion et d'agrégation de données botaniques

TABLEAU 1.2 – Liste des modules de *Botalista.Software*. Source : Réalisé par Dario GENGA

9. Il est possible d'augmenter le stockage disponible, mais à ce jour aucun membre n'y a eu recours.

a. Modules métiers et logiciels

L'équipe de développement de Botalista différencie les modules *métiers* et *logiciels* :

— Un module métier est unique au sein du cluster et partagé à toutes les instances de *Botalista.software* ;

— Un module logiciel correspond à un microservice d'une instance de *Botalista.software*.

b. Implémentation des modules au sein de Kubernetes

Le cluster Kubernetes de Botalista est composé de divers espaces de noms pour isoler les ressources Kubernetes appartenant à ces différents types de modules :

— `business-service` représente les modules *métiers* ;

— `botalista-<institut>-prd` correspond à une instance d'un membre de Botalista.

Des espaces de noms supplémentaires sont également mis en place pour isoler des composants additionnels au sein du cluster, notamment pour le développement.

Le tableau 1.3 contient les modules *métiers* actuellement déployés dans l'espace de nom `business-service`, pour une consommation totale d'environ 6 Go de RAM. Ils sont généralement utilisés au travers des modules *logiciels*.

Module	Description
`cit-checklist`	Octroie une liste de plantes
`cit-dwca`	Agrégation de nomenclature botanique (DWCA)
`cit-export-excel`	Génère un fichier excel à partir de JSON
`cit-formly`	Configuration des formulaires pour les instances
`cit-geocoding`	Gestion des données spatiales
`cit-nominatim`	Plateforme de géocodage
`cit-open-elevation`	API de données d'élévation
`cit-per`	Gestion centralisée des personnes
`cit-template`	Génération d'une liste de champs à exporter pour les instances
`cit-ticketing`	Gestion centralisée des tickets
`cit-traduction`	Gestion centralisée des traductions
`cit-uid-generator`	Génération d'UID pour l'ensemble des instances
`cit-vista-aggregation`	Agrégation des données sur la plateforme BOTAVISTA
`cit-wfo-aggregation`	Agrégation de nomenclature botanique (WFO)

TABLEAU 1.3 – Les modules métiers déployés. Source : Réalisé par Dario GENGA

L'implémentation des modules *logiciels* de *Botalista.software* au sein de Kubernetes est représentée par le tableau 1.4. Ces derniers n'ont pas forcément une équivalence direct avec les modules généraux du tableau 1.2.

Module	Description
`bota-acq`	Gestion des informations liées au matériel végétal nouvellement acquis
`bota-all`	Ensemble de divers modules (BOTA-BIB, BOTA-CMD, ...)
`bota-ani`	Gestion des informations liées aux animaux
`bota-col`	Gestion de collections vivantes
`bota-config`	Gestion de la configuration globale de l'instance
`bota-cult`	Gestion des spécimens en culture
`bota-doc`	Gestion des documents (texte, PDF, images, etc.)
`bota-exp`	Gestion de l'exportation
`bota-front`	Interface web, développée en Angular
`bota-frontend`	Interface web *legacy*, développée en AngularJS
`bota-hart`	Gestion des récoltes (BOTA-REC)
`bota-imp`	Gestion de l'importation
`bota-is`	Gestion des Index seminum
`bota-lot`	Gestion des lots de graines
`bota-map`	Gestion des données spatiales
`bota-mper`	Gestion des groupes de personnes (autorités, collectivités, ...)
`bota-nms`	Service de comparaison de noms
`bota-nom`	Gestion de la nomenclature et de la synonymie par projets
`bota-note`	Gestion des notes
`bota-orch`	Module d'administration de l'instance (BOTA-ADM)
`bota-per`	Gestion des informations liées aux personnes
`bota-proj`	Gestion des projets
`bota-rep`	Production de rapports
`bota-tax`	Gestion des taxons
`bota-time`	Gestion des heures travaillées par projet, tâche, ...
`bota-viz`	Outil d'administration pour la planification et la gestion de visites

TABLEAU 1.4 – Les modules logiciels qui composent une instance de *Botalista.software*. Source : Réalisé par Dario GENGA

L'intégralité des modules logiciels est dépendante de `bota-config` *afin d'être correctement configurée. Ce dernier est par conséquent toujours déployé en premier lors de la mise en place d'une nouvelle instance.*

Une instance de *Botalista.software* possède généralement 21 à 26 modules *logiciels*, pour une consommation en RAM qui varie en fonction des instances.

c. Présentation des instances

Le cluster Kubernetes déploie actuellement 16 instances de production, 1 de pré-production et 1 de démonstration. Chaque instance consomme en moyenne 13 Go de RAM, néanmoins certaines instances ont une consommation constante pouvant atteindre les 20 Go.

Le tableau 1.5 présente la consommation en RAM, mesurée le 15 août 2024, et le nombre de modules activés pour chaque instance déployées. Bien que la majorité des instances possèdent le même nombre de modules, soit 24 en moyenne, la consommation RAM varie grandement d'une instance à l'autre. Cette différence provient principalement du nombre d'utilisateurs concurrents et des données appartenant à chaque instance.

Région de l'institut	**Abréviation**	**Modules activés**	**Consommation RAM (Go)**
Berne (CH)	bern	23	12.5
Liège (BE)	blg	23	10.7
Bordeaux (FR)	bord	23	13
Tessin (CH)	briss	21	11
Soual (FR)	cbpf	23	9
Châtenay-Malabry (FR)	chat	23	10.4
Fribourg (CH)	fribg	23	13.3
Genève (CH)	g	26	20.4
Grenoble (FR)	gre	23	9.4
Paris (FR)	jbp	23	18.4
Metz (FR)	metz	23	9.7
Meyrin (CH)	mey	23	11.4
Nancy (FR)	ncy	23	15.6
Neuchâtel (CH)	neu	23	12.4
Nantes (FR)	ntm	23	14.8
Porrentruy (CH)	porty	23	9.3
Release candidate	*rc*	*26*	*10.2*
Bac à sable	*sandbox*	*22*	*17.7*

TABLEAU 1.5 – Consommation en RAM des instances de *Botalista.Software*. Source : Réalisé par Dario GENGA

d. Instance clone

Afin de former leurs utilisateurs, les membres de l'association peuvent demander le clonage de leur instance pendant une durée limitée. Cette opération duplique l'instance de production et créé des bases de données conteneurisées. Les modules du clone de l'instance se connectent aux bases conteneurisées, qui contiennent une copie récente des données de l'instance. Les utilisateurs peuvent ainsi être formés sans risquer des pertes de données en cas d'erreur de manipulations.

Cette opération est non-négligeable puisque le clone créé consomme autant de RAM que l'instance d'origine. En outre, plusieurs membres peuvent cloner simultanément leur instance.

1.5. Problèmes identifiés

L'infrastructure actuelle de Botalista contient divers problèmes qui empêchent d'assurer l'accès et l'utilisation du service en cas de panne d'un nœud. Elle est actuellement composée de deux machines critiques : *africa* et *antilia*. La moindre panne de l'une de ces deux machines signifie une interruption du fonctionnement de toutes les instances de *Botalista.software*.

Cette section présente les différents problèmes identifiés à ce jour et estime l'impact que provoquerait une panne pour chacun des nœuds.

a. Plan de contrôle à un seul nœud

Le plan de contrôle Kubernetes est uniquement dirigé par le serveur *africa*, tout comme les différentes bases de données et le serveur NFS. Ce nœud est donc le plus important de l'infrastructure, une panne de ce dernier empêche l'orchestration de l'intégralité du cluster, tout en empêchant l'accès aux bases de données et au serveur NFS.

b. Accès au service dépendant du nœud *antilia*

Le serveur *antilia* assume également un rôle indispensable pour Botalista, puisqu'il s'agit du nœud qui redirige les requêtes des utilisateurs vers les services demandés avec Træfik. Une défaillance de ce nœud signifie que l'accès et l'utilisation des instances *Botalista.software* deviennent impossibles pour tous les utilisateurs.

c. Bases de données sans réplications

Les bases de données sont uniquement disponible sur le nœud *africa*. Un arrêt de ce nœud provoque une indisponibilité des microservices dépendant des bases de données. Un arrêt peut également provoquer des pertes de données. Pour résoudre ce problème, il faut mettre en place une réplication des données sur des serveurs secondaires.

Cette réplication offrirait les avantages suivants :

— Assurer la disponibilité des bases de données ;

— Permettre de répartir les requêtes en lecture ou écriture sur des serveurs différents.

d. Volumes partagés dépendant d'un serveur NFS

Certains composants de Botalista nécessitent l'accès à un répertoire partagé, géré par le serveur NFS installé sur le nœud *africa*. Un arrêt de ce serveur NFS est un impact non-négligeable sur la santé du cluster :

— Les composants dépendant ne peuvent pas être créés si le serveur NFS est indisponible ;

— Les composants actuellement démarrés ne peuvent plus lire ou écrire de données dans le répertoire partagé, ce qui peut entraîner une perte de données.

Ci-dessous se trouve la liste des composants de Botalista ayant besoin du serveur NFS :

— Alfresco, à hauteur de 5 Go par instance ;

— Open-Elevation ;

— Un outil pour générer des questionnaires ;

— Nexus ;

— OpenERP (désormais Odoo) : outil de comptabilité ;

— OpenProject : gestion de projet ;

— pgAdmin : platforme d'administration pour PostgreSQL ;

— Sonar : surveille la qualité du code ;

— XWiki : pour la documentation ;

— Portainer : gestion des conteneurs ;

— Le site internet.

Il s'agit principalement de composants qui aident au développement, ils ne sont donc pas directement liés aux instances de Botalista. Néanmoins certaines fonctionnalités de *Botalista.software* sont dépendantes d'Alfresco et Open-elevation. Ainsi une panne affectant le serveur NFS impacterait également le bon fonctionnement des instances de Botalista.

e. Non-réplication des modules et applications

Les modules de *Botalista.software* sont déployés sans réplications. Cela signifie qu'une défaillance du pod (ou du nœud qui l'héberge) provoque une indisponibilité de ce dernier. Si la défaillance impacte le nœud, Kubernetes prendra 5 minutes avant de rétablir les pods concernés. L'impact d'une défaillance est variable : les modules *logiciels* n'affectent que les utilisateurs de l'instance concernée tandis que les modules *métiers* peuvent affecter l'ensemble des instances.

Les instances de *Botalista.software* dépendent de RabbitMQ pour établir des communications asynchrones entre les différents modules, et nécessitent Keycloak pour gérer l'authentification aux instances. Cette configuration transforme chaque nœud hébergeant ces applications en un point de défaillance unique.

f. Consommations en ressources RAM

Le dernier problème identifié est la consommation en ressources actuelle des workers, qui s'élève de 70% à 80% en consommation mémoire permanente [10]. Ceci est dû aux instances de *Botalista.software* qui nécessitent chacune entre 9 à 21 Go de RAM pour fonctionner.

Cette consommation élevée signifie qu'une panne de n'importe quel worker provoquerait, dans le pire des cas, une panne des autres, à cause du mécanisme d'auto-guérison de Kubernetes qui rétabli automatiquement les conteneurs défaillants sur un nœud disponible. Dans le cas où les nœuds resteraient opérationnels, la stabilité des instances de Botalista deviendrait incertaine selon les modules encore actifs suite à une panne.

À noter que l'implémentation actuelle de RabbitMQ entraîne un pic de consommation mémoire quotidien. Chaque nuit, les modules *logiciels* des instances de Botalista chargent en mémoire les données provenant de leur base PostgreSQL. En conséquence, la consommation RAM de chaque microservice augmente, jusqu'à 300 Mo pour certains modules. Ce processus s'effectue aléatoirement durant une plage horaire, l'impact mémoire global peut donc varier en fonction du nombre de microservice effectuant cette opération simultanément. Les données sont ensuite ajoutées à la pile RabbitMQ afin de les transmettre aux autres microservices. Ces opérations sont effectuées dans le but de synchroniser les données des microservices.

Ce procédé fut implémenté initialement afin de faciliter le développement de la solution, mais est cependant progressivement abandonné pour réduire l'impact mémoire qu'il provoque.

10. Ces valeurs ont été mesurées avant l'ajout du nœud *pandora* au cours du projet. Sans cet ajout, la consommation en RAM atteignait un seuil critique et provoquait des pannes telles que mentionnées dans le reste de cette section.

CHAPITRE 2 : ARCHITECTURE À HAUTE DISPONIBILITÉ

L'analyse effectuée au chapitre précédent a permis de mieux comprendre le fonctionnement de Botalista et d'identifier les problèmes actuels.

Une reproduction de l'existant, sur un environnement appartenant à l'HEPIA, a été effectué durant le projet de semestre pour prendre en main Kubernetes et étudier les améliorations envisageables pour l'architecture de Botalista.

Ce chapitre présente une nouvelle architecture qui a pour objectif de n'avoir aucun point de défaillance unique. Cette dernière est actuellement mise en place sur des machines virtuelles Ubuntu 22.04 sur le réseau de l'école. Un aperçu des nouveaux composants et des changements par rapport au déploiement actuel de Botalista est disponible en annexe 2.

2.1. INFRASTRUCTURE PROPOSÉE

L'illustration 2.1 et le tableau 2.1 représentent la nouvelle infrastructure proposée pour assurer la haute disponibilité de Botalista.

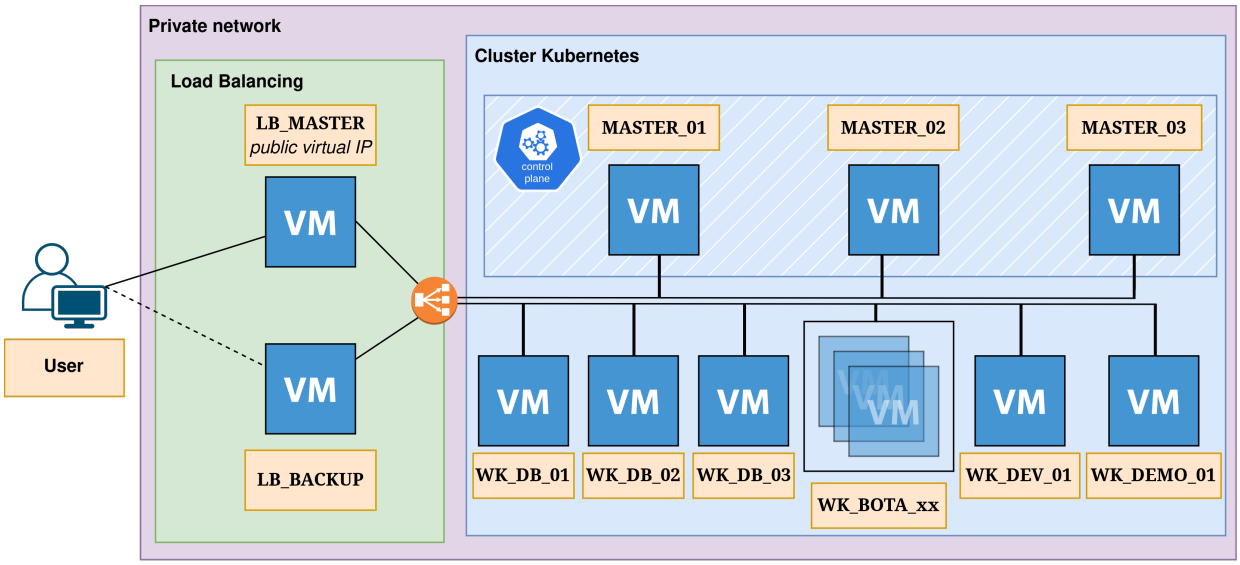

ILLUSTRATION 2.1 – Schéma de l'infrastructure à haute disponibilité proposée. Source : Réalisé par Dario GENGA

Cette nouvelle infrastructure utilise des machines virtuelles au lieu de serveurs physiques. Ceci permet d'ajouter ou retirer des workers à la demande, selon les ressources nécessaires, à un moment spécifique. Ces nœuds sont également séparés en 4 types (*tiny, small, medium, large*) selon leur spécification en RAM et CPU. Ceci permet d'ajuster au mieux les coûts de l'infrastructure par rapport aux ressources réellement utilisées.

Nom d'hôte	**Rôle**	**Type**	**CPU**	**RAM** (Go)
LB_MASTER	load balancing et registre Docker	*tiny*	2	4
LB_BACKUP	load balancing et registre Docker	*tiny*	2	4
MASTER_01	Plan de contrôle	*small*	4	8
MASTER_02	Plan de contrôle	*small*	4	8
MASTER_03	Plan de contrôle	*small*	4	8
WK_DB_01	worker et bases de données	*large*	8	32
WK_DB_02	worker et bases de données	*large*	8	32
WK_DB_03	worker et bases de données	*large*	8	32
WK_BOTA_xx	worker	*large*	8	32
WK_DEV_01	worker pour le développement	*large*	8	32
WK_DEMO_01	worker de présentation	*large*	8	32

TABLEAU 2.1 – Tableau de l'infrastructure à haute disponibilité proposée. Source : Réalisé par Dario GENGA

La nouvelle infrastructure apporte de nombreux changement par rapport à celle existante. Ces changements sont détaillés dans les sections suivantes de ce chapitre, tandis que la liste ci-dessous contient un bref résumé des changements principaux :

— Le plan de contrôle est désormais contrôlé par 3 nœuds (1 précédemment) ;
— Deux VMs, externes au cluster Kubernetes, permettent d'accéder aux instances de Botalista ainsi qu'à l'hébergement du registre Docker privé ;
— Les bases de données sont désormais conteneurisées et déployées, avec des instances de réplications, sur des workers dédiés ;
— Le serveur NFS n'est plus utilisé pour éviter un point de défaillance unique. Il est remplacé par Longhorn (un système de stockage pour Kubernetes) ;
— Un nœud est dédié au développement et à l'instance de pré-production, tandis qu'un deuxième nœud gère celle de présentation. Leur taux d'activité est inférieur aux autres nœuds afin de réduire les coûts de l'infrastructure.

Il serait souhaitable d'avoir une VM de type medium (4 CPU, 16 Go RAM) pour `WK_DEMO_01`*, mais ceci est actuellement insuffisant pour gérer l'instance de démonstration.*

Les logiciels Alfresco et Jenkins, déployé sur le nœud *jiangxia* de Botalista, ne sont pas représentés dans cette nouvelle infrastructure pour les raisons suivantes :

— L'achat d'une licence commerciale pour Alfresco est nécessaire afin de le déployer au sein du cluster Kubernetes ;
— J'ai choisi de ne pas dédier un nœud à Jenkins car ce dernier peut être installé sur Kubernetes.

2.2. CLUSTER KUBERNETES

Cette section décrit les modifications apportées à l'ensemble du cluster Kubernetes.

a. Plan de contrôle à haute disponibilité

Le plan de contrôle est l'élément le plus critique d'un cluster Kubernetes. Plusieurs machines doivent ainsi être responsables de ce dernier afin d'assurer la robustesse du système. Pour cela, la mise en place d'un plan de contrôle à haute disponibilité peut être effectuée selon deux topologies différentes représentées par les illustrations 2.2 et 2.3 :

— *etcd empilés*, chaque nœud du plan de contrôle possède son composant *etcd* ;

— *etcd externes*, des nœuds supplémentaires sont dédiés pour les composants *etcd*.

Pour ces deux topologies le principe est identique : un nombre impair de nœud est mis en place pour gérer le plan de contrôle et la redondance des données, les workers communiquent avec ces nœuds au travers d'un répartiteur de charge. La différence principale entre ces deux topologies est que la version avec des etcd externes est moins sensible aux pertes d'un nœud du plan de contrôle ou membre etcd. Elle requiert cependant deux fois plus de nœuds que la topologie empilée.

La raison pour laquelle ces topologies nécessitent un nombre impair de nœud est pour le quorum [11] requis par un cluster etcd qui utilise l'algorithme Raft. Pour un cluster avec n membres, le quorum est $\lfloor n/2 \rfloor + 1$. Ajouter un nœud à un cluster de taille impair augmentera toujours le nombre de nœud requis pour le quorum sans augmenter la tolérance de panne supportée.

Le tableau 2.2 représente les nœuds requis pour le quorum ainsi que la tolérance de pannes selon la taille du cluster.

Taille du cluster	Majorité	Tolérance de pannes
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3
8	5	3
9	5	4

TABLEAU 2.2 – Majorité et tolérance de pannes selon taille du cluster etcd. Source : Tiré de *etcd.io*, ref. URL07

11. «Nombre de votants nécessaire pour qu'une élection soit valable.» (larousse.fr)

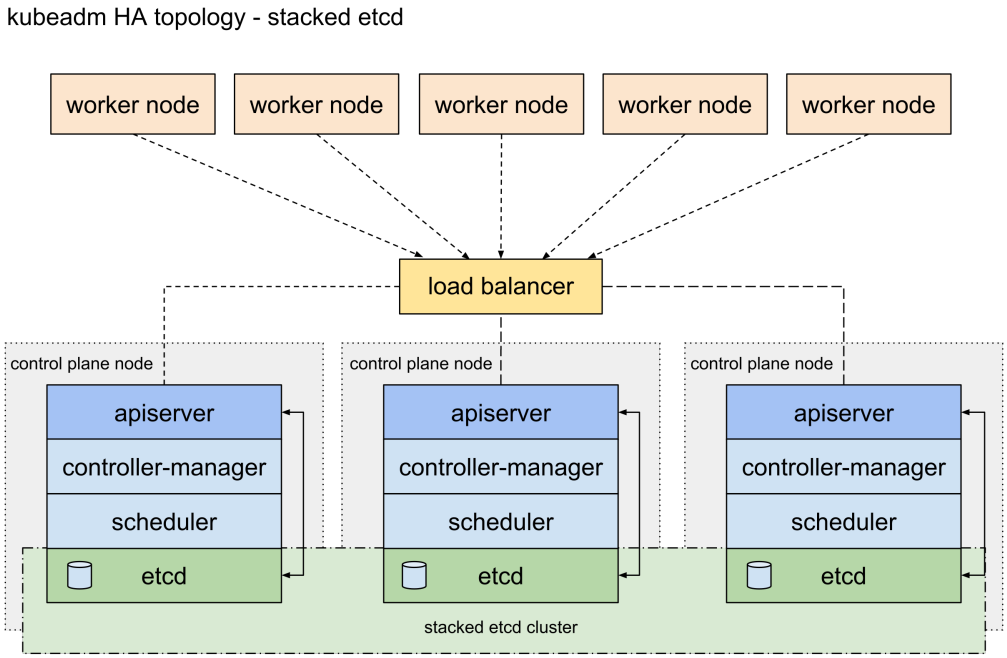

ILLUSTRATION 2.2 – Cluster à haute disponibilité - topologie empilée. Source : tiré de *kubernetes.io*, ref. URL06

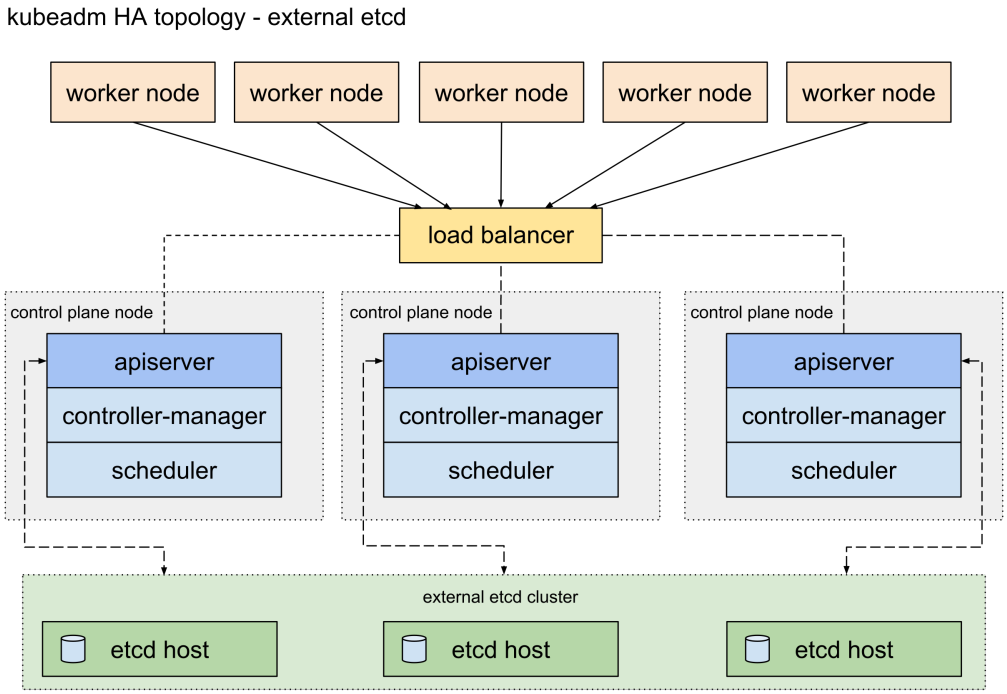

ILLUSTRATION 2.3 – Cluster à haute disponibilité - topologie externe. Source : tiré de *kubernetes.io*, ref. URL06

Pour la nouvelle architecture de Botalista, j'ai opté pour la topologie empilée avec 3 nœuds. Il s'agit de l'option la plus simple et la moins coûteuse à mettre en place afin d'assurer la haute disponibilité du plan de contrôle.

Concernant leurs caractéristiques, j'ai choisi des VMs de type *small*, soit 4 CPU et 8 Go RAM, afin de réduire les coûts tout en maintenant suffisamment de ressources disponibles pour gérer l'orchestration de Kubernetes. Ce choix est basé sur la consommation en mémoire des nœuds maîtres observée sur le cluster école, qui est d'environ 1.8 GB pour chaque nœud maître dans un cluster avec 7 workers et 200 pods.

b. Workers

Les caractéristiques et le nombre de workers peut varier au cours du temps. Il est cependant facile d'en ajouter ou d'en retirer puisqu'il s'agit de machines virtuelles, ce qui permet d'ajuster au mieux les ressources du cluster. Pour cette nouvelle architecture, je suggère dans un premier temps 12 VMs de type *large* (8 CPU et 32 Go RAM) qui assument le rôle de worker. Ceci permet de conserver la même quantité en ressources mémoires que celles des workers de Botalista (avant l'ajout du nœud *pandora*).

La nouvelle architecture utilise des bases de données conteneurisées et en cluster, avec une instance principale et deux de réplications. Puisque ces bases de données nécessitent un espace disque pouvant être conséquent, j'ai dédié 3 VMs aux déploiements des ces bases afin de faciliter leur configuration. D'autres applications peuvent également être déployées sur ces VMs.

Dans le but de réduire les coûts d'infrastructure, 2 nœuds sont dédiés aux instances de non-production :

- **`WK_DEV_01`** : l'instance de pré-production et les services en lien avec le développement sont uniquement déployés sur ce nœud, qui n'est actif que durant les heures de travail.
- **`WK_DEMO_01`** : l'instance de présentation est uniquement déployé sur ce nœud. Il devient actif lorsque l'on souhaite accéder à l'instance avant de se désactiver automatiquement après une période d'inactivité.

Kubernetes permet la mise en place de *taints* et *tolerations* pour contrôler le déploiement des pods sur les différents nœuds. Le listing 2.1 ajoute une *taint* sur les nœuds afin d'empêcher le déploiement de pods non voulus sur ces derniers.

```
1 kubectl taint nodes WK_DEV_01 node.role=dev:NoSchedule
2 kubectl taint nodes WK_DEMO_01 node.role=demo:NoSchedule
```

LISTING 2.1 – Ajout d'une *taint* pour les nœuds de développement et présentation

Il faut ensuite ajouter une *toleration* aux pods correspondants afin de les autoriser à être déployés sur ces nœuds. Ceci est représenté par le listing 2.2.

```
1 tolerations:
2 - key: "node.role"
3   operator: "Equal"
4   value: "dev"
5   effect: "NoSchedule"
```

LISTING 2.2 – Ajout d'une *toleration* au format YAML

Pour finir, l'ajout de la propriété `spec.nodeName: WK_DEV_01` et `spec.nodeName: WK_DEMO_01` permet de s'assurer que les pods soient uniquement déployés sur les nœuds appropriés.

c. Bases de données conteneurisées

Les bases de données PostgreSQL, MongoDB et Redis sont toutes déployées en tant que conteneurs et avec plusieurs instances, sur des nœuds différents, qui assurent la disponibilité des données. Chaque base possède une instance principale, qui autorise les requêtes en écriture et lecture, avec deux instances secondaires qui répliquent les données de la principale et autorisent les requêtes en lecture. La section 2.3 décrit le fonctionnement de la réplication et le mécanisme de failover qu'effectuent chaque base de données.

Afin de définir les nœuds sur lesquels les bases de données doivent être déployées, il faut mettre en place des règles d'affinité et d'anti-affinité [12]. Elles permettent d'autoriser, ou non, le déploiement d'un pod selon le label d'un nœud ou des pods déjà présent sur un nœud. Les commandes du listing 2.3 permettent d'ajouter un label aux nœuds des bases de données.

```
1 kubectl label node WK_DB_01 node-role.kubernetes.io/db-host=
2 kubectl label node WK_DB_02 node-role.kubernetes.io/db-host=
3 kubectl label node WK_DB_03 node-role.kubernetes.io/db-host=
```

LISTING 2.3 – Ajout d'un label pour les nœuds qui gèrent les bases de données

Le code 2.4 représente une affinité qui oblige le déploiement d'un pod sur un nœud possédant le label `node-role.kubernetes.io/db-host` créé précédemment. Ce code illustre également une règle d'anti-affinité qui empêche le déploiement d'une instance PostgreSQL sur un nœud qui possède déjà un pod ayant le label `app=postgres`.

```
1 spec:
2   scheduler:
3     affinity:
4       nodeAffinity:
```

12. Les *taints* et *tolerations* sont trop restrictifs. On souhaite garder la possibilité de déployer des modules de Botalista sur les nœuds dédiés aux bases de données.

```
5         requiredDuringSchedulingIgnoredDuringExecution:
6           nodeSelectorTerms:
7             - matchExpressions:
8                 - key: node-role.kubernetes.io/db-host
9                   operator: Exists
10      podAntiAffinity:
11        requiredDuringSchedulingIgnoredDuringExecution:
12          - labelSelector:
13              matchExpressions:
14                - key: app
15                  operator: In
16                  values:
17                    - postgres
18            topologyKey: kubernetes.io/hostname
```

LISTING 2.4 – L'affinité et anti-affinité pour les instances des bases de données PostgreSQL

d. Stockage distribué

L'une des difficultés lors de la mise en place manuel d'un cluster Kubernetes est la gestion du stockage des applications déployées. Lorsqu'il est géré par un fournisseur (tel que Google ou Amazon) ce dernier dispose automatiquement des composants pour lier le stockage des applications à celui proposé par le fournisseur. Lorsque Kubernetes est déployé manuellement, c'est à l'administrateur du cluster de mettre en place les composants requis pour gérer le stockage des applications.

Botalista gère actuellement ce stockage au travers d'un serveur NFS, sur lequel des volumes persistants sont créés pour les composants qui en ont besoin. Cette approche est simple à mettre en place, mais a le défaut de représenter un point de défaillance unique.

À ce jour, une panne de ce serveur NFS n'affecte que certaines fonctionnalités mineures des instances de Botalista. Néanmoins la nouvelle architecture déploie des bases de données conteneurisées qui nécessitent des volumes persistants fournis par Kubernetes.

L'accès aux données de ces bases est crucial et nécessite donc une solution plus robuste qu'un serveur NFS.

Cette solution est un système de stockage distribué avec réplication, où les données d'un volume sont copiées intégralement sur plusieurs nœuds. La réplication est indispensable afin d'assurer la disponibilité des volumes. Sans ce mécanisme, une panne d'un nœud ayant un volume signifie une perte d'accès aux données s'y trouvant.

Il existe différentes solutions, open-source ou propriétaire, permettant d'implémenter un

stockage distribué avec réplication au sein de Kubernetes. Les deux plus populaires sont Rook et Longhorn.

Rook est un orchestrateur open source qui implémente Ceph. Cette solution est mature, très robuste et performante. C'est généralement la solution recommandée pour la mise en place d'un stockage distribué. Ces nombreux avantages impliquent cependant une grande complexité et des contraintes conséquentes pour être mis en place, telles que la nécessité d'avoir un disque sans partitions ni système de fichier.

Longhorn est un système de stockage distribué pour Kubernetes. Cette solution est bien moins mature que Ceph et dépend de Kubernetes pour fonctionner. Elle est néanmoins bien plus simple à mettre en place, n'a aucune contrainte sur les partitions, tout en étant légère et robuste.

J'ai choisi Longhorn pour la nouvelle architecture de Botalista. Bien que cette solution soit moins performante et stable que Rook, elle me semble plus adaptée aux besoins de l'association au vu de sa simplicité.

e. Longhorn

Longhorn est un système de stockage distribué pour Kubernetes, qui se veut léger, fiable et puissant. Ce système est implémenté en utilisant des conteneurs et microservices. Longhorn créé un contrôleur de stockage dédié, appelé *Longhorn Engine*, pour chaque volume et le réplique de façon synchrone sur plusieurs *replicas* se trouvant sur différents nœuds. Le contrôleur et les *replicas* sont orchestrés par Kubernetes.

L'illustration 2.4 représente 3 volumes Kubernetes, chacun régit par leur propre *Longhorn Engine*. Ces derniers dupliquent les données dans 2 *replicas* différents qui sont stockés dans un disque SSD.

Un même disque peut contenir plusieurs *replicas*, néanmoins chaque *replicas* d'un même volume doit se situer sur un nœud différent afin d'assurer la disponibilité des données. Le système de Longhorn garantit l'accès aux données d'un volume tant qu'un *replica* au minimum est fonctionnel.

Lors de la mise en place de Longhorn, son nombre de *replicas* par défaut est de 3. Cette valeur n'est cependant pas pertinente pour les volumes des différentes bases de données puisque ces dernières ont déjà leur propre mécanisme de réplication.

J'ai par conséquent configuré Longhorn afin qu'il autorise 2 approches lors de la mise en place d'un volume :

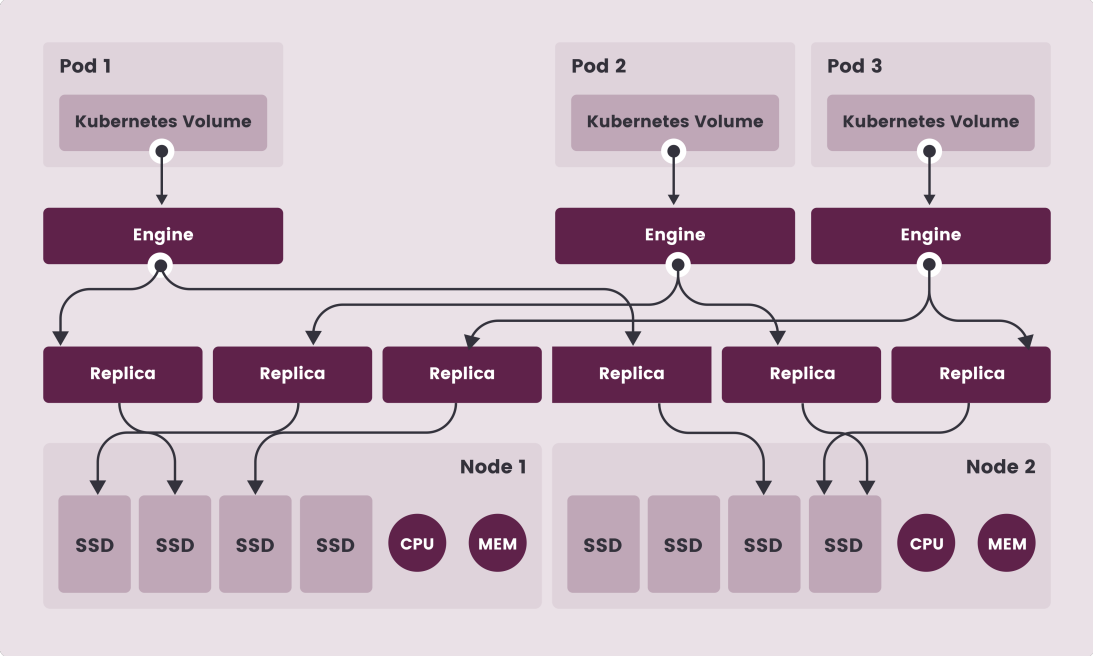

ILLUSTRATION 2.4 – Architecture de Longhorn. Source : tiré de *longhorn.io*, ref. URL09

— Pour les volumes des bases de données : seulement 1 *replicas* est créé et se trouve obligatoirement sur le même nœud du pod en ayant besoin ;
— Pour les autres volumes : 2 *replicas* sont créés. Ce nombre est un compromis pour avoir une réplication des données tout en réduisant l'espace total requis au sein du cluster.

Les commandes pour mettre en place et configurer Longhorn sont disponibles en annexe 3.

2.3. BASES DE DONNÉES AVEC RÉPLICATION

La réplication des différents serveurs de bases de données suit le même principe : une instance principale accepte les requêtes en lecture/écriture et transmet les changements aux instances secondaires.

L'un des objectifs de cette réplication est d'assurer un accès en lecture des données même en cas d'une défaillance de l'instance principale. Un autre objectif est la gestion du failover qui, suite à une panne de l'instance principale, permet de rétablir le cluster en effectuant la promotion d'une instance secondaire en instance principale.

Les sections suivantes décrivent en détail l'implémentation des bases de données au sein du cluster Kubernetes et explique le fonctionnement du failover de ces dernières. L'installation et la configuration de ces bases sont expliquées dans l'annexe 4.

a. PostgreSQL

PostgreSQL est déployé avec Kubegres, un opérateur Kubernetes qui permet le déploiement d'instances PostgreSQL en cluster avec réplication. Kubegres créé 2 services pour accéder à ces instances, un pour celle principale et un autre pour celles secondaires. L'illustration 2.5 représente ces services pour un cluster avec une instance principale et deux de réplications.

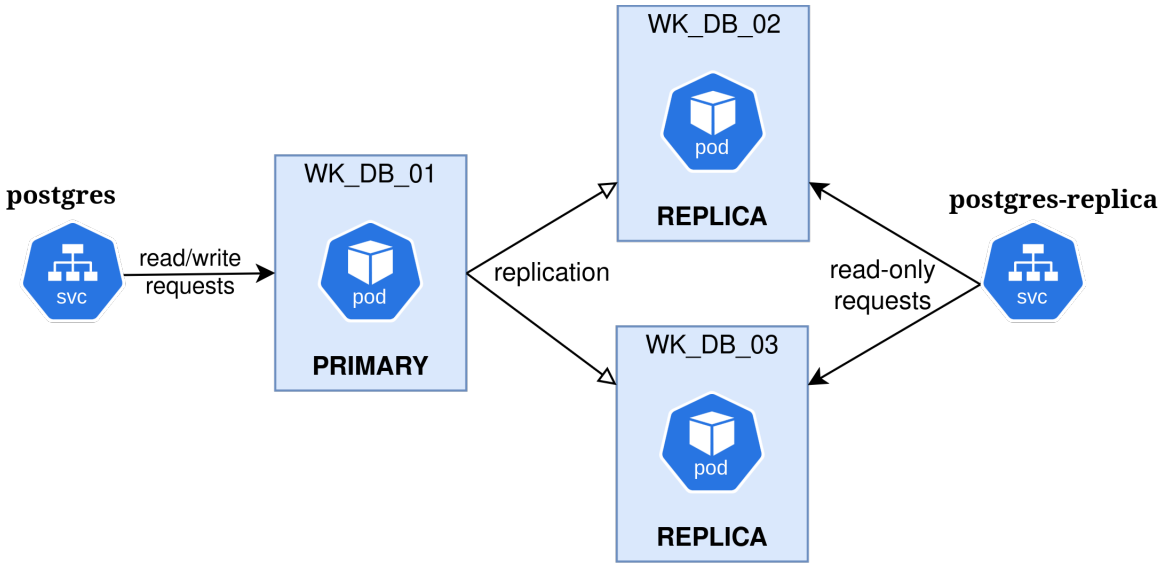

ILLUSTRATION 2.5 – Schéma du cluster PostgreSQL et des services Kubernetes associés. Source : Réalisé par Dario GENGA

Une fois mis en place, l'opérateur Kubegres surveille en permanence si l'instance primaire est opérationnelle. Pour cela, il s'appuie sur :

— La santé du conteneur Postgres, avec l'exécutable pg_ready fourni par PostgreSQL ;

— Les notification de Kubernetes pour avertir la panne d'un nœud.

En cas d'une panne de l'instance principale, Kubegres démarre le processus de failover en 2 étapes :

1. Kubegres vérifie qu'une instance secondaire est disponible. S'il en trouve une alors cette dernière est promue en instance principale.
2. Une fois la promotion effectuée, Kubegres déploie une nouvelle instance secondaire sur un nœud disponible.

Afin de mitiger l'impact du failover pour les clients, la documentation de Kubegres recommande les approches suivantes :

— Les applications doivent se connecter aux bases de données PostgreSQL via le service Kubernetes principal. Lors d'un failover, ce service se reconnecte automatiquement à la

nouvelle instance primaire ;

— Durant la période de promotion, certaines requêtes en lecture/écriture pour l'instance primaire pourraient échouer. Il est ainsi encouragé de mettre en place un mécanisme pour les exécuter à nouveau afin qu'elles puissent être validées une fois la nouvelle instance primaire disponible ;

— Les clients exécutant des requêtes en lecture seule doivent se connecter depuis le service Kubernetes de réplication afin qu'elles ne soient pas affectées durant un failover.

b. MongoDB

MongoDB est déployé avec son *Community Kubernetes Operator*. Il met en place un *Replica Set*, un groupe de processus *mongod* qui possèdent les mêmes données, composé d'une instance primaire pour les requêtes en lecture/écriture et plusieurs instances secondaires qui n'autorisent que les requêtes en lecture.

Toutes les instances communiques entre-elles afin de s'assurer qu'elles soient toutes opérationnelles. Lorsqu'une instance principale ne communique pas pendant plus de 10 secondes, une instance secondaire éligible demande une élection pour se désigner comme nouvelle instance principale. L'illustration 2.6 représente les communications entre chaque membre d'un *Replica Set*, tandis que l'illustration 2.7 représente l'élection d'une nouvelle instance principale.

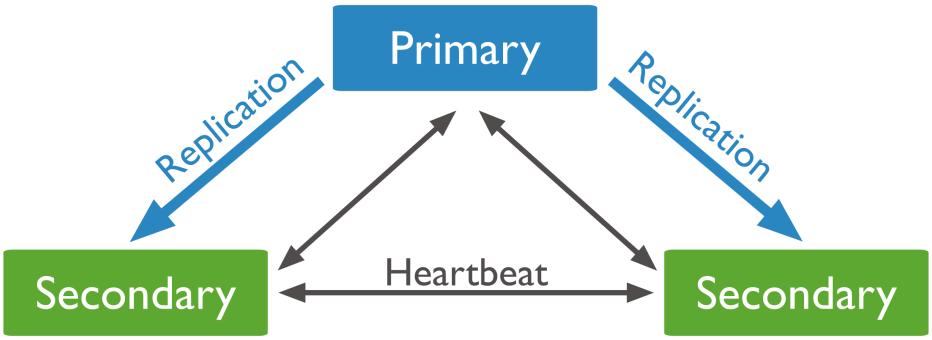

ILLUSTRATION 2.6 – Un *Replica Set* MongoDB composé d'une instance primaire et deux secondaires. Source : tiré de *mongodb.com*, ref. URL10

Le *Community Kubernetes Operator* permet de créer un *Replica Set* et expose un service Kubernetes pour s'y connecter. Néanmoins ce service à lui seul n'offre pas de distinction entre une instance principale et secondaire, ce qui peut provoquer des erreurs pour les requêtes en écritures. Pour y remédier, les applications doivent se connecter aux instances en spécifiant qu'il s'agit d'un *Replica Set*. MongoDB redirige automatiquement les requêtes en écriture à l'instance principale lorsqu'un client est connecté à un *Replica Set*.

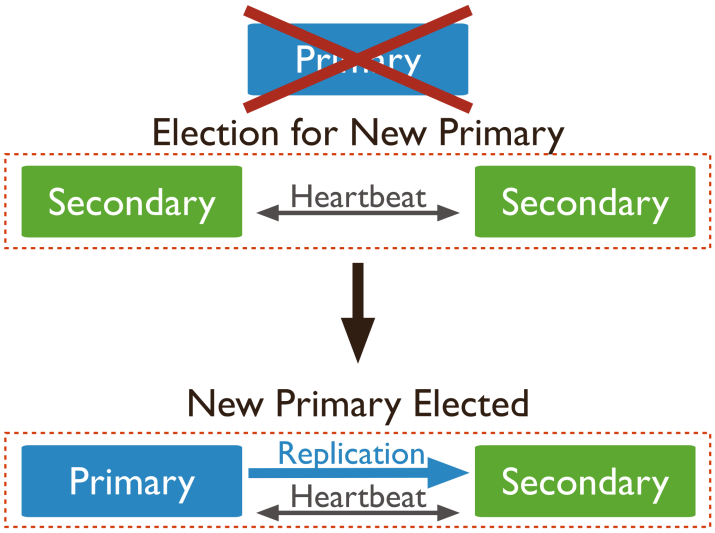

ILLUSTRATION 2.7 – Élection d'une nouvelle instance primaire pour un *Replica Set* MongoDB. Source : tiré de *mongodb.com*, ref. URL10

c. Redis

Redis est déployé avec le Helm chart de Bitnami [13] qui déploie un cluster *maître-replicas*. Ce cluster est configuré pour déployer des instances supplémentaires en mode *Sentinel*, qui surveillent l'instance Redis principale et ses *replicas*. Si les *Sentinels* détectent une défaillance de l'instance principale, alors elles enclenchent un failover automatique afin de promouvoir une instance secondaire en principale.

L'illustration 2.8 représente un cluster Redis composé d'une instance principale en rouge, deux instances de réplication en gris et trois processus *Sentinel* en vert. L'instance principal transmet les changements aux instances secondaires, qui gardent donc une copie des données. Les *Sentinels* surveillent l'instance principale et ses *replicas*. Si une majorité de processus *Sentinels* détectent une défaillance pour l'instance principale alors le *replica* ayant les données les plus récentes est promu en maître.

Tout comme MongoDB, seulement un service Kubernetes est exposé afin de se connecter au cluster Redis. Le port 6379 de ce service autorise les requêtes en lecture, tandis que le port 26379 permet d'accéder aux *Sentinels*. Pour effectuer des requêtes en écriture, il est nécessaire d'exécuter la commande 2.5 ci-dessous pour récupérer le maître actuel.

```
1 SENTINEL get-master-addr-by-name <name of your MasterSet>
```

LISTING 2.5 – Commande pour récupérer le Redis maître

13. Une plateforme, appartenant à VMware, qui facilite le déploiement d'applications web open-source grâce à des packages préconfigurés pour divers environnements (cloud, conteneurs, machines virtuelles).

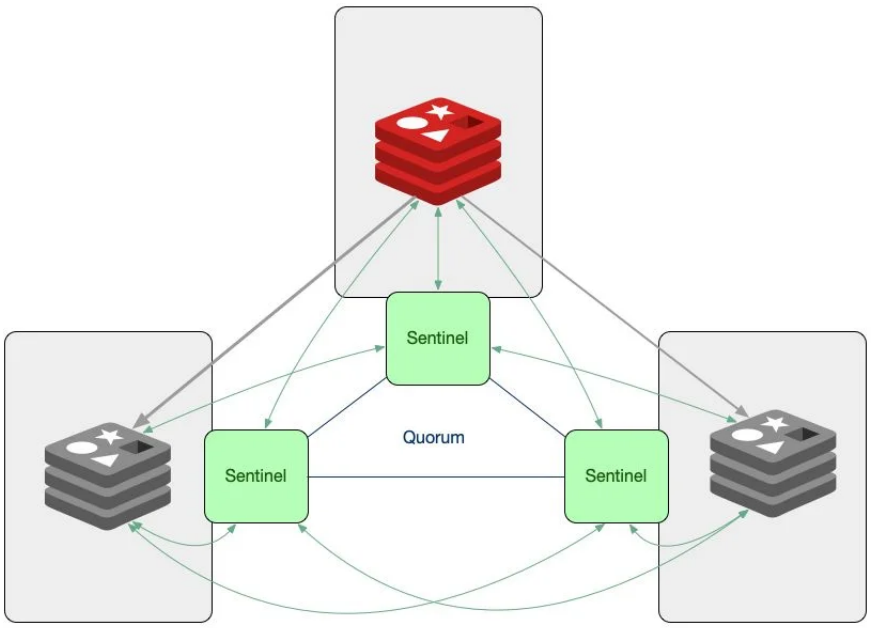

ILLUSTRATION 2.8 – Représentation d'un cluster Redis avec *Sentinel*. Source : tiré de *redis.io*, ref. URL11

2.4. ACCÈS AU CLUSTER

L'un des problèmes majeur à résoudre est l'accès au cluster Kubernetes de Botalista. La nouvelle architecture résous ce problème avec le logiciel HAProxy, qui fournit une répartition de charge (load balancing) et proxy à haute disponibilité. Il redirige les requêtes HTTP ou TCP reçues et les répartit sur plusieurs serveurs.

Le point d'entrée du cluster est désormais régulé par HAProxy à la place de Træfik (bien qu'il soit toujours utilisé). Cet accès ne doit cependant pas être géré par qu'une seule machine afin d'éviter un point de défaillance unique.

Une solution pour assurer la disponibilité du répartiteur de charge est l'utilisation d'une adresse IP virtuelle (VIP). Cette dernière est partagée, au minimum, entre deux machines. Le principe est d'attribuer cette adresse VIP à une machine principale (appelée *MASTER*). Si cette dernière devient inaccessible, alors une machine secondaire (appelée *BACKUP*) prend le relais en récupérant l'adresse VIP.

Cette gestion de l'adresse VIP est effectuée avec le logiciel Keepalived, qui gère cette attribution en surveillant l'état de santé des machines.

La mise en place de HAProxy et Keepalived sur un ensemble de machines, externes au cluster Kubernetes, représente ainsi un point d'entrée à haute disponibilité avec un mécanisme de répartition de charge. Le fait d'utiliser des machines externes offres plusieurs avantages :

— Les nœuds du cluster ne nécessitent pas un accès direct depuis l'extérieur, ce qui permet de bloquer les connexions externes et ainsi augmenter la sécurité des serveurs ;
— En cas de panne, seulement 1 type de composant est impacté, ce qui améliore la robustesse du système.

a. HAProxy

HAProxy est un logiciel libre et open source qui fournit un équilibreur de charge et un proxy à haute disponibilité, pour les applications TCP et HTTP, qui répartit les demandes sur plusieurs serveurs.

La configuration de HAProxy a généralement 3 objectifs principaux :
— Définir les adresses IP et ports à écouter pour la réception du trafic. Correspond au *frontend* ;
— Définir une plage de serveurs sur lequel rediriger le trafic. Correspond au *backend* ;
— Définir des règles spécifiques pour certaines requêtes utilisateurs. Correspond à l'*Access Control List (ACL)*.

La mise en place d'un plan de contrôle à haute disponibilité nécessite un répartiteur de charge (tel que HAProxy) qui expose le *kube-apiserver* (port 6443) pour les workers.

La configuration 2.6 ci-après de HAProxy (fichier *haproxy.cfg*) met en place ce fonctionnement.

```
frontend apiserver
    bind :6443
    mode tcp
    option tcplog
    default_backend controlplane


backend controlplane
    option httpchk
    http-check connect ssl
    http-check send meth GET uri /healthz
    http-check expect status 200
    mode tcp

    balance roundrobin
    server master1 10.136.26.73:6443 check verify none
    server master2 10.136.26.74:6443 check verify none
    server master2 10.136.26.75:6443 check verify none
```

LISTING 2.6 – HAProxy configuration pour plan de contrôle à haute disponibilité

Cette configuration définit un *frontend* appelé *apiserver* qui écoute toutes les requêtes TCP sur le port 6443 pour les transmettre au *backend* nommé *controlplane*. Ce dernier redirige les requêtes aux différents nœuds en appliquant une répartition de charge avec l'algorithme [14] *round robin*, représenté par l'illustration 2.9.

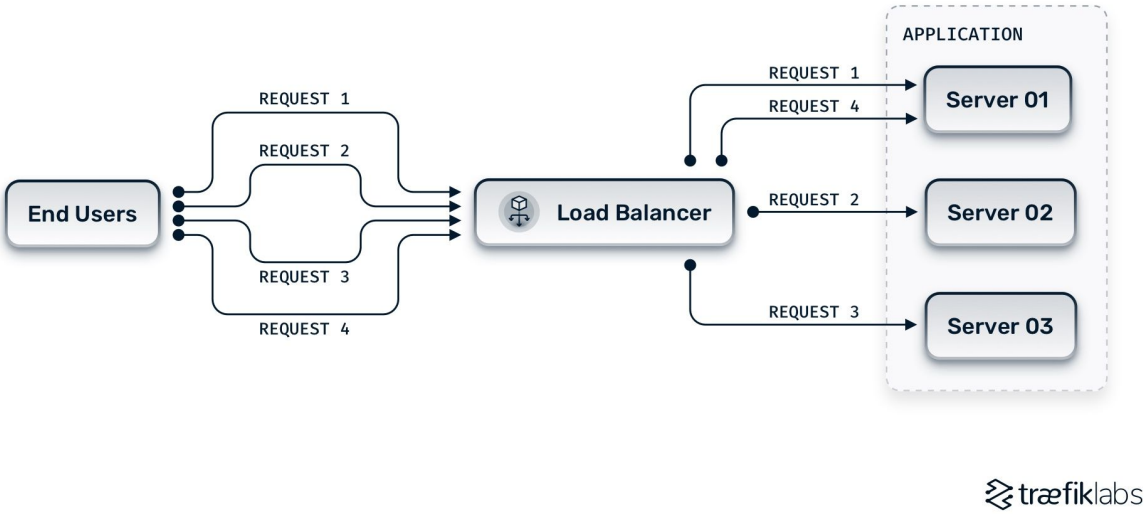

ILLUSTRATION 2.9 – Répartition de charge par *round robin*. Source : tiré de *traefik.io*, ref. URL08

Cette configuration transmet la première requête reçue au nœud *master1*, la deuxième au nœud *master2* et la troisième au *master3* avant de recommencer avec *master1*. Concernant les connexions HTTP (port 80), la configuration 2.7 suivante de HAProxy transmet ces dernières aux workers sur le port 30080, qui représente un service *NodePort* dédié à Traefik.

```
frontend default
    bind :80
    mode http
    option httplog
    default_backend traefik

backend traefik
    balance roundrobin
    server worker1 10.136.26.76:30080 check
    server worker2 10.136.26.77:30080 check
    server worker3 10.136.26.78:30080 check
```

LISTING 2.7 – HAProxy configuration HTTP simple

14. La liste des algorithmes supportés est disponible dans le manuel de configuration (docs.haproxy.org).

La section «c. Détails de l'implémentation» explique en détail le fonctionnement qui permet d'accéder à une instance de Botalista.

b. Keepalived

Keepalived est un daemon de surveillance et secours pour cluster de serveurs virtuels Linux (LVS) qui offre des fonctionnalités de haute disponibilité et d'équilibrage de charge à l'aide du protocole de redondance de routeur virtuel (VRRP).

Dans le cadre de ce projet, Keepalived est utilisé pour attribuer une adresse IP virtuelle à un nœud gérant HAProxy. Pour cela, Keepalived octroie un niveau de priorité à la machine (défini initialement dans son fichier de configuration), qui est influencé par l'état de santé du processus HAProxy. Le nœud ayant la plus grande priorité se voit attribuer l'adresse IP virtuelle.

La configuration 2.8 suivante, utilisée sur la machine `LB_MASTER`, met en place une instance VRRP qui surveille le processus HAProxy.

```
vrrp_track_process haproxy {
    process haproxy
    weight 100
}
vrrp_instance haproxy-vip {
    state MASTER
    interface ens160
    virtual_router_id 51
    priority 101
    advert_int 1
    virtual_ipaddress {
        10.136.26.70
    }
    track_process {
        haproxy
    }
}
```

LISTING 2.8 – Keepalived configuration pour la machine principale

La machine `LB_BACKUP` possède une configuration similaire, elle défini cependant son état (*state*) initial à BACKUP et sa priorité (priority) à 51.

La documentation de Keepalived recommande une différence d'au moins 50 de priorité entre les instances MASTER et celles BACKUP. Par conséquent le processus HAProxy se voit attribuer un poids (weight) de 100, qui est ajouté à la priorité lorsque le processus est fonctionnel.

— Lorsque le processus HAProxy est correctement démarré sur chaque serveur : le nœud `LB_MASTER` obtient 201 en priorité contre 151 pour `LB_BACKUP` ;

— Si HAProxy s'arrête sur `LB_MASTER` alors sa priorité descend à 101 ;

— Si HAProxy s'arrête sur `LB_BACKUP` alors sa priorité descend à 51 ;

— L'adresse IP `10.136.26.70` [15] est ajoutée au nœud ayant la plus haute priorité.

Pour l'instant seulement deux machines sont utilisées pour assurer la disponibilité de HAProxy. Il est bien évidemment possible d'en ajouter d'autres afin d'augmenter la robustesse du système.

c. Détails de l'implémentation

Actuellement, l'accès à une instance sur le cluster de Botalista est dirigé par Træfik sur le nœud *Antilia*. Son accès se fait par le biais d'un service `LoadBalancer` dont l'adresse IP externe est défini manuellement pour correspondre à l'IP publique du nœud *Antilia* (`146.59.253.176`). Træfik redirige les requêtes reçues, selon leur hôte [16] et chemin de l'URL, au service (type *ClusterIP*) web de l'instance correspondante. Ceci est implémenté avec les IngressRoute de Træfik.

La nouvelle architecture garde la même approche mais possède désormais 3 instances de Træfik accessible au travers d'un service `NodePort` ayant le port 30080 en HTTP et 30443 en HTTPS. La mise en place d'un service `NodePort` est nécessaire puisque les nœuds qui gèrent HAProxy ne peuvent pas communiquer directement avec les services Kubernetes au travers de leur *cluster IP*. Ces changements permettent de ne pas exposer publiquement un nœud du cluster tout en évitant d'en être dépendant pour accéder à une instance, ce qui augmente la robustesse du système.

Pour implémenter cette approche, chaque espace de nom d'une instance de Botalista doit créer un objet `IngressRoute` qui redirige les requêtes sur le service correspondant à l'URL de la requête.

```
1  apiVersion: traefik.io/v1alpha1
2  kind: IngressRoute
3  metadata:
4    name: traefik-ir
5    namespace: traefik
6  spec:
7    entryPoints:
8      - web
9    routes:
10   - match: Host(`traefik.bota-hepia.ch`)
11     kind: Rule
```

15. Cette adresse est privée dans l'environnement école mais elle doit être publique pour celui de Botalista.
16. Le nom de domaine du serveur auquel la requête est envoyée.

```
12    services:
13    - name: whoami
14      port: 80
```

LISTING 2.9 – Exemple d'un objet `IngressRoute` au format YAML

L'exemple 2.9 précédent définit une route `traefik.bota-hepia.ch`. Lorsque Træfik détecte une requête pour cette dernière, il la transmet au service *whoami* qui doit se trouver dans le même espace de nom que l'`IngressRoute`. Cette contrainte provient des ingress de Kubernetes qui ont une visibilité limitée à leur espace de nom. Ceci signifie qu'il est nécessaire de créer un objet `IngressRoute` dans chaque espace de nom que l'on souhaite accéder au travers de Træfik.

Pour résumé le fonctionnement, HAProxy reçoit une requête d'un client et la transmet aux workers sur le port 30080 (ou 30443 si HTTPS) qui correspond au service `NodePort` de Træfik. Ce dernier observe l'URL de la requête afin de la rediriger au service approprié pour finalement accéder à l'application. L'illustration 2.10 représente les composants Kubernetes qui gèrent ce fonctionnement.

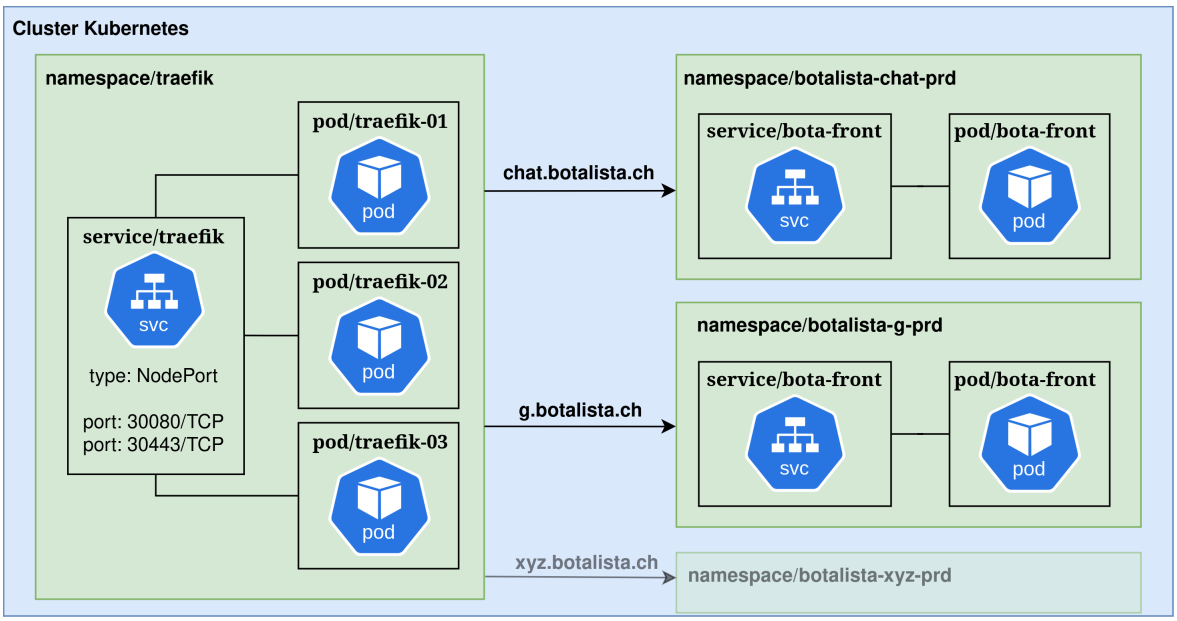

ILLUSTRATION 2.10 – Schéma des composants Kubernetes permettant d'accéder à une instance de *Botalista.software*. Source : Réalisé par Dario GENGA

Afin de gérer les accès non-autorisés, Botalista utilise Keycloak pour protéger les instances (et autres services) par une page d'authentification. Ce fonctionnement n'est pas représenté dans le schéma précédent pour en simplifier sa lecture.

Cette approche offre beaucoup de souplesse et robustesse au travers des fonctionnalités de HAProxy et Træfik, tout en n'ayant besoin que d'une seule adresse IP publique pour accéder au cluster Kubernetes.

2.5. RABBITMQ

Botalista ne déploie qu'un seul conteneur RabbitMQ pour gérer ses communications asynchrones entre microservices. Bien que cela soit suffisant pour le bon fonctionnement des instances de *Botalista.software*, ce conteneur RabbitMQ est actuellement un point de défaillance unique. Pour assurer sa haute disponibilité, son déploiement s'effectue désormais à partir du *RabbitMQ Cluster Kubernetes Operator*.

Cette section met en place un cluster RabbitMQ composé de 3 instances.

a. Installation de l'opérateur

La commande 2.10 permet d'installer l'opérateur Kubernetes.

```
kubectl apply -f https://github.com/rabbitmq/cluster-operator/releases/
    latest/download/cluster-operator.yml
```

LISTING 2.10 – Installation du RabbitMQ Cluster Kubernetes Operator

b. Création du cluster RabbitMQ

Une fois l'opérateur installé, il est possible de créer un objet `RabbitmqCluster` pour déployer nos instances RabbitMQ.

```
apiVersion: rabbitmq.com/v1beta1
kind: RabbitmqCluster
metadata:
  name: rabbitmq
  namespace: rabbitmq
spec:
  replicas: 3
  image: rabbitmq:3.13.6-management
  persistence:
    storageClassName: longhorn
    storage: 1Gi
  resources:
    requests:
      cpu: 200m
      memory: 500Mi
    limits:
      cpu: 500m
      memory: 1Gi
  rabbitmq:
    additionalConfig: |
      disk_free_limit.absolute = 50MB
```

```
22  affinity:
23    podAntiAffinity:
24      requiredDuringSchedulingIgnoredDuringExecution:
25      - labelSelector:
26          matchExpressions:
27          - key: app.kubernetes.io/name
28            operator: In
29            values:
30            - rabbitmq
31        topologyKey: kubernetes.io/hostname
```

LISTING 2.11 – Un objet RabbitmqCluster avec 3 replicas au format YAML

Le listing 2.11 met en place un cluster RabbitMQ avec 3 pods. Une règle d'anti-affinité est définie pour contraindre les pods à être déployés sur des nœuds distincts.

2.6. REGISTRE DOCKER

Botalista utilise un registre Docker privé pour les images de ses microservices. Ce dernier est techniquement un point de défaillance unique mais uniquement dans une situation peu probable :

— Le registre, ou le serveur qui en est responsable, doit être inaccessible ;
— Kubernetes doit créer un nouveau pod pour une raison quelconque (création manuelle, suite à une erreur, …) ;
— Le pod est déployé sur un serveur qui ne possède pas l'image du ou des conteneurs à créer.

Dans cette situation le pod ne peut pas être créé, ce qui impact le bon fonctionnement du cluster.

Afin de limiter les risques, chaque serveur de load balancing possède son propre registre Docker qui est accessible via l'adresse IP virtuelle que gère Keepalived.

2.7. RÉPLICATION DES MODULES

Les différents modules *métiers* et *logiciels* sont généralement déployés avec un pod unique. Une défaillance du pod provoque donc une indisponibilité du module associé. Cela représente un impact particulièrement conséquent pour les modules *métiers*.

Une panne du module `cit-traduction` *empêche l'ensemble des utilisateurs de changer la langue de l'interface.*

Une solution à ce problème consiste à définir au moins 2 *replicas* pour les différents modules, tout en appliquant une règle d'anti-affinité qui oblige la création des *replicas* sur des nœuds distincts. Cette approche est envisageable pour les modules *métiers*, elle n'augmenterait que de 6 Go la consommation en mémoire du cluster. Concernant les modules *logiciels*, ces derniers

n'ont pas été développés pour être facilement extensibles et implique donc des coûts importants en ressources pour assurer leur disponibilité. Par conséquent, 3 approches sont possibles :

— Modifier les modules *logiciels* pour les rendre extensibles ;

— Investir en ressources matérielles afin de répliquer ces modules ;

— Ne rien faire et accepter des indisponibilités de services potentielles.

L'architecture proposée ne prévoit pas les ressources permettant de répliquer l'ensemble des instances de *Botalista.software*. Il reste néanmoins possible de seulement répliquer les modules jugés indispensables pour minimiser l'impact d'une panne ou maintenance d'un worker.

CHAPITRE 3 - ANALYSE DE LA ROBUSTESSE

Ce chapitre étudie la tolérance de l'architecture proposée en simulant le dysfonctionnement de ses composants.

La première partie de cette analyse se concentre sur les nœuds de load balancing pour valider le bon fonctionnement de Keepalived et HAProxy. La deuxième partie est dédiée au cluster Kubernetes, tant pour le plan de contrôle, que les workers, ou applications déployées. Afin de clore l'analyse, les niveaux de tolérances aux pannes des divers composants sont déterminés à l'aide de formules mathématiques.

3.1. ROBUSTESSE DES COMPOSANTS EXTERNES

Les serveurs `LB_MASTER` et `LB_BACKUP` sont les seuls composants externes au cluster Kubernetes. Cette section vérifie l'attribution de l'adresse IP virtuelle par Keepalived qui permet l'équilibrage de la charge avec HAProxy.

Une analyse de la charge est également effectuée pour valider la robustesse des différents proxy (HAProxy et Træfik) lors de requêtes simultanées.

a. Attribution de l'adresse IP virtuelle par Keepalived

Le statut du service HAProxy détermine le serveur qui reçoit la VIP de Keepalived. Pour valider le comportement de cette attribution, il suffit d'arrêter le service HAProxy et d'observer les journaux de Keepalived.

```
1 # systemctl stop haproxy
2 # systemctl status snap.keepalived.daemon
3 Quorum lost for tracked process haproxy
4 (haproxy-vip) Changing effective priority from 201 to 101
5 (haproxy-vip) Master received advert from 10.136.26.85 with higher
     priority 151, ours 101
6 (haproxy-vip) Entering BACKUP STATE
7 # ip addr | grep 10.136.26.70
8
```

LISTING 3.1 – Arrêt du service HAProxy sur LB_MASTER

```
1 # systemctl status snap.keepalived.daemon
2 (haproxy-vip) received lower priority (101) advert from 10.136.26.84 -
     discarding
3 (haproxy-vip) received lower priority (101) advert from 10.136.26.84 -
     discarding
4 (haproxy-vip) received lower priority (101) advert from 10.136.26.84 -
     discarding
```

```
5 (haproxy-vip) Entering MASTER STATE
6 # ip addr | grep 10.136.26.70
7     inet 10.136.26.70/32 scope global ens160
```

LISTING 3.2 – Attribution de l'adresse IP virtuelle pour LB_BACKUP

Le listing 3.1 effectue un arrêt du service HAProxy pour `LB_MASTER`, ce qui baisse sa priorité en dessous de celle du nœud `LB_BACKUP` qui se voit attribuer la VIP par Keepalived comme illustré par le listing 3.2.

Le tableau 3.1 résume l'attribution de l'adresse IP virtuelle, selon le statut du service HAProxy [17], dans tous les scénarios potentiels.

Hôte	HAProxy	Priorité	VIP attribuée	Cluster k8s
LB_MASTER	Actif	201	Oui	Opérationnel
LB_BACKUP	Actif	151	Non	Opérationnel
Hôte	**HAProxy**	**Priorité**	**VIP attribuée**	**Cluster k8s**
LB_MASTER	Actif	201	Oui	Opérationnel
LB_BACKUP	Indisponible	51	Non	Opérationnel
Hôte	**HAProxy**	**Priorité**	**VIP attribuée**	**Cluster k8s**
LB_MASTER	Indisponible	101	Non	Opérationnel
LB_BACKUP	Actif	151	Oui	Opérationnel
Hôte	**HAProxy**	**Priorité**	**VIP attribuée**	**Cluster k8s**
LB_MASTER	Indisponible	101	*Oui**	Défaillant
LB_BACKUP	Indisponible	51	Non	Défaillant

TABLEAU 3.1 – Attribution de l'adresse IP virtuelle selon le statut du service HAProxy. Source : Réalisé par Dario GENGA

La VIP est attribuée même si le service HAProxy est indisponible sur l'ensemble des serveurs. Cette situation empêche cependant tout accès au cluster Kubernetes depuis l'extérieur, ainsi qu'aux différents composants du plan de contrôle (`kubeapi-server`, `etcd`, ...). Des échecs de connexions vont survenir pour ces composants, ce qui impacte la santé globale du cluster Kubernetes.

b. Analyse de la charge

L'utilitaire Siege est utilisé pour simuler plusieurs connexions simultanées à une instance de *Botalista.software*.

Actuellement, les instances les plus importantes atteignent une dizaine d'utilisateurs concurrents. Bien que les proxy HAProxy et Træfik soient conçus pour supporter des charges bien

17. L'attribution est identique en cas d'un arrêt du service Keepalived.

plus importantes, j'ai tout de même décidé d'effectuer la simulation de charge 3.3 suivante pour valider un total de 200 utilisateurs concurrents.

```
$ siege -b -c 200 -r 10 https://g.bota-hepia.ch
** SIEGE 4.1.6
** Preparing 200 concurrent users for battle.
Transactions:                  20000 hits
Availability:                 100.00 %
Elapsed time:                 107.69 secs
Data transferred:            8595.65 MB
Response time:                  1.05 secs
Transaction rate:             185.72 trans/sec
Throughput:                    79.82 MB/sec
Concurrency:                  195.54
Successful transactions:       20000
Failed transactions:               0
Longest transaction:            6.00
Shortest transaction:           0.02
```

LISTING 3.3 – Simulation de 200 utilisateurs concurrent avec l'utilitaire Siege

La simulation est effectuée avec les paramètres suivants :

— '`-c 200`' : Effectue un siège avec 200 utilisateurs concurrents ;

— '`-r 10`' : Le siège est effectué 10 fois ;

— '`https://g.bota-hepia.ch`' : La cible des requêtes à effectuer, ici une instance de *Botalista.software*.

Le résultat de cette simulation nous indique qu'un total de 20000 transactions [18] ont été effectuées, avec un taux de succès à 100% sur une durée de 107 secondes. Cette simulation permet de valider les connexions simultanées à l'architecture.

3.2. ROBUSTESSE DU CLUSTER KUBERNETES

Cette section analyse la robustesse du cluster durant la panne ou la maintenance d'un nœud. L'arrêt du service `kubelet` est effectué pour simuler une panne d'un worker. Pour le plan de contrôle, il est nécessaire d'arrêter directement le serveur.

Par défaut, Kubernetes prend 1 minute pour détecter un nœud indisponible. Il est marqué comme `NodeNotReady`, ce qui empêche le déploiement de pods sur ce dernier. Si après 5 minutes il n'est toujours pas rétabli, et qu'il s'agit d'un worker, alors Kubernetes commencera à évincer les pods qui s'y trouvent pour les déployer sur d'autres nœuds.

18. Le nombre d'accès à un serveur effectué. Cette valeur est supérieure à celle planifiée, car elle prend en compte les éventuels scripts et fichiers de style à charger.

a. Panne du plan de contrôle

Comme mentionné précédemment dans la section 4.2.a, une indisponibilité des serveurs de répartition de charge provoque une panne complète des composants du plan de contrôle. En dehors de ce cas de figure, 3 situations peuvent survenir concernant les nœuds du plan de contrôle :

Panne d'un seul nœud : Les composants du plan de contrôle et les applications déployées continuent de fonctionner correctement.

Panne de deux nœuds : Le plan de contrôle reste opérationnel, mais est désormais dans un état pouvant provoquer une perte de données si le nœud restant devient indisponible avant la remise en marche des deux autres. Les applications déjà déployées ne sont pas affectées.

Panne des trois nœuds : Le plan de contrôle est désormais défaillant. L'interface web des instances de *Botalista.software* reste accessible, néanmoins les communications avec les micro-services échouerons.

b. Panne d'un worker

L'impact d'une panne d'un worker varie selon les applications déployées sur ce dernier et selon les ressources disponibles sur les autres nœuds.

Les nœuds `WK_DEV_01` et `WK_DEMO_01` sont conçus pour avoir un taux d'activité réduit. Une panne affectant ces derniers n'affecte, par conséquent, pas la stabilité du cluster. La tolérance en cas d'une panne concernant les autres workers est mesurée dans la section 3.3.c.

La défaillance d'un worker nécessite au moins 5 minutes avant le redéploiement des pods présents. Une application non-répliquée est donc indisponible pendant cette durée.

3.3. TOLÉRANCES DE L'ARCHITECTURE

La tolérance représente le nombre de nœuds pouvant être défaillant. Cette section résume la tolérance autorisée, pour chaque composant de l'architecture, en tant que formules mathématiques.

a. Tolérance des serveurs de répartition de la charge

Il est nécessaire d'avoir au minimum 1 nœud pour gérer la répartition de la charge. Pour n étant le nombre de nœuds assumant ce rôle, la tolérance T_{lb} est représentée par l'équation 3.1.

$$T_{lb} = n - 1 \tag{3.1}$$

L'infrastructure proposée tolère jusqu'à 1 nœud en panne.

$$T_{lb} = n - 1 = 2 - 1 = 1$$

b. Tolérance du plan de contrôle

Il est nécessaire d'avoir au minimum 1 nœud pour gérer le plan de contrôle. Il en faut cependant 2 pour maintenir le quorum `etcd`. Pour n étant le nombre de nœuds qui régulent le plan de contrôle, la tolérance du quorum T_q correspond à l'équation 3.2, tandis que la tolérance du plan de contrôle T_{cp} correspond à l'équation 3.3.

$$T_q = n - (\lfloor n/2 \rfloor + 1) \tag{3.2}$$

$$T_{cp} = n - 1 \tag{3.3}$$

Pour l'infrastructure proposée, la tolérance au quorum est de 1 tandis que celle du plan de contrôle est de 2.

$$T_q = n - (\lfloor n/2 \rfloor + 1) = 3 - (\lfloor 3/2 \rfloor + 1) = 3 - 2 = 1$$

$$T_{cp} = n - 1 = 3 - 1 = 2$$

c. Tolérance des workers

L'infrastructure comporte au minimum 5 nœuds : 3 pour les bases de données et 2 pour des instances spécifiques. Les nœuds qui gèrent ces instances sont configurés pour avoir un taux d'activité réduit, il n'est donc pas nécessaire d'assurer la disponibilité de ces instances.

Les bases de données tolèrent jusqu'à 2 nœuds en panne, néanmoins la panne d'un worker

peut représenter jusqu'à 32 Go RAM d'applications. Si cette valeur n'est pas disponible sur les autres nœuds du cluster, alors certaines applications ne pourront pas être redéployées.

Pour m la RAM totale disponible sur les workers (autres que `WK_DEV_01` et `WK_DEMO_01`), la tolérance des workers T_w est représentée par l'équation 3.4.

$$T_w = \begin{cases} 4 & \text{si } m >= 64 \\ 3 & \text{si } m >= 32 \\ 2 & \text{sinon} \end{cases} \tag{3.4}$$

d. Tolérance de l'architecture

La tolérance de l'architecture T_a est simplement la somme de toutes les tolérances (hors celle du quorum).

$$T_a = T_{lb} + T_{cp} + T_w \tag{3.5}$$

L'équation 3.6 permet de prouver que l'architecture proposée est capable de tolérer jusqu'à 7 nœuds en panne, tant que la tolérance de chaque composant est respectée.

$$T_a = T_{lb} + T_{cp} + T_w = 1 + 2 + 4 = 7 \tag{3.6}$$

e. Tableau récapitulatif

Le tableau 3.2 représente la tolérance de l'ensemble des composants de l'architecture.

Composants	**Formule**	**Tolérance**
Serveurs de répartition de la charge	$T_{lb} = n - 1$	1
Quorum du plan de contrôle	$T_q = n - (\lfloor n/2 \rfloor + 1)$	1
Composants du plan de contrôle	$T_{cp} = n - 1$	2
Workers	$T_w = \begin{cases} 4 & \text{si } m >= 64 \\ 3 & \text{si } m >= 32 \\ 2 & \text{sinon} \end{cases}$	4
Architecture	$T_a = T_{lb} + T_{cp} + T_w$	*7*

TABLEAU 3.2 – Tableau récapitulatif de la tolérance de l'architecture. Source : Réalisé par Dario GENGA

CHAPITRE 4 - ÉTUDE COMPARATIVE

L'association Botalista a exprimé son ouverture à un éventuel changement de fournisseur cloud, de préférence en Suisse, selon la robustesse, les frais et l'emprunte écologique associés à la nouvelle architecture. Ainsi, ce chapitre représente l'étude comparative effectuée au cours de ce projet dans le but de sélectionner le prestataire de service le plus pertinent.

Les fournisseurs cloud sélectionnés pour cette étude sont :

— OVHcloud : le fournisseur actuel utilisé par Botalista, basé en France ;

— Infomaniak : l'un des plus grand fournisseur cloud en Suisse ;

— Exoscale : un prestataire cloud Suisse ayant déjà collaboré avec HEPIA.

La première partie de cette étude présente les frais actuels de Botalista et définit les ressources matérielles requises par l'architecture. Les sections suivantes analysent les aspects financiers et environnementaux de chaque fournisseur, aboutissant à une comparaison globale pour déterminer le prestataire le plus adapté.

Les tarifs des différents produits de cette étude ont été recueillis le 19 août 2024. Ils sont disponibles publiquement sur le site internet respectif des fournisseurs :

— OVHcloud : https://www.ovhcloud.com/fr/public-cloud/prices/

— Infomaniak : https://www.infomaniak.com/fr/hebergement/public-cloud/tarifs

— Exoscale : https://www.exoscale.com/pricing/

L'ensemble des prix de cette étude sont en euro (€), sans taxes appliquées.

4.1. SITUATION ACTUELLE

L'architecture actuelle de Botalista est composée de 6 serveurs physiques appartenant au fournisseur OVHcloud. Les charges annuelles de cette architecture sont représentées par le tableau 4.1, s'élevant à un total de 11 482€ par an. Les frais liés aux disques de stockages sont directement inclus à ceux des serveurs.

Produit facturé	**Prix annuel (€)**
Noms de domaine + emails	150
jiangxia	1 187
africa	2 054
antilia + *europa* + *mediaterra*	6 591
pandora	1 500
Total	**11 482**

TABLEAU 4.1 – Tableau des frais annuels pour l'architecture actuelle de Botalista. Source : Tableau réalisé par Dario GENGA, tarifs communiqués par M. Cyril BOILLAT

4.2. RESSOURCES REQUISES

L'étude comparative est effectuée selon les ressources suivantes pour chaque fournisseur :

— 3 VMs avec 8 Go RAM et 20 Go en espace disque
— 2 VMs avec 4 Go RAM et 20 Go en espace disque
— 12 VMs avec 32 Go RAM et 50 Go en espace disque, dont :
  * 10 VMs avec 100% en taux d'activité
  * 1 VM avec 50% en taux d'activité
  * 1 VM avec 40% en taux d'activité
— Un total de 4.1 To en *block storage* :
  * 2 To pour les données spatiales d'Open-Elevation
  * 3x200 Go pour les bases de données
  * 1.5 To pour le registre Docker, Jenkins, Alfresco et Elasticsearch [19]
— 1 adresse IP publique

Les workers de l'architecture nécessitent un total de 384 Go RAM. Des VMs de 32 Go ont ainsi été sélectionnées car ces dernières ont généralement le meilleur rapport en prix/RAM/CPU. Elles sont également plus intéressantes pour compléter un besoin de ressources spontanées, lors par exemple du clonage d'une instance de *Botalista.software*.

Puisque *Botalista.software* n'est pas une solution qui requiert des temps de calculs conséquents, il a été décidé de ne pas imposer de ressources CPU minimales.

Chacun des fournisseurs sélectionnés propose des tarifs qui sont facturés selon le temps réel d'exécution. Le taux d'activité mentionné précédemment est mensuel, tel que :

— 100% correspond à 730 heures par mois ($\approx$ 30.42 jours) ;
— 50% correspond à 365 heures par mois (soit 720 minutes par jour) ;
— 40% correspond à 292 heures par mois (soit 576 minutes par jour).

4.3. OVHCLOUD

OVHcloud est un acteur mondial, et le leader européen du cloud, opérant plus de 450 000 serveurs dans 37 centres de données sur 4 continents, à destination de 1,6 million de clients dans plus de 140 pays. Ce fournisseur propose des serveurs dédiés, correspondant à l'infrastructure actuelle de Botalista, mais également un public cloud avec un service Kubernetes intégré.

Les sections suivantes présentent les engagements et résultats environnementaux de l'entreprise. Suite à ça, une étude des frais pour une architecture public cloud, avec et sans le service Kubernetes intégré, est effectuée.

19. Elasticsearch est un logiciel pour surveiller et stocker les métriques du cluster.

a. Impact environnemental

OVHcloud est transparente sur ses objectifs et performances environnementaux. Dans le cadre de ses engagements, elle communique sur son impact environnemental global, non seulement en matière d'émissions de gaz à effet de serre, mais aussi concernant sa gestion de l'énergie, de l'eau, des déchets, des transports et son empreinte carbone.

Cette section présente ses principaux indicateurs environnementaux en 2023. Ils sont conformes aux normes ISO 30134-2, ISO 30134-3, ISO 30134-8 et ISO 30134-9.

— 1.29 en **Power Usage Effectiveness (PUE)** : Il mesure l'efficience du datacenter. Plus le site est performant, plus le résultat tend vers 1 ;
— 0.30 L/kWh IT en **Water Usage Effectiveness (WUE)** : Il évalue l'utilisation de l'eau pour les opérations du datacenter. Plus le WUE est proche de 0, mieux c'est ;
— 0.18 kg CO2e / kWh IT en **Carbon Usage Effectiveness (CUE)** : Il quantifie l'intensité du carbone au sein du datacenter. Plus le CUE est proche de 0, mieux c'est ;
— 91% en **Renewable Energy Factor (REF)** : Il détermine le taux d'énergies renouvelables dans le datacenter ;
— 36% de **ratio de composants réutilisés** : Il indique que sur 100 composants utilisés, 36 sont des pièces reconditionnées.

OVHcloud possède également la certification ISO 50001 relative aux systèmes de management de l'énergie et participe au Code de conduite européen pour l'efficacité énergétique dans les centres de données.

b. Public cloud

Le prix horaire des différentes instances cloud, localisées en France à Graveline, est représenté par le tableau 4.2. On y constate que l'instance `r3-32`, optimisée pour la mémoire, possède un tarif plus avantageux que l'instance `b3-32` qui est pour un usage général.

Instance	CPU	RAM (Go)	Stockage (Go)	Prix horaire (€)
b3-8	2	8	50	0.0465
b3-16	4	16	100	0.0930
b3-32	8	32	200	0.1860
b3-64	16	64	400	0.3720
c3-4	2	4	50	0.0415
c3-8	4	8	100	0.0830
r3-16	2	16	50	0.0602
r3-32	4	32	100	0.1203
r3-64	8	64	200	0.2407

TABLEAU 4.2 – Tableaux comparatifs des instances OVHcloud. Source : Réalisé par Dario GENGA

Le tarif d'entrée pour le *block storage*, correspondant à un *volume classique*, s'élève à 0.042€/mois/Go.

Le tableau 4.3 représente la tarification mensuelle totale pour une architecture public cloud chez OVHcloud. Elle contient l'ensemble des machines virtuelles nécessaires à l'architecture, le *block storage* et une IP publique.

Produit	Quantité	Taux d'activité	Prix horaire (€)	Prix mensuel (€)
r3-32	10	100%	1.203	878.19
r3-32	1	50%	0.06015	43.91
r3-32	1	40%	0.04812	35.13
c3-4	2	100%	0.093	60.59
b3-8	3	100%	0.1395	101.84
Volume classique	4.1 To	100%	-	172.20
Floating IP IPv4	1	100%	0.0025	1.83
Total	-	-	-	**1 293.69**

TABLEAU 4.3 – Tarification de l'architecture public cloud chez OVHcloud. Source : Réalisé par Dario GENGA

Cette architecture s'élève à 1 293.69€ mensuels, soit 15 524.28€/an.

c. Managed Kubernetes Service

OVHcloud propose un service Kubernetes (MKS) directement intégré au public cloud, où les nœuds du plan de contrôle sont gérés par OVHcloud sans le moindre frais supplémentaires. Cette section étudie les frais d'architecture lorsque l'on dédie la gestion du plan de contrôle Kubernetes et les serveurs de load balancing au fournisseur.

Ainsi, les seules instances à gérer et provisionner sont les 12 workers du cluster Kubernetes.

L'accès aux instances de Botalista nécessitera l'ajout d'un load balancer appartenant à OVH, qui sera utilisé par le service Træfik [20].

Le tableau 4.4 représente la tarification mensuelle totale pour une architecture public cloud avec service Kubernetes chez OVHcloud.

Produit	**Quantité**	**Taux d'activité**	**Prix horaire (€)**	**Prix mensuel (€)**
`r3-32`	10	100%	1.203	878.19
`r3-32`	1	50%	0.06015	43.91
`r3-32`	1	40%	0.04812	35.13
Volume classique	4.1 To	100%	-	172.20
Load Balancer S	1	100%	0.0083	6.06
Total	-	-	-	**1 135.49**

TABLEAU 4.4 – Tarification de l'architecture avec service Kubernetes chez OVHcloud. Source : Réalisé par Dario GENGA

Cette architecture s'élève à 1 135.49€ mensuels, soit 13 625.88€/an.

4.4. INFOMANIAK

Infomaniak est l'un des plus grands acteurs cloud en Suisse, qui se présente comme un cloud éthique sans compromis sur l'écologie, la vie privée et l'humain.

Les sections suivantes présentent les engagements et résultats environnementaux de l'entreprise. Suite à ça, une étude des frais pour une architecture public cloud est effectuée.

a. Impact environnemental

Infomaniak possède de nombreux engagements écologiques, dont notamment la compensation de ses émissions de gaz à effet de serre à 200%. Pour cela, elle finance divers projets en Suisse et au Nicaragua permettant la réduction des émissions de gaz à effet de serre dans le monde.

Infomaniak est certifiée de deux normes ISO depuis avril 2015 : la ISO 14001 et la ISO 50001.

ISO 14001 Cette norme définit des critères de gestion pour mesurer et réduire l'impact de ses activités sur l'environnement.

ISO 50001 Cette norme définit un cadre d'exigences pour fixer des objectifs de performance énergétique mesurables afin d'analyser l'efficacité et l'efficience de sa politique énergétique.

20. Le type du service devra être modifié en `LoadBalancer` au lieu de `NodePort`.

Infomaniak est également certifiée par les Services Industriels Genevois (SIG), 100% de l'énergie utilisée par Infomaniak provient d'énergies renouvelables.

b. Publique cloud

Le prix horaire des différentes instances cloud est représenté par le tableau 4.5.

Instance	CPU	RAM (Go)	Stockage (Go)	Prix horaire (€)
a2_ram4_disk20_perf1	2	4	20	0.00939
a4_ram8_disk20_perf1	4	8	20	0.01664
a4_ram16_disk50_perf1	4	16	50	0.02232
a8_ram16_disk50_perf1	8	16	50	0.03115
a8_ram32_disk50_perf1	8	32	50	0.04572
a16_ram32_disk50_perf1	16	32	50	0.06336
a16_ram64_disk50_perf1	16	64	50	0.08608

TABLEAU 4.5 – Tableaux comparatifs des instances Infomaniak. Source : Réalisé par Dario GENGA

Le tableau 4.6 représente la tarification mensuelle totale pour une architecture public cloud chez Infomaniak.

Produit	Quantité	Taux d'activité	Prix horaire	Prix mensuel
a8_ram32_disk50_perf1	10	100%	0.45720	333.76
a8_ram32_disk50_perf1	1	50%	0.02561	18.70
a8_ram32_disk50_perf1	1	40%	0.02159	15.76
a2_ram4_disk20_perf1	2	100%	0.01878	13.71
a4_ram8_disk20_perf1	3	100%	0.04992	36.44
Block storage (Perf1)	4.1 To	100%	0.45100	329.23
IPv4 réservée	1	100%	0.00411	3.00
Total	-	-		**750.60**

TABLEAU 4.6 – Tarification de l'architecture public cloud chez Infomaniak. Source : Réalisé par Dario GENGA

Cette architecture s'élève à 750.60€ mensuels, soit 9 007.20€/an.

4.5. EXOSCALE

Exoscale est un fournisseur de services cloud basé en Suisse depuis 2011.

Les sections suivantes présentent les engagements et résultats environnementaux de l'entreprise. Suite à ça, une étude des frais pour une architecture public cloud est effectuée.

a. Impact environnemental

Exoscale adopte diverses mesures en faveur du développement durable.

Actuellement, 90% de son électricité utilisée provient d'énergies renouvelables. Exoscale s'est engagée à atteindre les 100% d'ici 2025.

Exoscale ne possède pas encore de certifications en faveur du développement durable. Elle est cependant en train de mettre en œuvre des mesures pour la norme ISO 50001 afin d'obtenir une certification en 2024.

b. Publique cloud

Le prix horaire des différentes instances cloud est représenté par le tableau 4.7. L'espace de stockage de ces instances n'est pas inclus dans le prix, il est facturé séparément à hauteur de 0.00014€/heure/Go.

Instance	**CPU**	**RAM (Go)**	**Stockage (Go)**	**Prix horaire (€)**
Medium	2	4	-	0.04666
Large	4	8	-	0.09333
Extra-large	4	16	-	0.18667
Huge	8	32	-	0.37334
Mega	12	64	-	0.74667
Extra-large (mem. op)	2	16	-	0.14000
Huge (mem. op.)	4	32	-	0.28000
Mega (mem. op.)	8	64	-	0.56000

TABLEAU 4.7 – Tableaux comparatifs des instances Exoscale. Source : Réalisé par Dario GENGA

Les instances à mémoires optimisées (mem. op.) possèdent un tarif plus avantageux que les instance à usages générales.

Le tableau 4.8 représente la tarification mensuelle totale pour une architecture public cloud chez Exoscale.

Cette architecture s'élève à 3 001.19€ mensuels, soit 36 014.28€/an.

Produit	Quantité	Taux d'activité	Prix horaire (€)	Prix mensuel (€)
Huge (mem. op.)	10	100%	2.80000	2044.00
Huge (mem. op.)	1	50%	0.14000	102.20
Huge (mem. op.)	1	40%	0.11200	81.76
Medium	2	100%	0.09332	68.13
Large	3	100%	0.27999	204.40
Local storage	700 Go	100%	0.09800	71.54
Block storage	4.1 To	100%	0.57400	419.02
Elastic IP	1	100%	0.01389	10.14
Total	-	-	4.1112	**3 001.19**

TABLEAU 4.8 – Tarification de l'architecture public cloud chez Exoscale. Source : Réalisé par Dario GENGA

4.6. RÉSUMÉ

Cette étude se conclut sur un résumé comparatif, concernant les impacts environnementaux et les frais d'architecture, entre chaque fournisseur cloud.

a. Comparaison des impacts environnementaux

L'ensemble des fournisseurs étudié adopte des mesures pour réduire leur empreinte écologique.

Exoscale est le plus jeune des acteurs cloud étudiés. Les mesures qu'il applique concernant le développement durable ne sont pas encore suffisamment complètes pour prétendre à une certificat ISO 50001. Sa transparence sur ses consommations énergétiques est également moins présente par rapport aux autres acteurs de cette étude. Néanmoins, Exoscale reste engagé dans la mise en place d'un cloud durable et continu de fournir des efforts pour atteindre cet objectif.

Infomaniak est un acteur cloud qui participe à de nombreux projets pour atténuer son empreinte écologique. Ses engagements écologiques font d'Infomaniak un cloud durable ayant reçu de nombreuses certifications et récompenses environnementales. Infomaniak sensibilise également son personnel et collaborateurs sur leur impact écologique.

OVHcloud est un pionnier du cloud durable depuis plus de 20 ans. Ses nombreux engagements environnementaux permettent à OVHcloud de réduire son empreinte carbone, ainsi que celle de ses clients. OVHcloud fait également preuve de transparence concernant ses engagements, objectifs et résultats environnementaux.

Infomaniak et OVHcloud sont adaptés à accueillir la nouvelle architecture de Botalista sur le plan environnementale. Leurs mesures sur le développement durable sont reconnues mondialement et assurent une empreinte écologique minimale. L'un des facteurs pouvant départager ces deux acteurs est la communication de leurs résultats. En effet, OVHcloud communique régulièrement ses rapports environnementaux et dispose d'une calculatrice carbone permettant de générer des rapports mensuels sur les émissions carbones de ses clients.

Suite à cette étude, j'ai évalué OVHcloud comme étant le prestataire le plus adapté, sur l'aspect environnementale, pour héberger la nouvelle architecture de Botalista.

b. Comparaison des frais d'architecture

Le tableau 4.9 représente la tarification mensuelle et annuelle des différents fournisseurs cloud étudiés.

	OVHcloud				
	Bare metal	**Public cloud**	**MKS**	**Infomaniak**	**Exoscale**
Prix mensuel (€)	956.83	1 293.69	1 135.49	750.60	3 001.19
Prix annuel (€)	11 482	15 524.28	13 625.88	9 007.20	36 014.28

TABLEAU 4.9 – Tableau comparatif des tarifications (hors TVA) annuelles et mensuelles entre OVHcloud, Infomaniak et Exoscale. Source : Réalisé par Dario GENGA

Cette étude démontre que le fournisseur cloud Infomaniak est le plus adapté, sur l'aspect financier, pour accueillir la nouvelle architecture de Botalista . Choisir ce fournisseur cloud permettrait de réduire les coûts d'architecture de 2 474.8€ par an.

c. Prestataire de service sélectionné

À la suite de cette étude comparative, sur les aspects financiers et écologiques, j'ai choisi Infomaniak pour héberger la nouvelle architecture de Botalista. L'empreinte carbonne d'Infomaniak est minimale tout en conservant des tarifs attractifs et avantageux par rapport au marché cloud.

L'infrastructure de ce prestataire repose sur l'écosystème OpenStack, ce qui permet l'extensibilité de l'architecture avec des outils comme Ansible ou Terraform.

CONCLUSION

Le but de ce travail était de proposer une nouvelle architecture en cluster, sans le moindre point de défaillance unique, afin d'assurer la disponibilité de la solution *Botalista.software*. Afin de concevoir cette architecture, je devais tout d'abord étudier les aspects matériels et logiciels de la solution existante, qui est en constante évolution. L'étude de ce travail est, par conséquent, menée d'une manière plus approfondie que celle réalisée au cours du projet de semestre. Suite à ça, j'ai pu conceptualiser et déployer la nouvelle architecture sur un environnement virtuel mis à disposition par l'HEPIA. Ce dernier m'a permis de valider la robustesse de l'architecture en simulant des défaillances pour les divers composants. Finalement, une fois la robustesse de l'architecture validée, j'ai effectué une étude comparative entre différents fournisseurs cloud afin de déterminer le plus pertinent selon les aspects financiers et écologiques.

Ce travail de Bachelor était pour moi l'opportunité de développer mes connaissances et compétences concernant la mise en production d'une solution informatique. Cela a été une expérience très enrichissante durant laquelle j'ai pu conceptualiser une architecture à haute disponibilité pour un projet concret, tout en évaluant les aspects financiers et écologiques de la solution. J'ai également pris beaucoup de plaisir à découvrir et manipuler Kubernetes malgré la complexité de cette plateforme. Bien que chaque jour représentait son lot de défis à surmonter, les résultats de cette thèse me permettent d'affirmer avoir correctement tiré parti des fonctionnalités de Kubernetes.

Avant de conclure cette thèse, je vais présenter des améliorations envisageables au projet et à la solution *Botalista.software*.

La nouvelle architecture repose sur des machines virtuelles afin de la rendre extensible. Bien que je n'ai pas eu le temps d'approfondir cet aspect durant la réalisation de ce travail, l'architecture supporte l'ajout ou le retrait dynamique d'une machine virtuelle. Actuellement, les hausses en consommations RAM de Botalista peuvent survenir dans trois scénarios :

— Lorsqu'un membre de l'association fait une demande pour cloner son instance ;

— Lorsqu'un nouvel institut rejoint l'association, créant ainsi une nouvelle instance ;

— Chaque nuit lorsque les modules *logiciels* propagent leurs données sur la pile RabbitMQ.

Les deux premiers scénarios sont des événements prévisibles durant lesquelles on peut provisionner manuellement (depuis une interface web ou en ligne de commande) une nouvelle machine virtuelle. Le dernier scénario nécessite la mise en place d'un outil pour surveiller les ressources du cluster, comme par exemple Prometheus ou Elasticsearch. Il est ensuite possible

de provisionner automatiquement une machine virtuelle lorsque les ressources surveillées sont jugées insuffisantes. Le même principe peut également être appliqué pour retirer une instance du cluster si une quantité élevée des ressources disponibles n'est pas utilisée.

Pour finir, une amélioration envisageable pour Botalista, qu'elle a déjà commencé à implémenter, est la mise en place de Function as a Service (FaaS). Une FaaS est un service cloud qui permet d'exécuter du code lorsqu'un événement spécifique se produit. Les ressources requises à l'exécution du code ne sont pas consommées tant que l'événement associé au service n'est pas déclenché.

Des nombreux modules de Botalista sont en exécution permanente malgré des besoins occasionnels. Transformer ces modules en FaaS permettrait de réduire les ressources consommées, augmenter la robustesse de l'architecture et finalement réduire l'impact environnemental de Botalista.

Annexes

ANNEXE 1 - KUBERNETES

Cette annexe a pour but d'expliquer les terminologies et composants principaux d'un cluster Kubernetes. Ceci dans le but de pouvoir analyser correctement l'architecture de Botalista. Avant d'expliquer Kubernetes, il faut expliquer ce qu'est un conteneur :

Un conteneur est une forme de virtualisation qui permet d'isoler un ensemble de processus du système hôte et des autres conteneurs.

Ce principe permet de réunir les différents composants nécessaires (*p. e. dépendances*) d'une application pour pouvoir notamment l'installer ou la déployer facilement.

Kubernetes, aussi abbrégé en k8s, est une plateforme d'orchestration de conteneurs. Elle permet de gérer l'automatisation du déploiement, la mise à l'échelle et la gestion d'applications conteneurisées sur un ensemble de machines. Un déploiement fonctionnel de cette plateforme produit un *cluster Kubernetes*, composé d'un ensemble de machines ayant 2 rôles possibles :

— Au moins un nœud maître, qui gère l'orchestration du cluster ;

— Plusieurs nœuds, appelés worker, sur lesquels les containeurs sont déployés et exécutés.

Le reste de cet annexe est découpé en quatre sections principales. La première présente et explique les différents composants d'un cluster Kubernetes. La deuxième décrit comment les conteneurs sont gérés. La troisième présente le modèle réseau de Kubernetes et les différents services permettant d'exposer un accès externe aux applications. La quatrième parle brièvement des méthodes d'installation de Kubernetes ainsi que des différentes distributions existantes. Une cinquième section est également présente pour illustrer brièvement la création d'un objet Kubernetes.

1.1 COMPOSANTS D'UN CLUSTER

Le plan de contrôle, ou *control plane* en anglais, représente l'ensemble des composants qui gèrent l'orchestration d'un cluster Kubernetes. Il contient :

— Un serveur API (*kube-api-server*)

— Un ordonnanceur (*kube-scheduler*)

— Un gestionnaire de contrôle (*kube-controller-manager*)

— Une base de données en clé-valeur, par défaut *etcd*

— Optionnellement, un contrôleur Cloud (*cloud-control-manager*)

Ces composants sont initialements tous installés sur une même machine qui gère donc le plan de contrôle.

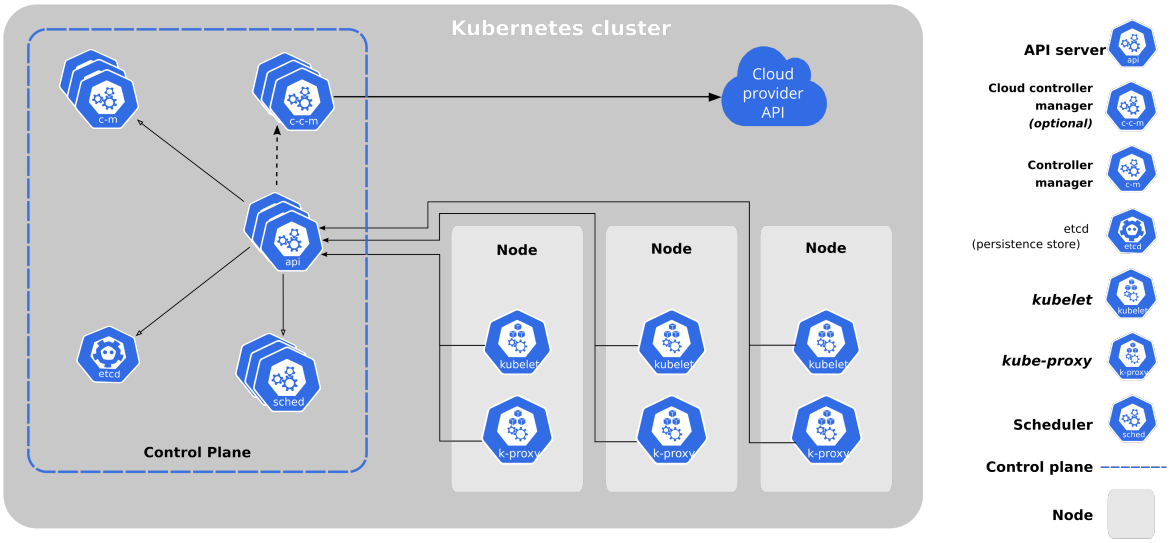

ILLUSTRATION 1.1 – Les composants d'un cluster Kubernetes. Source : tiré de *kubernetes.io*, ref. URL02

a. kube-apiserver

Le *kube-apiserver* est un composant du plan de contrôle qui expose l'API REST de Kubernetes.

Il s'agit du *front-end* pour le plan de contrôle, qui permet donc de gérer les requêtes internes du cluster entre les différents composants.

b. kube-scheduler

Le *kube-scheduler* est un composant du plan de contrôle qui gère l'ordonnancement des pods [21] à un nœud.

Ce composant surveille la création des pods au sein du cluster afin de les assigner à un nœud. Cette assignation se base sur divers critères tels que les contraintes de ressources ou les spécifications d'affinités [22] et d'anti-affinités.

c. kube-controller-manager

Le *kube-controller-manager* est un composant du plan de contrôle. Il représente un ensemble de controllers qui surveillent l'état de santé du cluster et effectuent des requêtes pour atteindre l'état souhaité.

21. Représente un ou plusieurs containeurs. Il s'agit de l'unité la plus petite que l'on peut créer et déployer.
22. L'affinité permet de définir des contraintes d'assignements selon des labels.

d. etcd

Kubernetes utilise par défaut *etcd* pour stocker les données du cluster. *etcd* est un magasin de données clés-valeurs qui permet de stocker la configuration de systèmes ou cluster distribués.

e. kubelet

Un processus présent sur chaque worker qui s'assure que les conteneurs sont exécutés dans un pod. Ce processus ne gère que les conteneurs créés par Kubernetes et il communique avec le plan de contrôle via le serveur API.

f. kube-proxy

Un proxy réseau présent sur chaque worker qui permet de gérer les règles réseaux d'un nœud. Il s'agit d'un composant indispensable pour assurer les communications réseaux à l'intérieur et extérieur du cluster.

g. Interface d'exécution du conteneur

L'interface d'exécution du conteneur, ou Container Runtime Interface (CRI) en anglais, est un composant indispensable pour gérer les conteneurs. Cette interface est responsable de la gestion des exécutions et cycles de vie des conteneurs au sein de Kubernetes.

Une CRI, telle que containerd ou cri-o, doit ainsi être installée sur chaque nœud du cluster.

1.2 Charges de travail

Le terme *charge de travail*, ou *workload* en anglais, représente une application exécutée avec Kubernetes.

Dans Kubernetes, une application est exécutée en tant que conteneur au sein d'un pod.

a. Pod

L'unité de calcul la plus petite que l'on peut créer et gérer dans Kubernetes est appelé un *pod*. Il s'agit d'un ensemble d'uns ou plusieurs conteneurs co-localisés et co-planifiés, s'exécutant dans un contexte commun sur la même machine.

Les pods ont un cycle de vie qui détermine l'état actuel du ou des conteneurs qui les composent.

— **Pending** : Le pod a été créé et attend d'être assigné à un nœud ;

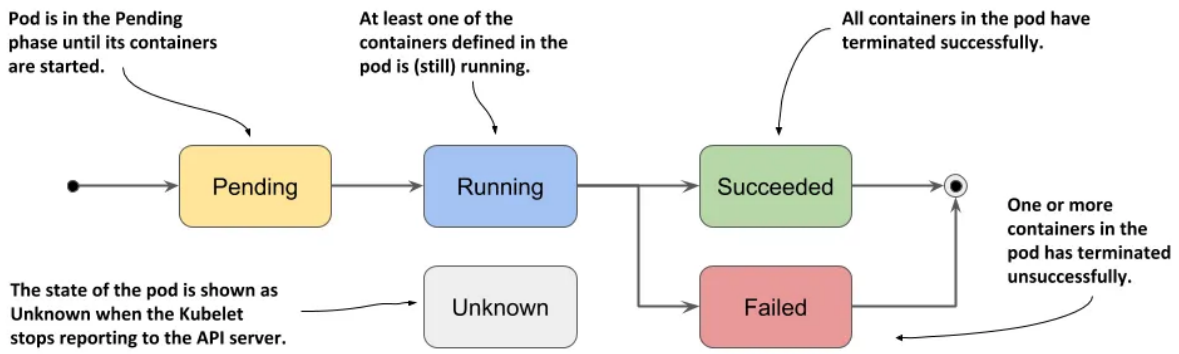

ILLUSTRATION 1.2 – Le cycle de vie d'un pod. Source : tiré de *medium.com*, ref. URL03

— **Running** : Le pod a été assigné et au moins l'un de ses conteneurs est démarré ;
— **Succeeded** : Tous les conteneurs se sont terminés sans erreurs ;
— **Failed** : Tous les conteneurs se sont terminés, mais au moins un conteneur a eu une erreur ;
— **Unknown** : L'état du pod n'a pas pu être récupéré. Ceci se produit généralement suite à une erreur de communication avec le nœud responsable du pod.

L'état de santé d'un pod est surveillé par un contrôleur qui va gérer l'action à entreprendre selon cet état, notamment pour les mécanismes d'auto-guérison.

Afin de faciliter la gestion des pods, ou plus précisément des conteneurs, l'API de Kubernetes permet la création d'objets de type workload.

b. Gestion des charges de travail

Les workloads sont une abstraction d'un niveau supérieur à un pod et sont gérés automatiquement par le plan de contrôle, selon les spécifications définies au sein de l'objet.

Il en existe de divers types :

— Un **Déploiement**, ou *deployment* en anglais, permet de définir l'état désiré pour un pod, notamment ses *ReplicaSets* [23] ;
— **StatefulSet** est similaire à un déploiement, mais garanti l'ordre [24] et l'unicité [25] des pods ;
— **DaemonSet** permet de s'assurer qu'un pod est présent sur chaque nœud du cluster ;
— Un **Job** est une tâche unique à réaliser ;
— Un **Cronjob** est identique à un job, mais est réalisé plusieurs fois selon un calendrier.

Botalista utilise des déploiements pour gérer ses différents modules.

23. Le nombre de répliques d'un pod qui doivent être existantes en même temps
24. Les répliques d'un pod ne sont créées que si celle précédente n'a pas rencontrée d'erreur durant sa création.
25. Définit une identité au pod qui permet d'assurer sa configuration, notamment avec des volumes persistants.

1.3 Le modèle réseau de Kubernetes

L'un des objectifs de Kubernetes d'un point de vue réseau est de permettre la communication entre un pod avec tous les autres pods sur n'importe quel autre nœud. Chaque nœud doit également être capable de communiquer avec les pods se trouvant sur ce dernier.

Pour ce faire, le Container Runtime Interface (CRI) utilisent des plugins d'interface réseau du conteneur qui configurent le modèle réseau du cluster et de ses pods.

Cette configuration correspond à :

— l'assignation d'une adresse IP au pod via un plugin IP Address Management (IPAM) ;

— une configuration des interfaces réseaux entre le nœud et le conteneur ;

— la création de routes de routage kernel sur chaque nœud du cluster ;

— la mise en place de règles *iptables* sur chaque nœud du cluster.

Cette configuration est supervisée par le kubelet selon le plugin de CNI défini.

a. Interface réseau du conteneur

L'interface réseau du conteneur, ou Container Network Interface (CNI) en anglais, est une librairie indispensable pour implémenter le modèle réseau de Kubernetes. Divers plugins, tels que Calico ou Flannel, utilisent la CNI pour créer et configurer les interfaces réseaux des conteneurs. Le kubelet d'un nœud est responsable de la mise en place ces interfaces au travers d'un plugin CNI.

Cette mise en place, illustrée par la figure 1.3, se déroule de la manière suivante :

1. Le pod est créé ;
2. Kubelet charge le plugin CNI et invoque la commande *ADD* ;
3. Le réseau du pod est configuré par le plugin CNI ;
4. Kubelet sauvegarde la configuration dans le pod ;
5. La CRI est notifiée que le réseau est prêt et qu'il est possible de démarrer le pod.

Une fois les interfaces réseaux créées et configurées, la mise en place d'un *service* est nécessaire pour exposer le réseau d'une application exécutée au sein d'un pod. Il existe trois types de services pour cela :

— ClusterIP

— NodePort

— LoadBalancer

Un *ingress* permet également d'exposer une application.

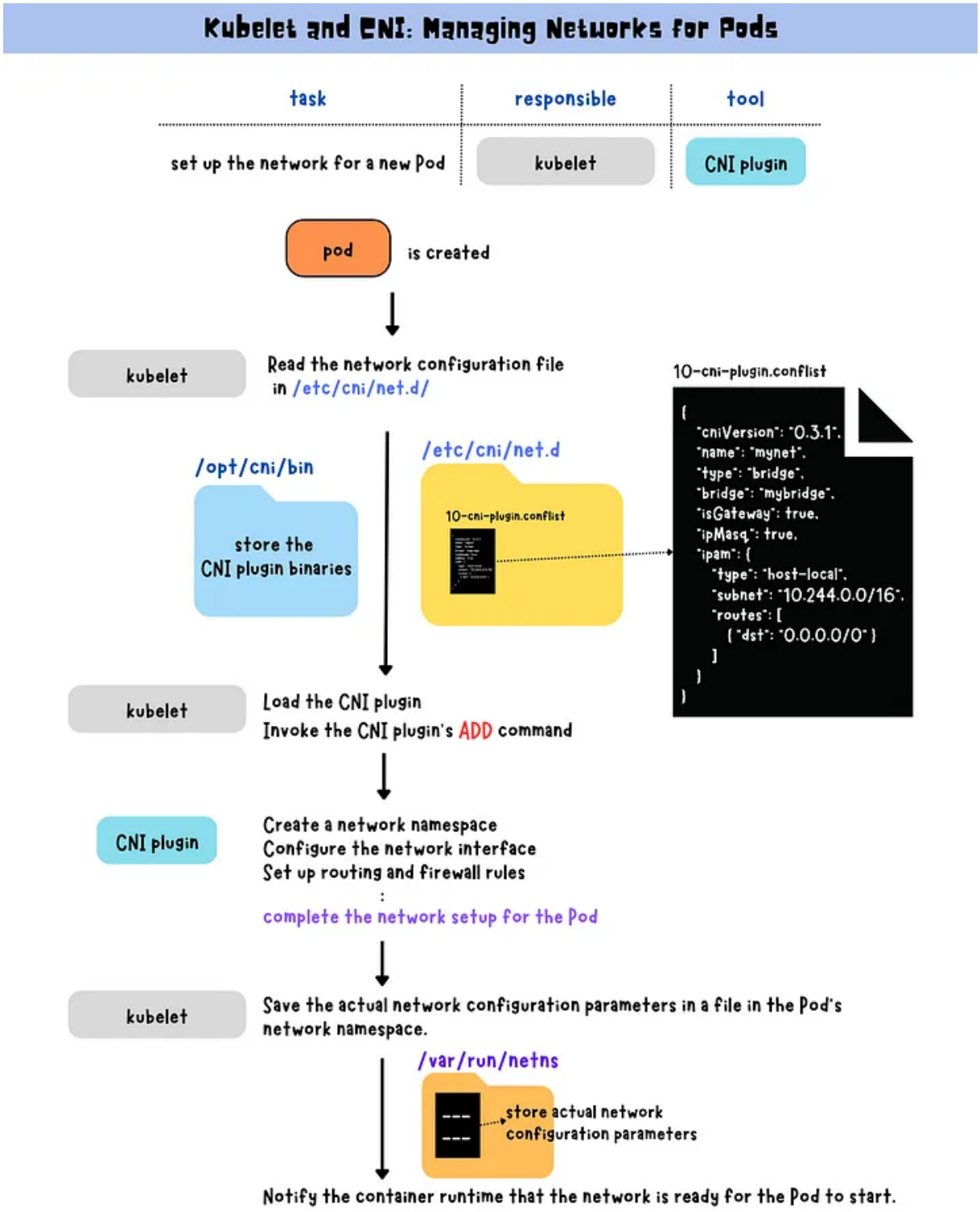

ILLUSTRATION 1.3 – La gestion du réseau d'un pod par le kubelet et CNI. Source : tiré de *medium.com*, ref. URL04

b. ClusterIP

Le service par défaut de Kubernetes est le ClusterIP. Il assigne simplement une adresse IP au pod. Cette IP n'est accessible que depuis le cluster, néanmoins il est possible d'exposer cette IP au travers d'un proxy afin de pouvoir y accéder depuis l'extérieur.

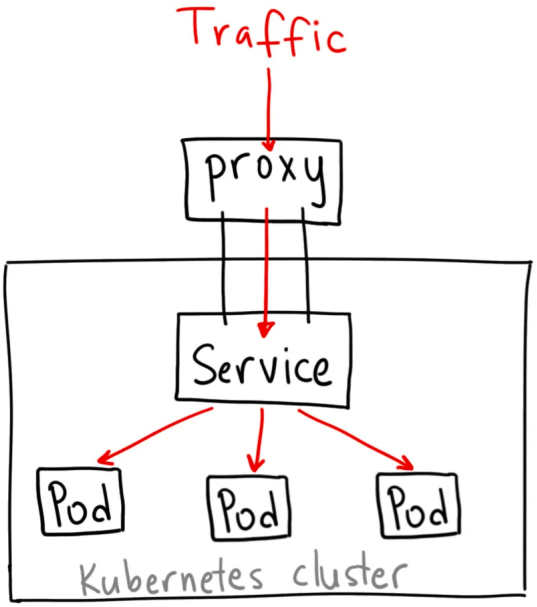

ILLUSTRATION 1.4 – Visualisation d'un service ClusterIP. Source : tiré de *tamerlan.dev*, ref. URL05

c. NodePort

Un service de type NodePort expose un port entre 30000 et 32767, sur l'intégralité des nœuds du cluster, afin d'accéder à un pod. Le service est aussi bien accessible depuis l'intérieur que l'extérieur du cluster.

d. LoadBalancer

Kubernetes peut créer des services de type LoadBalancer dans des environnements où le fournisseur cloud supporte des load balancers externes. L'implémentation de ce service est similaire à une création d'un NodePort, elle est cependant gérée par le cloud-controller-manager.

Ce type de service permet d'exposer une application en profitant des avantages d'un load balancer. Il n'est cependant pas possible d'en profiter dans des environnements où le cluster Kubernetes n'est géré par un fournisseur cloud.

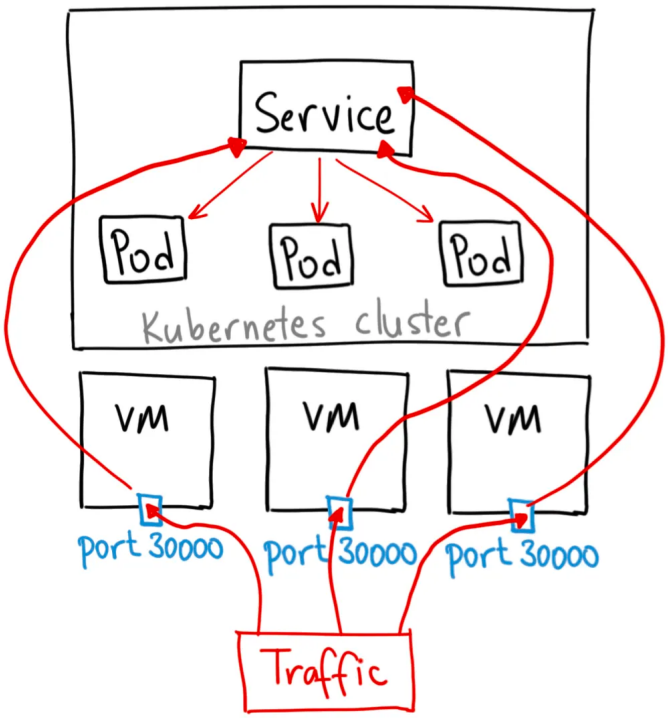

ILLUSTRATION 1.5 – Visualisation d'un service NodePort. Source : tiré de *tamerlan.dev*, ref. URL05

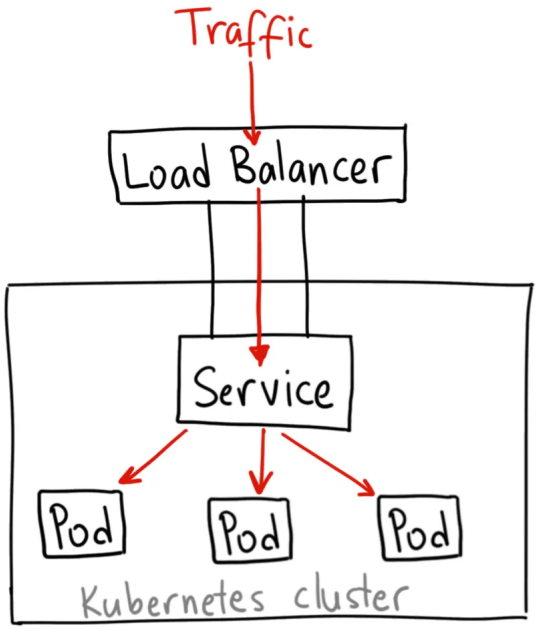

ILLUSTRATION 1.6 – Visualisation d'un service LoadBalancer. Source : tiré de *tamerlan.dev*, ref. URL05

e. Ingress

L'ingress n'est pas un service néanmoins, il s'agit d'un objet Kubernetes qui permet d'exposer des routes HTTP et HTTPS de l'extérieur du cluster vers des services à l'intérieur de ce dernier.

Un ingress agit comme un routeur intelligent, que l'on peut configurer pour avoir un fonctionnement similaire à un load balancer traditionnel.

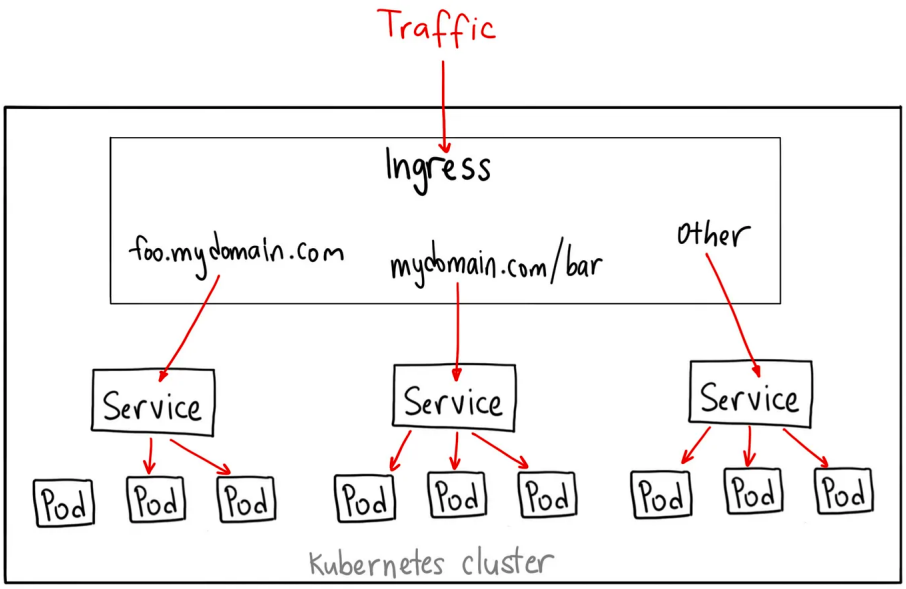

ILLUSTRATION 1.7 – Visualisation du trafic avec un ingress. Source : tiré de *tamerlan.dev*, ref. URL05

1.4 INSTALLATION

Il existe diverses méthodes pour installer et mettre en place un cluster Kubernetes, chacune adaptée à un contexte particulier. Cela concerne tout particulièrement :

— Les distributions Kubernetes ;

— Les outils d'installation ;

— Les différentes Container Runtime Interface ;

— Le plugin exploitant la Container Network Interface.

Cette section présente brièvement ces différentes méthodes et alternatives possibles.

a. Les distributions Kubernetes

Lorsque l'on parle de Kubernetes on fait référence à la version officielle, abrégée en k8s. Elle a cependant le défaut d'être difficile à mettre en place, ce qui impacte sa prise en main. D'autres

distributions ont ainsi vu le jour pour répondre à des besoins spécifiques et également remédier à ce soucis.

Le reste de cette section présente les distributions populaires sans pour autant les comparer en détail.

minikube L'une des premières distributions est minikube, qui permet la création d'un cluster local. Ceci est principalement conçu pour la prise en main de Kubernetes et pour du développement. Il s'agit de la distribution que j'ai utilisé au tout début de mon travail lors de mon apprentissage de Kubernetes.

microk8s, k3s, k0s Ces distributions de Kubernetes ont deux objectifs principaux : être légères et faciles à installer. Ce premier objectif permet la mise en place d'un cluster Kubernetes sur des machines avec ressources limitées, ce qui est très adapté pour de l'IoT. Le deuxième est conçu de sorte à pouvoir installer la distribution depuis un simple script bash, dans le but de pouvoir immédiatement prendre en main la distribution.

AKS, EKS, GKE Ces distributions sont gérées sur le cloud respectivement par Microsoft, Amazon et Google. Il s'agit de distributions commerciales qui proposent diverses intégrations à d'autres services.

De nombreuses autres distributions existent, ayant chacune leurs particularités et différences.

Botalista utilise la distribution officielle k8s, qui est également celle utilisée dans le cadre de ce travail. Malgré une installation plus complexe, cette distribution permet de bien comprendre les composants nécessaire au fonctionnement de Kubernetes.

b. Les outils d'installation

Les distributions telles que microk8s ou k0s permettent l'installation de Kubernetes avec un simple script d'installation. Dans le cas de k8s, il existe 3 outils différents : kubeadm, kOps et Kubespray.

- **kubeadm** permet une création simple et performante de clusters Kubernetes.
- **kOps** est intéressant si le déploiement du cluster Kubernetes se fait depuis un service cloud supporté (AWS et GCE).
- **Kubespray** [26] est recommandé en cas de familiarité avec l'outil d'automatisation Ansible ou pour mettre en place un cluster sur plusieurs plateformes différentes.

26. Kubespray est également une surcouche à kubeadm.

Dans le cadre de ce travail, kubeadm a été choisi pour sa simplicité, ce qui facilite la mise en place du cluster. Il s'agit également de l'outil utilisé par Botalista.

c. Container Runtime Interface

Chaque nœud doit posséder une CRI. Il en existe 3 officiellement supportées par Kubernetes :

— **containerd** : simple, robuste et portable. Également utilisé par Docker.

— **cri-o** : Une légère CRI optimisée pour Kubernetes. Utilisée par minikube.

— **Docker Engine** (avec cri-dockerd) : Bien que le Docker Engine n'implémente pas de CRI [27], il est possible d'installer le service cri-dockerd pour tout de même l'utiliser.

Botalista utilise Docker Engine pour son cluster, qui est encore supporté par la version 1.21.14 de Kubernetes actuellement utilisée. Dans le cadre de ce travail, containerd est néanmoins utilisé pour sa simplicité.

d. Container Network Interface

Chaque cluster doit mettre en place une CNI pour assurer la communication réseau entre les différents nœuds. La documentation officielle de Kubernetes liste une dizaine de plugin CNI possibles.

Dans le cadre de ce travail, Calico est utilisé car il s'agit du plugin qu'utilise actuellement Botalista.

27. Kubernetes ne supporte plus Docker Engine, sans cri-dockerd, depuis la version 1.24.

ANNEXE 2 - LISTE DES COMPOSANTS

Cette annexe présente brièvement les composants de la nouvelle architecture proposée, ainsi que les changements par rapport au déploiement actuel de Botalista.

2.1 TABLEAU DES VERSIONS

a. Composants principaux

L'intégralité de l'infrastructure utilise des machines virtuelles `Ubuntu 22.04.4 LTS`.

Nom	Version	Note
HAProxy	`3.0.2-1ppa1 jammy`	
Keepalived	`v2.3.1`	Installé avec Snap
Kubernetes	`v1.31.0`	
Containerd	`1.7.19`	

TABLEAU 2.1 – Tableau des composants principaux pour la nouvelle architecture. Source : Réalisé par Dario GENGA

b. Déploiements Kubernetes

Nom	Version	Note
Calico	`v3.28.1`	
Traefik	`v3.1.1`	
Kubegres	`1.18`	
Postgres	`16`	Déployé avec l'image `postgis:16-3.4`
Elastic Stack	`8.14.3`	
RabbitMQ	`3.13.6-management`	
Keycloak	`18.0.2`	Cette version n'est pas à jour

TABLEAU 2.2 – Tableau des déploiements Kubernetes. Source : Réalisé par Dario GENGA

c. Helm charts

Nom	Version	Chart	Repo
Longhorn	`v1.6.2`	longhorn-1.6.2	charts.longhorn.io
MongoDB	`0.10.0`	community-operator-0.10.0	mongodb.github.io/helm-charts
Redis	`7.2.5`	redis-19.5.5	charts.bitnami.com/bitnami
Prometheus	`v2.53.1`	prometheus-25.24.1	prometheus-community.github.io/helm-charts
Grafana	`11.1.0`	grafana-8.3.7	grafana.github.io/helm-charts

TABLEAU 2.3 – Tableau des *Helm charts* installés. Source : Réalisé par Dario GENGA

2.2 KUBERNETES

a. Version

Kubernetes est désormais à la version `v1.31.0` (précédemment `v1.21.14`).

b. CRI

Les nœuds du cluster utilisent désormais `containerd` en tant que CRI au lieu de Docker qui n'est plus supporté.

c. Initialisation

Le plan de contrôle est désormais composé de 3 nœuds. Cela signifie que la commande d'initialisation doit être adapté :

— L'IP du plan de contrôle doit correspondre à une adresse IP virtuelle qui permet d'accéder à chacun des nœuds maître

— l'argument `--upload-certs` doit être ajouté.

Voici la nouvelle commande d'initialisation :

```
sudo kubeadm init \
    --control-plane-endpoint "10.136.26.70:6443" \
    --upload-certs \
    --pod-network-cidr=192.168.0.0/16 \
    --cri-socket=unix:///run/containerd/containerd.sock
```

— La valeur du `control-plane-endpoint` est propre au réseau école

— Le dernier argument permet de spécifier le CRI utilisé, ici containerd.

2.3 CALICO

La version de Calico est désormais la `v3.28.1` (précédemment `v3.25.0`). Il est également mis en place avec son *operator*.

2.4 PROVISIONNEUR DE VOLUME

Longhorn est utilisé pour gérer les demandes de volumes au sein du cluster Kubernetes. Il correspond à la `StorageClass` par défaut et remplace le `nfs-client-provisioner`.

Longhorn est installé à l'aide de Helm.

— Version : `v1.6.2`

— Chart : `longhorn-1.6.2`

— Repo : `https://charts.longhorn.io`

2.5 TRAEFIK

Cette section présente les changements de Træfik.

a. Replicas

Træfik est désormais déployé avec 3 *replicas* (qui ne sont jamais sur le même noeud).

b. Version

La version de Træfik est désormais la `v3.1.1` (précédemment `v2.5.7`).

c. API

Les fichiers YAML pour créer les différents objets Træfik ont une nouvelle version API.

Nouvelle version API :

— `traefik.io/v1alpha1` (précédemment `traefik.containo.us/v1alpha1`)

d. TLS

N'ayant pas de nom de domaine publique pour ce projet, j'ai remplacé le *certResolver*

```
  tls:
    certResolver: letsencryptresolver
```

par le domaine (personnalisé) suivant

```
  tls:
    domains:
    - main: bota-hepia.ch
```

Ceci permet d'activer le TLS sans l'utilisation d'un *resolver* (qui provoquerait des erreurs dans ce contexte).

Dans certains contexte, tel que Kibana qui impose une connexion TLS, il est nécessaire de créer un objet `ServersTransport` avec l'option `insecureSkipVerify` à `true` afin d'autoriser les certificats auto-signés pour accéder à des services spécifiques.

e. Espaces de noms

Træfik peut être déployé dans son propre espace de nom, néanmoins il est important que chaque objets (IngressRoute, Middleware, ServersTransport, ...) qui sont interdépendant se trouvent dans le même espace de nom du service Kubernetes à atteindre.

Dans le cas contraire, une erreur dans le style suivant peut survenir :

```
ERR Failed to create middleware keys
error="middleware traefik/ipwhitelist is not in the IngressRoute namespace default"
ingress=bota-map-web-tls
namespace=default
providerName=kubernetescrd
```

f. Amélioration possible

Le middleware `IPWhiteList` est obsolète, il est recommandé de le remplacer par IPAllowList.

2.6 Bases de données

Les bases de données sont désormais toutes conteneurisées et en cluster.

a. PostgreSQL

Installé avec l'opérateur `Kubegres`.

— Operateur : Kubegres en version `1.18`

— Image des instances : `postgis:16-3.4`

b. MongoDB

Installé avec Helm.

— Version : `0.10.0`

— Chart : `community-operator-0.10.0`

— Repo : `https://mongodb.github.io/helm-charts`

Remarque : les clients utilisant MongoDB doivent modifier la configuration de leur connexion afin qu'elle puisse s'effectuer sur un Replica Set. Ceci est expliqué dans la documentation de MongoDB.

c. Redis

Installé avec Helm

— Version : `7.2.5`

— Chart : `redis-19.5.5`

— Repo : `https://charts.bitnami.com/bitnami`

Il est recommandé d'activer *Sentinel* pour mettre en place le mécanisme d'élection d'une nouvelle instance maître lorsque nécessaire. Néanmoins cette configuration n'expose qu'un seul service Kubernetes, ce qui empêche d'assurer une connexion à l'instance maître et peut donc provoquer des erreurs pour les requêtes en écriture.

Les requêtes en écritures nécessitent donc l'exécution de la requête suivante pour récupérer l'adresse de l'instance maître.

```
SENTINEL get-master-addr-by-name <name of your MasterSet. e.g: mymaster>
```

Ceci est expliqué dans le Bitnami package for Redis(R).

Si *Sentinel* n'est pas jugé pertinent, alors sa désactivation permettra d'exposer 2 services différents :

— *Redis Master service* qui redirige sur l'instance maître et autorise les requêtes en lecture/écriture ;

— *Redis Replicas service* qui redirige sur les instances de réplications et qui n'autorise que les requêtes en lectures seules.

2.7 NFS

L'architecture actuelle ne dépend désormais plus d'un serveur NFS. Sa mise en place est donc optionnel.

2.8 Surveillance de l'état de santé du cluster

a. Prometheus et Grafana

Ces deux outils sont des ajouts qui permettent de récupérer les métriques du cluster et visualiser l'état de santé de ce dernier.

Ils sont tous les deux installés à l'aide de Helm.

Prometheus :

— Version : `v2.53.1`

— Chart : `prometheus-25.24.1`

— Repo : `https://prometheus-community.github.io/helm-charts`

Grafana :

— Version : `11.1.0`

— Chart : `grafana-8.3.7`

— Repo : `https://grafana.github.io/helm-charts`

b. Elastic Stack

Les composants d'Elastic sont désormais à la version `8.14.3` (précédemment `7.14.0`).

2.9 RabbitMQ

RabbitMQ est désormais géré par le *RabbitMQ Cluster Kubernetes Operator* et déployé avec 3 *replicas* (qui ne sont jamais sur le même noeud).

L'image utilisée est la `rabbitmq:3.13.6-management`.

2.10 Keycloak

Keycloak est actuellement déployé en utilisant les mêmes fichiers manifestes de Botalista par soucis de temps manquant. La version utilisée (`18.0.2`) n'est cependant pas à jour, une mise à niveau est par conséquente recommandée.

2.11 Alfresco

Alfresco est composant de multiples composants, dont la majorité ne sont disponibles que via sa version *enterprise* (notamment pour les déploiements Kubernetes).

Par conséquent, Alfresco n'est pas déployé sur ce nouveau cluster Kubernetes.

ANNEXE 3 - INSTALLATION ET CONFIGURATION DE LONGHORN

Cette annexe décrit comment installer et configurer Longhorn, un système de stockage pour Kubernetes.

3.1 INSTALLATION

Longhorn est installé avec Helm.

```
# Add the Helm repo and fetch the latest charts
helm repo add longhorn https://charts.longhorn.io
helm repo update

# Install Longhorn
helm install longhorn longhorn/longhorn \
  --create-namespace -n longhorn-system \
  --set persistence.defaultClassReplicaCount=2 \
  --set defaultSettings.storageMinimalAvailablePercentage=15 \
  --set defaultSettings.storageReservedPercentageForDefaultDisk=15 \
  --set defaultSettings.nodeDrainPolicy="allow-if-replica-is-stopped"
```

Installation de Longhorn avec Helm

Attention, il est recommandé de protéger l'accès à l'UI de Longhorn pour protéger les volumes des applications. Si vous ne prévoyez pas d'utiliser l'UI, vous pouvez définir le nombre de replicas à 0.

La commande d'installation ci-dessus configure Longhorn pour :

— Définir le nombre de replica par défaut à 2 au lieu de 3 ;

— Autoriser l'allocation des volumes jusqu'à 85% de l'espace total des disques ;

— Autoriser le drainage d'un nœud si le dernier replica d'un volume est arrêté. Ceci facilite les opérations de maintenances des nœuds.

3.2 CONFIGURATION

Il n'est pas nécessaire d'utiliser le mécanisme de réplication de volumes de Longhorn pour les composants qui gèrent déjà la réplication des données, tels que les bases de données en cluster.

Par conséquent, une nouvelle StorageClass doit être créée afin de n'avoir qu'un seul replica se trouvant localement sur le même nœud du pod.

```
cat <<EOF > sc_longhorn-local.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
```

```
  name: longhorn-local
  namespace: longhorn-system
provisioner: driver.longhorn.io
allowVolumeExpansion: true
reclaimPolicy: "Delete"
volumeBindingMode: Immediate
parameters:
  numberOfReplicas: "1"
  staleReplicaTimeout: "30"
  fromBackup: ""
  fsType: "ext4"
  dataLocality: "strict-local"
  unmapMarkSnapChainRemoved: "ignored"
EOF

kubectl apply -f sc_longhorn-local.yaml
```

Création d'une `StorageClass` personnalisée pour Longhorn

ANNEXE 4 - INSTALLATION ET CONFIGURATION DES BASES DE DONNÉES

Cette annexe décrit comment installer les bases de données PostgreSQL, MongoDB et Redis au sein d'un cluster Kubernetes. Plusieurs instances de ces bases sont déployées afin d'assurer la haute disponibilité des données.

4.1 POSTGRESQL

La mise en place des bases de données PostgreSQL est effectuée à l'aide de `kubegres`, un opérateur Kubernetes qui met en place automatiquement des instances de réplications.

a. Installation

Installer `kubegres` via son fichier manifest.

```
kubectl apply -f https://raw.githubusercontent.com/reactive-tech/kubegres/v1.18/kubegres.yaml
```

Installation de l'opérateur kubegres

Créer un espace de nom `postgres` ainsi qu'un secret qui contient le mot de passe du *super user* et du *replication user*.

```
kubectl create ns postgres
cat <<EOF > secret-postgres.yaml
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
  namespace: postgres
type: Opaque
stringData:
  superUserPassword: superUserPassword
  replicationUserPassword: replicationUserPassword
EOF
kubectl create -f secret-postgres.yaml
```

Création du secret postgres

Il est possible de définir n'importe quel nom pour les clefs superUserPassword et replicationUserPassword

Si souhaité, il est possible de définir un script bash `primary_init_script.sh` qui sera exécuté la première fois qu'une instance primaire est créée. Ceci permet de créer en avance les bases de données et utilisateurs.

Créer un fichier `postgres-conf.yaml` avec le contenu suivant :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mypostgres-conf
  namespace: postgres
data:
  primary_init_script.sh: |
    #!/bin/bash
    set -e
    dt=$(date '+%d/%m/%Y %H:%M:%S');
    echo "$dt - Running init script the 1st time Primary PostgreSql
   container is created...";
    POSTGRES_USER="postgres"
    export PGPASSWORD=$POSTGRES_PASSWORD
    echo "$dt - Running: psql -v ON_ERROR_STOP=1 --username $POSTGRES_USER
   ";
    psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL
    CREATE USER backup_user;
    CREATE DATABASE cit_geocoding_db;
    CREATE DATABASE cit_per_db;
    EOSQL
    echo "$dt - Init script is completed";
```

Script d'initialisation pour l'instance PostgreSQL principale

Appliquer cette ConfigMap avec `kubectl apply -f postgres-conf.yaml`.

Les bases de données Postgres de Botalista nécessitent l'extension Postgis. Il faut donc créer un objet `Kubegres` à partir d'une image `postgis`, qui utilise `longhorn-local` comme StorageClass et `mypostgres-conf` comme ConfigMap.

```
nano postgres.yaml
```

```
apiVersion: kubegres.reactive-tech.io/v1
kind: Kubegres
metadata:
  name: postgres
  namespace: postgres
spec:
  replicas: 3
  image: postgis/postgis:16-3.4
  database:
    size: 25Gi
    storageClassName: longhorn-local
  customConfig: mypostgres-conf
  scheduler:
    affinity:
      nodeAffinity:
```

```
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: node-role.kubernetes.io/db-host
                  operator: Exists
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: app
                  operator: In
                  values:
                    - postgres
            topologyKey: kubernetes.io/hostname
  env:
    - name: POSTGRES_PASSWORD
      valueFrom:
        secretKeyRef:
          name: postgres-secret
          key: superUserPassword
    - name: POSTGRES_REPLICATION_PASSWORD
      valueFrom:
        secretKeyRef:
          name: postgres-secret
          key: replicationUserPassword
```

Objet Kubegres au format YAML

```
kubectl apply -f postgres.yaml
```

La création de l'objet `Kubegres` va créer 3 instances de PostgreSQL, avec un service pour accéder à l'instance principale et un autre pour celles de réplication.

4.2 MongoDB

MongoDB est installé avec Helm.

a. Installation de l'opérateur

```
helm repo add mongodb https://mongodb.github.io/helm-charts
helm repo update
helm install community-operator mongodb/community-operator \
 -n mongodb --create-namespace
kubectl config set-context --current --namespace=mongodb
```

Installation de l'opérateur MongoDB

b. Création du Replica Set

Une fois l'opérateur installé, créer le fichier `mongodbcommunity_cr.yaml` suivant

```
kind: MongoDBCommunity
apiVersion: mongodbcommunity.mongodb.com/v1
metadata:
  name: mongodb
  namespace: mongodb
spec:
  members: 3
  type: ReplicaSet
  version: "6.0.5"
  security:
    authentication:
      # SCRAM-SHA-1 seems required, but should but avoided
      modes: ["SCRAM", "SCRAM-SHA-1"]
  users:
    # Admin user
    - name: bota-admin
      db: admin
      passwordSecretRef:
        name: admin-password
      roles:
        - name: clusterAdmin
          db: admin
        - name: userAdminAnyDatabase
          db: admin
      scramCredentialsSecretName: admin-scram
    # Botalista users
    - name: g_bota_exp_user
      db: g_bota_exp_db
      passwordSecretRef:
        name: g-password
      roles:
        - name: readWrite
          db: g_bota_exp_db
      scramCredentialsSecretName: g-scram
  additionalMongodConfig:
    storage.wiredTiger.engineConfig.journalCompressor: zlib
  statefulSet:
    spec:
      template:
        spec:
          affinity:
            nodeAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
```

```
                nodeSelectorTerms:
                  - matchExpressions:
                      - key: node-role.kubernetes.io/db-host
                        operator: Exists
            podAntiAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
                - labelSelector:
                    matchExpressions:
                      - key: app
                        operator: In
                        values:
                          - mongodb-svc
                  topologyKey: kubernetes.io/hostname
      volumeClaimTemplates:
        - metadata:
            name: data-volume
          spec:
            accessModes: ["ReadWriteOnce"]
            storageClassName: longhorn-local
            resources:
              requests:
                storage: 2Gi # You may need to increase the storage size
        - metadata:
            name: logs-volume
          spec:
            accessModes: ["ReadWriteOnce"]
            storageClassName: longhorn-local
            resources:
              requests:
                storage: 1Gi # You may need to increase the storage size
```

Objet MongoDBCommunity au format YAML

Ce fichier permet de créer un *Replica Set* MongoDB avec 3 membres, tout en y définissant des utilisateurs à créer à partir d'un secret Kubernetes.

```
# This secret can be deleted after creating the mongodb cluster
apiVersion: v1
kind: Secret
metadata:
  name: admin-password
  namespace: mongodb
type: Opaque
stringData:
  password: superUserPassword
```

Exemple de secret Kubernetes pour l'administrateur MongoDB

Une fois les secrets créés, le *Replica Set* peut à son tour être créé :

```
kubectl apply -f mongodbcommunity_cr.yaml
```

Il est recommandé de supprimer les secrets Kubernetes une fois le *Replica Set* opérationnel.

La chaîne de connexion suivante permet d'accéder au *Replica Set* créé :

```
mongodb://mongodb-svc.mongodb:27017/?replicaSet=mongodb
```

Chaîne de connexion pour le MongoDB Replica Set

Se connecter via le service sans spécifier le *Replica Set* peut provoquer des erreurs pour les requêtes en écritures, car le service à lui seul ne permet pas de garantir une connexion à l'instance principale.

4.3 REDIS

Redis *Sentinel* est utilisé pour assurer la haute disponibilité de la base de données. L'installation est effectuée avec Helm.

a. Installation

Tout d'abord créer les fichiers `values.yaml` et `redis-password` pour configurer Redis.

```
cat <<EOF > values.yaml
global:
  storageClass: longhorn-local
auth:
  usePasswordFiles: true
  existingSecret: redis-password-secret
master:
  persistence:
    size: 1Gi
replica:
  replicaCount: 3
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/db-host
                operator: Exists
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: app.kubernetes.io/name
                operator: In
                values:
                  - redis
          topologyKey: kubernetes.io/hostname
sentinel:
```

```
  enabled: true
  masterSet: redis
  persistence:
    size: 100Mi
  resources:
    requests:
      cpu: 50m
      memory: 256Mi
    limits:
      cpu: 250m
      memory: 512Mi
EOF
```

Configuration Helm pour Redis avec Sentinel d'activé

Le fichier `redis-password` contient uniquement le mot de passe souhaité.

Créer ensuite un espace de nom `redis` qui contiendra le secret ainsi que les instances de Redis.

```
# Create the namespace
kubectl create ns redis
kubectl config set-context --current --namespace=redis
# Create the secret
kubectl create secret generic redis-password-secret \
 --from-file=redis-password
# Install Redis with Helm
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
helm install redis bitnami/redis -f values.yaml
```

Installation d'un cluster Redis avec Helm

b. Connexion à Redis

Seulement 1 service Kubernetes est créé lorsque Sentinel est activé. Il dispose de deux ports :

— Le port `6379` pour les requêtes en lecture

— Le port `26379` pour l'accès à Sentinel

Pour les commandes en écriture, il faut se connecter à Sentinel et récupérer le maître actuel avec la commande suivante :

```
# SENTINEL get-master-addr-by-name <name of your MasterSet>
SENTINEL get-master-addr-by-name redis
```

Commande pour récupérer le Redis maître

c. Réplication sans failover

La désactivation de Sentinel est envisageable si le mécanisme de failover avec Sentinel n'est pas souhaité ou nécessaire. Voici le fichier `values.yaml` modifié en conséquence :

```yaml
global:
  storageClass: longhorn-local
auth:
  usePasswordFiles: true
  existingSecret: redis-password-secret
master:
  persistence:
    size: 1Gi
  resources:
    requests:
      cpu: 50m
      memory: 256Mi
    limits:
      cpu: 250m
      memory: 512Mi
replica:
  replicaCount: 2 # This was changed from 3 to 2
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/db-host
                operator: Exists
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: app.kubernetes.io/name
                operator: In
                values:
                  - redis
          topologyKey: kubernetes.io/hostname
  persistence:
    size: 1Gi
  resources:
    requests:
      cpu: 50m
      memory: 256Mi
    limits:
      cpu: 250m
```

```
      memory: 512Mi
sentinel:
  enabled: false # This is now disabled
```

Configuration Helm pour Redis sans Sentinel

Cette configuration expose 2 services (port `6379`) au lieu d'un seul :

— `service/redis-master` : pour les requêtes en lecture/écriture

— `service/redis-replicas` : pour les requêtes en lecture

RÉFÉRENCES DOCUMENTAIRES

ANKIT, Sahay, 2022. *Redis Multi-node deployment : Replication vs. Cluster vs. Sentinels* [en ligne]. [visité le 2024-06-11]. Disp. à l'adr. : https://ankitsahay.medium.com/redis-multi-node-deployment-replication-vs-cluster-vs-sentinels-8099d15dcc09.

ANTHONY, Critelli, 2020. *Setting up a Linux cluster with Keepalived : Basic configuration* [en ligne]. [visité le 2024-05-30]. Disp. à l'adr. : https://www.redhat.com/sysadmin/keepalived-basics.

CHAD, Lavoie, 2018. *The Four Essential Sections of an HAProxy Configuration* [en ligne]. [visité le 2024-05-30]. Disp. à l'adr. : https://www.haproxy.com/blog/the-four-essential-sections-of-an-haproxy-configuration.

CISCO, [s. d.]. *What is High Availability ?* [en ligne]. [visité le 2024-06-25]. Disp. à l'adr. : https://www.cisco.com/c/en/us/solutions/hybrid-work/what-is-high-availability.html.

Connect to MongoDB, [s. d.] [en ligne]. [visité le 2024-08-03]. Disp. à l'adr. : https://www.mongodb.com/docs/drivers/java/sync/current/fundamentals/connection/connect/.

ERIKA, Heidi, 2021. *What is High Availability ?* [en ligne]. [visité le 2024-06-25]. Disp. à l'adr. : https://www.digitalocean.com/community/tutorials/what-is-high-availability.

etcd FAQ, 2024 [en ligne]. [visité le 2024-06-28]. Disp. à l'adr. : https://etcd.io/docs/v3.6/faq/.

Everything You Need to Know About Round Robin Load Balancing, [s. d.] [en ligne]. [visité le 2024-07-01]. Disp. à l'adr. : https://traefik.io/glossary/round-robin-load-balancing/.

High Availability Considerations, 2024 [en ligne]. [visité le 2024-05-30]. Disp. à l'adr. : https://github.com/kubernetes/kubeadm/blob/main/docs/ha-considerations.md.

High Availability, Load Balancing, and Replication, 2024 [en ligne]. [visité le 2024-06-11]. Disp. à l'adr. : https://www.postgresql.org/docs/current/high-availability.html.

Installing RabbitMQ Cluster Operator in a Kubernetes Cluster, 2024 [en ligne]. [visité le 2024-07-31]. Disp. à l'adr. : https://www.rabbitmq.com/kubernetes/operator/install-operator.

Keepalived, 2024 [en ligne]. [visité le 2024-05-30]. Disp. à l'adr. : https://wiki.archlinux.org/title/Keepalived.

keepalived.conf - configuration file for Keepalived, 2022 [en ligne]. [visité le 2024-07-02]. Disp. à l'adr. : https://manpages.debian.org/bookworm/keepalived/keepalived.conf.5.en.html.

Kubernetes Documentation, 2023 [en ligne]. [visité le 2024-01-11]. Disp. à l'adr. : https://kubernetes.io/docs/home/.

Le projet Botalista, 2024 [en ligne]. [visité le 2024-03-03]. Disp. à l'adr. : https://botalista.community/docs/Botalista_Presentation_Projet_FR.pdf.

Le projet BOTALISTA : un progiciel de gestion et une association, 2024 [en ligne]. [visité le 2024-03-15]. Disp. à l'adr. : https://botalista.community/.

MATT, Ghafouri, 2024. *Redis Deployment Types (Replication, Cluster, Sentinel)* [en ligne]. [visité le 2024-06-11]. Disp. à l'adr. : https://medium.com/codeboulevard/redis-deployment-types-replication-cluster-sentinel-fb23fa8ca523.

NICK, Ramirez, 2021. *HAProxy Configuration Basics : Load Balance Your Servers* [en ligne]. [visité le 2024-05-30]. Disp. à l'adr. : https://www.haproxy.com/blog/the-four-essential-sections-of-an-haproxy-configuration.

Replication, [s. d.] [en ligne]. [visité le 2024-06-11]. Disp. à l'adr. : https://www.mongodb.com/docs/manual/replication/.

Rook, 2024 [en ligne]. [visité le 2024-07-04]. Disp. à l'adr. : https://rook.io/docs/rook/latest-release/.

RUAN, Bekker, 2023. *Achieving High Availability with HAProxy and Keepalived : Building a Redundant Load Balancer* [en ligne]. [visité le 2024-05-30]. Disp. à l'adr. : https://sysadmins.co.za/achieving-high-availability-with-haproxy-and-keepalived-building-a-redundant-load-balancer/.

The Future of Storage™, [s. d.] [en ligne]. [visité le 2024-07-04]. Disp. à l'adr. : https://ceph.com/en/.

The Longhorn Documentation, 2024 [en ligne]. [visité le 2024-07-04]. Disp. à l'adr. : https://longhorn.io/docs/1.6.2/.

The Raft Consensus Algorithm, 2024 [en ligne]. [visité le 2024-06-28]. Disp. à l'adr. : https://raft.github.io/.

Traefik Documentation, 2024 [en ligne]. [visité le 2024-07-02]. Disp. à l'adr. : https://doc.traefik.io/traefik/.

Understanding Sentinels, 2024 [en ligne]. [visité le 2024-08-12]. Disp. à l'adr. : https://redis.io/learn/operate/redis-at-scale/high-availability/understanding-sentinels.