

---

# Design, implémentation, benchmark et études budgétaires d'une architecture pour l'ingestion de données IoT au sein de la plateforme Shayp

---

Travail de fin d'études présenté par  
Saïkou Ahmadou Barry

En vue de l'obtention du diplôme de  
Master en Sciences de l'Ingénieur Industriel orientation Informatique

*Promoteurs:* Grégoire de Hemptinne

*Superviseur:* Sébastien Combéfis

**Année Académique : 2019 - 2020**

# **CAHIER DES CHARGES RELATIF au TRAVAIL DE FIN D'ETUDES de**

Saïkou Ahmadou Barry inscrit en 5 MIN.....

Année académique : 5 MIN

Titre provisoire : Design, implémentation, benchmark et études budgétaires d'une architecture pour l'ingestion de données IoT au sein de la plateforme Shayp.

Objectifs à atteindre :

Intégrer au sein de la plateforme Shayp, le nouveau type d'appareils connectés en NBiot.  
Pour se faire proposer une comparaison budgétaire de deux architectures avec implémentation de ceux-ci et sélection d'une architecture.

Principales étapes :

- Design d'une architecture sur AWS et un autre cloud provider
- Implémentation des différentes architectures
- Analyses de performances
- Etudes budgétaire

Fait en trois exemplaires à Bruxelles, le 15 novembre 2019.

L'Etudiant

Nom-prénom :  
Barry Saïkou Ahmadou

Le Tuteur

Nom-prénom :  
Combéfis Sébastien

Le Promoteur

Nom-prénom :  
de Hemptinne Grégoire

Signature

Signature

Signature

Département/Unité  
Génie électrique.

Société  
Shayp

L'entreprise présentée dans ce rapport :



Shayp  
Rue des Pères Blancs 4  
1040 Brussels  
Tel : +32 475 45 72 38  
Email : info@shayp.com

*Remerciements à l'équipe de Shayp qui m'a accueilli dans leurs bureaux et à toute personne ayant contribué au bon déroulement de ce projet.*

# Remerciements

Je tiens à remercier toutes les personnes qui ont contribué de près ou de loin à la réalisation de ce travail.

Je pense tout d'abord à l'entreprise Shayp, qui m'a fait confiance et m'a donné les moyens de mener à bien ce projet.

J'adresse également mes plus vifs remerciements à mes promoteurs et superviseurs, Mr de Hemptinne et Mr Combéfis, qui par leurs conseils avisés m'ont aidé sans ménager leur temps et leurs efforts.

Je remercie aussi :

- Mr Josué Motte de Hackages pour ses conseils niveau infrastructure ;
- Mes différents relecteurs pour leurs conseils et corrections ;

Enfin, je ne manquerai pas de remercier sans cesse ma famille et mes proches de m'avoir soutenu et accompagné durant toute ma scolarité.

# Abstract

L'internet des objets joue un rôle de plus en plus fondamental dans un large éventail de secteurs, notamment l'industrie, l'agriculture, les soins de santé et d'autres services primordiaux dans notre quotidien. Il permet aux dispositifs embarqués de collecter et d'échanger des données d'une machine à l'autre. Dans de nombreux cas, le "cloud-computing" sert de paradigme évolutif et efficace pour la mise en œuvre d'applications back-end pouvant communiquer et interagir avec ces objets.

Cette thèse propose une architecture cloud pour une application IoT responsable de la collecte des données de consommations d'eau présentant plusieurs passerelles de communication différentes : Sigfox et NB-IoT. Deux solutions sont proposées et implémentées, une architecture basée sur le concept de "microservices" et une architecture basée sur le concept de "fonctions serverless". Une série de tests de performance ainsi qu'une étude budgétaire sont effectuées. Les résultats obtenus nous permettent de valider l'architecture microservices comme étant la plus optimale.

**Keywords**— IoT, Container, Microservices, Cloud, Serverless

*"If you want to accomplish something in the world, idealism is not enough -  
you need to choose a method that works to achieve the goal."*

**-Richard Stallman**

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Objectif . . . . .	9
1.2	Structure de la thèse . . . . .	10
<b>2</b>	<b>Background</b>	<b>11</b>
2.1	Virtualisation . . . . .	12
2.2	Isolation logicielle par conteneur . . . . .	12
2.3	Cloud computing . . . . .	14
2.3.1	Les cinq caractéristiques essentielles . . . . .	14
2.3.2	Les modèles de services . . . . .	14
2.3.3	Les modèles de déploiement . . . . .	16
<b>3</b>	<b>Approche architecturale</b>	<b>17</b>
3.1	Architecture monolithique . . . . .	18
3.2	Architecture microservices event-driven . . . . .	20
3.3	Architecture serverless . . . . .	23
3.4	Design conceptuel . . . . .	25
<b>4</b>	<b>Architecture Serverless - Microservice chez AWS</b>	<b>27</b>
4.1	Composants de l'architecture . . . . .	30
4.2	Fonctionnement . . . . .	33
4.3	Performances . . . . .	37
4.4	Bilan . . . . .	39
4.5	Estimation des coûts . . . . .	41
4.5.1	Variante 1 - Ingestion mixte (serverless et microservice) & data processing serverless . . . . .	41
4.5.2	Variante 2 - Ingestion microservice & data processing serverless . . . . .	42
4.6	Inconvénients et améliorations possibles . . . . .	45
4.6.1	Le coût peut être sensible à l'échelle . . . . .	45
4.6.2	Traitements par lots . . . . .	45
<b>5</b>	<b>Architecture microservices chez Scaleway</b>	<b>48</b>
5.1	Composants de l'architecture . . . . .	50
5.2	Fonctionnement . . . . .	55
5.3	Performances . . . . .	57
5.4	Bilan . . . . .	58
5.5	Estimation des coûts . . . . .	60
<b>6</b>	<b>Conclusion</b>	<b>61</b>
6.1	Retour sur les objectifs . . . . .	62
6.2	Réflexion . . . . .	62
6.3	Tests sur d'autres fournisseurs de cloud . . . . .	62
6.4	Serverless vs Microservices . . . . .	63

6.5 Conclusion . . . . .	63
<b>A Annexes</b>	<b>64</b>
A.1 Infrastructure as Code . . . . .	65
A.1.1 Les outils . . . . .	65
A.1.2 Utilisation au sein du projet . . . . .	66
A.2 Design complet architecture AWS . . . . .	69
A.3 Design complet architecture Scaleway . . . . .	71
A.4 Dashboards Portainer . . . . .	72
A.5 Dashboards Grafana . . . . .	76
<b>Acronyms</b>	<b>81</b>

# Introduction

---

1.1	Objectif . . . . .	9
1.2	Structure de la thèse . . . . .	10

---

Ce travail de thèse est réalisé sous l'égide de Shayp, une start-up basée à Bruxelles. La start-up a développé un compteur d'eau intelligent basé sur la technologie Sigfox et qui offre une surveillance en temps réel de la consommation d'eau et la détection de fuites d'eau. La solution s'appuie sur un écosystème composé d'un compteur intelligent pour surveiller la consommation, d'un algorithme de détection de fuites d'eau, d'une plateforme web et mobile offrant des informations sur la consommation d'eau et d'un système de messages SMS/e-mail en cas de fuite.

Un des principaux besoins de Shayp est d'améliorer son système existant d'ingestion de données de consommation d'eau afin de préparer l'arrivée de leur nouveau type de compteur utilisant le protocole de communication LPWAN NarrowBand IoT. À ce stade, Shayp souhaite améliorer son système existant en concevant une architecture appropriée qui permettra de réaliser un système évolutif d'ingestion de données, hétérogènes et hautement disponibles. Ainsi, le but de cette thèse est de proposer une architecture IoT pour l'ingestion des données massives. En alignement avec la croissance constante du parc d'objets connectés.

## 1.1 Objectif

La problématique à laquelle répond cette thèse se définit par la question suivante ; Comment concevoir une architecture cloud pour une application à lourde charge ?

De ce fait, ce travail vise à développer une solution qui s'inscrit dans le contexte de l'IoT et tient compte de l'existant. Différentes approches architecturales sont donc présentées, évaluées et comparées les unes aux autres. Différents objectifs et contraintes doivent être définis et pris en compte pour évaluer les solutions architecturales. Les parties prenantes ont des attentes différentes vis-à-vis des fonctionnalités de l'application. Un utilisateur s'attend, par exemple, à ce que l'application fonctionne de manière cohérente et avec une bonne performance. Un développeur souhaite une solution architecturale facile à développer et à maintenir tandis que l'entreprise et les investisseurs aspirent à une solution fonctionnelle, fiable et à coût réduit. De ce fait, les paramètres pris en compte pour la conception de l'architecture seront d'une part les attributs de qualité tel que la disponibilité, l'évolutivité, la maintenabilité, l'hétérogénéité et la flexibilité et d'autre part les contraintes d'optimisation de coût et de ressource.

- **Haute disponibilité de l'application :** La disponibilité est la proportion du temps d'un service dans un état de fonctionnement et d'accès. L'architecture devrait être hautement disponible afin de réduire au maximum le risque de perte de données des capteurs. Un fournisseur cloud peut fournir des garanties sur la disponibilité de ses services avec un accord de niveau de service (SLA, Service Level Agreement), où une disponibilité concrète est définie pour chaque service et la compensation si la disponibilité promise n'a pas été fournie. Une approche commune pour améliorer la disponibilité est de réduire le nombre de points uniques de défaillance dans une architecture.
- **Architecture évolutive :** L'évolutivité est la capacité d'un système à fournir la quantité correcte de ressources en fonction de la charge. Dans le cas de Shayp et son parc d'appareils connectés, l'architecture cloud devrait traiter la quantité de messages actuels ainsi que le double par mesure de prévention si la taille du parc venait à doubler, idéalement en respectant les contraintes d'optimisation de ressources et de coûts.
- **Prise en charge de l'hétérogénéité et de la flexibilité :** L'hétérogénéité est une propriété essentielle de tout réseau IoT. Notre solution devrait fonctionner avec plusieurs types d'appareils IoT déjà existants ou futurs de manière flexible de sorte que l'existant ne soit pas affecté.
- **Optimisation de coûts et de ressources :** Ces exigences non fonctionnelles vont être respectées par les attributs de qualité de maintenabilité et d'évolutivité car un bon niveau de maintenance garantit un faible coût de développement et de déploiement puisque ces derniers seront faits avec aisance en un temps réduit. En ce qui concerne l'évolutivité, cela permet de maintenir les coûts au niveau de la charge en d'autres termes au même niveau des revenus générés par la plateforme. Il est donc important d'avoir un coût réduit par objet connecté, ainsi qu'une marge positive par appareil connecté.

Dans le processus de conception de l'architecture, les décisions et les évaluations sont prises en tenant compte des éléments présentés ci-dessus.

## 1.2 Structure de la thèse

La structure de la thèse est la suivante : dans un premier temps, le chapitre 2 fournit les concepts-clés au lecteur pour lire la suite de cette thèse. Ensuite, le chapitre 3 présente plusieurs approches architecturalles, le design conceptuel de l'application finale et introduit les deux architectures implémentées basées sur le design conceptuel. Après cela, les deux architectures sont présentées et évaluées dans les chapitres 4 et 5. Suivi d'une conclusion concernant le choix final de l'architecture dans le contexte de Shayp et d'une manière générale dans le même domaine.

# Background

---

2.1	Virtualisation . . . . .	12
2.2	Isolation logicielle par conteneur . . . . .	12
2.3	Cloud computing . . . . .	14

---

Ce chapitre partage des informations de base sur les technologies et les concepts pertinents abordés dans cette thèse. Les sections suivantes présentent le concept de virtualisation, de conteneurs et d'outils d'orchestrations et le cloud computing.

## 2.1 Virtualisation

La virtualisation d'infrastructure permet aux utilisateurs d'exécuter plusieurs instances de machine virtualisées sur un même serveur. Les instances fonctionnent de manière totalement isolées de l'hôte et offrent également une évolutivité rapide, une meilleure utilisation et gestion de ressources de calcul fournies par ce même hôte et par conséquent une réduction des coûts matériels. La virtualisation la plus populaire est la virtualisation par hyperviseur.

### Virtualisation par hyperviseur

La virtualisation par hyperviseur est une méthode populaire pour déployer des machines virtuelles sur un hôte. Cette approche repose sur un logiciel appelé hyperviseur ou moniteur de machine virtuelle qui se trouve entre le matériel physique et des machines virtuelles. Il gère et distribue les ressources pour plusieurs machines virtuelles. Le principal avantage de la virtualisation par hyperviseur est qu'elle permet aux utilisateurs d'exécuter plusieurs machines virtuelles de manière isolée sur une seule machine physique. Cela permet aux développeurs de créer plusieurs environnements avec différents outils et systèmes d'exploitation sur un seul serveur physique.

## 2.2 Isolation logicielle par conteneur

Une alternative légère à l'utilisation d'un hyperviseur est l'isolation par conteneur. Les conteneurs fonctionnent au niveau du système d'exploitation, ils partagent donc efficacement le même noyau hôte du système d'exploitation. Les conteneurs représentent la virtualisation de l'environnement requis pour le bon fonctionnement d'une application, ce qui permet d'avoir un environnement virtuel léger en terme de ressources et contextuellement lié à l'application en ce qui concerne le système d'exploitation et librairies nécessaires. Par conséquent cette méthode est beaucoup plus efficace en terme de gestion de ressources quand il s'agit de déployer des applications découpées.

### Docker

Docker est un écosystème open source pour la construction et le déploiement de conteneurs. Les applications sous forme de conteneurs sur Docker peuvent être déployées plus rapidement que les applications traditionnelles non conteneurisées. De plus, Docker permet aux utilisateurs de configurer de nombreux composants, tels que la mémoire, le CPU et le réseau, par un fichier (`docker-compose.yml`) ou la ligne de commande. Docker fournit une plateforme permettant d'exécuter presque toutes les applications de manière isolée dans un conteneur. Cette isolation permet au Docker engine d'exécuter plusieurs conteneurs simultanément sur un même hôte. Docker fournit tout l'outillage nécessaire à la gestion des conteneurs.

Docker aide donc les utilisateurs à réduire le temps de mise sur le marché de leur application.

### Orchestration de conteneurs

Comme mentionné précédemment, les gestionnaires de conteneurs, en particulier Docker, facilitent la construction et déploiement d'applications. Au fur et à mesure que le nombre de conteneurs augmente, il devient très important d'automatiser la gestion de ces derniers. Cette automatisation est réalisée par un orchestrateur de conteneurs.

L'orchestration est l'ensemble des techniques permettant d'automatiser le déploiement, la gestion et la mise à l'échelle des conteneurs de manière horizontale sur plusieurs machines distantes et différentes. L'orchestration permet de provisionner et d'instancier de nouveaux conteneurs. Elle est également chargée de maintenir l'état sain du système en créant un nouveau conteneur en cas de défaillance ou de charge excessive. Enfin, elle maintient la connectivité entre les conteneurs et expose les services en cours d'exécution à des hôtes externes. Parmi les nombreux cadres d'orchestration disponibles sur le marché, les plus populaires sont Docker swarm, Kubernetes, Google container engine et Amazon ECS.

## 2.3 Cloud computing

Il existe plusieurs définitions du cloud computing. Cependant le NIST a fourni une définition assez complète.

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models." The National Institute of Standards and Technology (NIST), USA (2011) [17]

Les trois sections suivantes présentent les caractéristiques essentielles, les modèles de service et les modèles de déploiement du cloud computing tels que défini par le NIST.

### 2.3.1 Les cinq caractéristiques essentielles

Cinq caractéristiques sont essentielles lorsque l'on développe un service en cloud computing :

- Le service doit être en libre-service à la demande. Un utilisateur du service peut automatiquement demander un service amélioré ou réduit sans interaction humaine.
- Il doit être accessible sur l'ensemble d'un réseau. Le service fourni est disponible sur le réseau au moyen de protocoles standard. Cela permet à toute machine connectée à Internet d'y accéder.
- Il doit y avoir une mutualisation des ressources. Les clients du service sont regroupés sur la même machine. En outre, l'emplacement physique de la machine ne doit pas être apparent pour l'utilisateur, mais il peut être visible à un niveau supérieur, comme la région ou le pays.
- Il doit être rapidement élastique (adaptation rapide à une variation du besoin). Les capacités peuvent être rapidement et élastiquement provisionnées tel que de nouveau CPU ou RAM, parfois automatiquement, pour évoluer rapidement et être disponibles dans un temps réduit. Pour le consommateur, les capacités disponibles pour l'approvisionnement semblent souvent illimitées et peuvent être achetées en n'importe quelle quantité et à tout moment.
- Le service doit être mesurable, la façon dont les ressources d'un seul utilisateur sont utilisées est à la fois rassemblé et présenté à l'utilisateur.

### 2.3.2 Les modèles de services

Chacun des modèles de service offre différents niveaux de capacités et de responsabilités au fournisseur et au consommateur du service. Ces services peuvent s'appuyer sur une offre de service à partir d'un modèle de service de niveau inférieur. De cette manière, les services cloud dans les niveaux supérieurs, dont la plateforme et le logiciel, peuvent être entièrement construits au sommet d'autres services de cloud computing.

Le NIST distingue trois niveaux de service :

- Le logiciel en tant que service (SaaS) : La capacité offerte au consommateur consiste à utiliser les applications du fournisseur s'exécutant sur une infrastructure cloud. Exemple, Amazon SQS fournit une file d'attente de messages durable en tant que service.
- La plateforme en tant que service (PaaS) : la capacité offerte au consommateur est de déployer sur l'infrastructure du cloud des applications créées ou acquises par le consommateur, programmées avec des langages et outils pris en charge par le fournisseur. Exemple, la plateforme Heroku ou Amazon Elastic Beanstalk.
- L'infrastructure en tant que service (IaaS) : la capacité offerte au consommateur consiste à fournir une puissance de traitement, un espace de stockage, des réseaux et d'autres ressources informatiques fondamentales, en permettant au consommateur de déployer et d'exécuter des logiciels de son choix, notamment des systèmes d'exploitation et des applications. Exemple, AWS EC2 ou Azure VMs.

Les définitions IaaS, PaaS et SaaS ont été remarquablement résilientes et la plupart des fournisseurs de cloud computing utilisent encore ces termes dans leurs documents marketing. Mais au fil des années, le secteur a évolué rapidement pour inclure diverses technologies telles que les conteneurs et le serverless.

Une des évolutions de IaaS/PaaS/SaaS est Everything as a Service, typiquement appelé XaaS. La définition de XaaS est “toute technologie livrée sur Internet qui était auparavant livrée sur place”. Les fournisseurs de services dans le cloud sont en mesure d'offrir d'autres technologies comme services cloud maintenant parce que l'accès à Internet est devenu de plus en plus fiable et rapide, et que la virtualisation des serveurs et les progrès du sans serveur rendent de puissantes plateformes et services informatiques facilement accessibles. XaaS permet de réagir rapidement aux changements du marché. Pour en citer quelques-uns :

- Base de données en tant que service (DBaaS) : la capacité offerte aux consommateurs est une certaine forme d'accès à une base de données sans qu'il soit nécessaire de configurer le matériel physique, d'installer des logiciels ou de configurer les performances. Toutes les tâches administratives et la maintenance sont prises en charge par le fournisseur de services, de sorte que l'utilisateur ou le propriétaire de l'application n'a qu'à utiliser la base de données.
- Fonctions en tant que service (FaaS) : la capacité offerte au consommateur consiste à fournir une plateforme permettant aux clients de développer, d'exécuter et de gérer les fonctionnalités des applications sans la complexité de la mise en place et de la maintenance de l'infrastructure généralement associée au développement et au lancement d'une application.
- Le conteneur en tant que service (CaaS) : la capacité offerte au consommateur de gérer et de déployer des conteneurs, des applications et des clusters grâce à la virtualisation basée sur les conteneurs. Le CaaS est souvent considéré comme un sous-ensemble du IaaS, mais il inclut les conteneurs comme sa ressource fondamentale, par opposition aux machines virtuelles.
- Backend as a Service (BaaS) : Services tiers basés sur une API qui remplacent des sous-ensembles de fonctionnalités de base dans une application.
- Security as a Service (SECloudaaS) : Un fournisseur intègre ses services de sécurité dans une infrastructure d'entreprise sur la base d'un abonnement. Ces services de sécurité comprennent souvent, entre autres, l'authentification, l'anti-virus, l'anti-malware/spyware, la détection des intrusions et la gestion des événements de sécurité.

### 2.3.3 Les modèles de déploiement

Le Cloud Computing est une solution qui fournit un espace dans lequel il est possible de placer virtuellement des infrastructures serveur ou réseau, des plateformes de développement ou d'exécution. Il existe quatre modèles de déploiement et ils sont définis en fonction de leur relation avec une entreprise ou un organisme.

- Privé : Utilisé par un seul organisme, il peut être hébergé en interne ou en externe.
- Hybride : Mélange de plusieurs modèles de cloud reliés entre eux offrant les avantages des différents environnements.
- Communautaire : Partagé par plusieurs organismes, il est généralement hébergé en externe, mais peut être hébergé en interne par un des membres.
- Public : Déployé par un fournisseur de cloud tiers, il est ouvert au public et partagé par les utilisateurs.

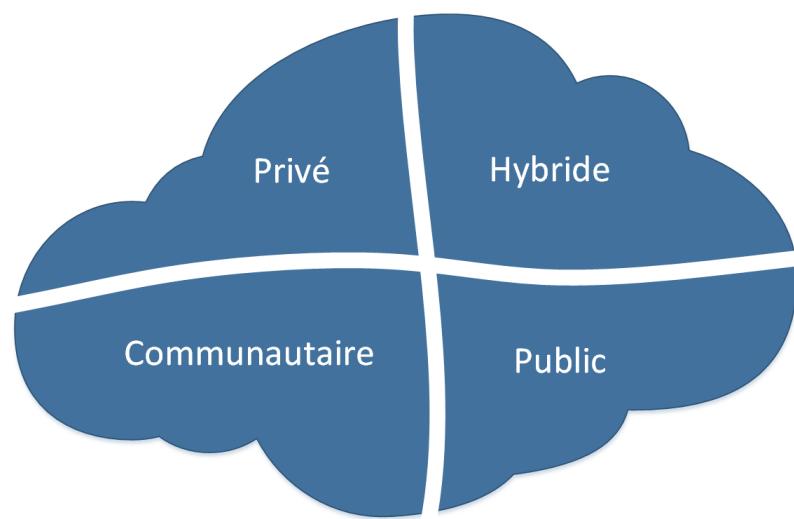


FIGURE 2.1 – Les 4 modèles de déploiement du cloud computing selon NIST [8]

# Approche architecturale

---

3.1	Architecture monolithique . . . . .	18
3.2	Architecture microservices event-driven . . . . .	20
3.3	Architecture serverless . . . . .	23
3.4	Design conceptuel . . . . .	25

---

Bien qu'il n'existe pas une seule approche pour concevoir une architecture dans le cloud, les différentes approches partagent les mêmes attributs de qualités d'évolutivité, de disponibilité et de haute fiabilité. En outre, il existe aujourd'hui une grande variété de solutions de plus en plus nombreuses et différentes, ainsi que différents fournisseurs publics de cloud computing, ce qui rend difficile le choix d'architecture et de fournisseur cloud. Dans ce chapitre, les différentes conceptions architecturales sont présentées et discutées. Ces architectures peuvent être construites sur tous les grands fournisseurs de cloud public comme Amazon, Google, Microsoft ou IBM.

## 3.1 Architecture monolithique

L'architecture monolithique 3.1 est considérée comme la façon traditionnelle de construire une application. Une application monolithique est construite comme une unité unique, indivisible et couplée. Habituellement, une telle solution se base sur une architecture 3-tier, une interface utilisateur côté client, une application côté serveur et une base de données. Elle est unifiée et toutes les fonctions sont gérées et servies en un seul endroit. Normalement, les applications monolithiques manquent de modularité et le couplage entre les différentes fonctions est très fort. En conséquence, la mise à jour logicielle devient très pénible car un changement affectera l'ensemble des liens couplés.

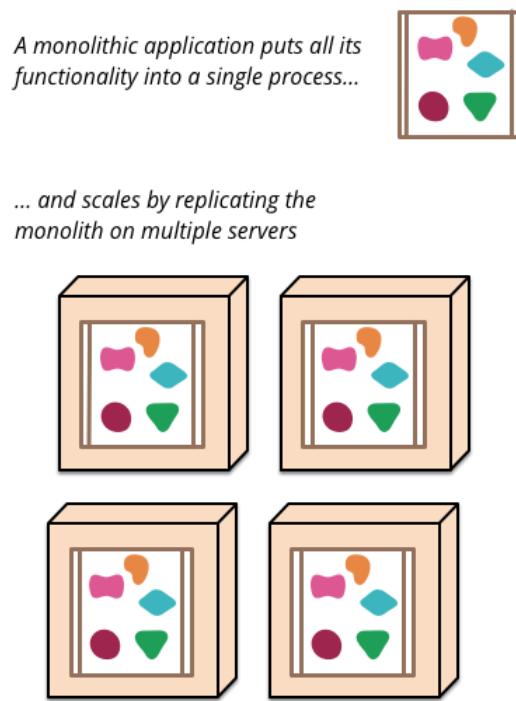


FIGURE 3.1 – Architecture monolithique [12]

### Avantages

- Simple à développer : Au début d'un projet, il est beaucoup plus facile d'opter pour l'architecture monolithique.
- Simple à mettre à l'échelle horizontalement en exécutant plusieurs copies derrière un équilibrEUR de charge. Cependant si une seule fonctionnalité du monolithe est la cause de cette mise à l'échelle, tout ce dernier doit être redéployé sur une nouvelle machine pour satisfaire les demandes. Ce qui n'est pas optimal en ce qui concerne les coûts de l'infrastructure.
- Simple à déployer : Un autre avantage associé à la simplicité des applications monolithiques est la facilité de déploiement. Lorsqu'il s'agit d'applications monolithiques, vous n'avez pas à gérer de nombreux déploiements.

## Désavantages

- Difficile à comprendre : Le couplage étroit et les nombreuses interconnexions entre modules dans une application monolithique peuvent entraîner une base de code et des interdépendances difficiles à comprendre dans leur intégralité. Les nouveaux développeurs devront avoir une bonne compréhension de la façon dont l'ensemble de l'application s'articule. Cela peut faire de l'intégration de nouveaux talents une tâche intimidante.
- Difficile à faire des changements : Il est plus difficile de mettre en œuvre des changements dans une application aussi vaste et complexe avec un couplage très serré. Tout changement de code affecte l'ensemble du système et doit donc être parfaitement coordonné. Cela rend le processus de développement global beaucoup plus long.
- L'évolutivité : Vous ne pouvez pas faire évoluer les composants indépendamment, mais seulement l'ensemble de l'application.
- Fiabilité : Un bogue dans n'importe quel module (par exemple, une fuite de mémoire) peut faire chuter tout le processus. De plus, comme toutes les instances de l'application sont identiques, ce bogue a un impact sur la disponibilité de l'application entière.

## 3.2 Architecture microservices event-driven

L'architecture de microservice 3.2 est une approche pour développer une application unique en tant que suite de petits services, chacun fonctionnant dans son propre processus et communiquant avec des mécanismes légers, souvent une API de ressources HTTP ou un bus de messages. Ces services s'articulent autour des capacités de l'entreprise et peuvent être déployés de manière indépendante par des machines de déploiement entièrement automatisées. Il y a un minimum de gestion centralisée de ces services, qui peuvent être écrits dans différents langages de programmation et utiliser différentes technologies de stockage de données.

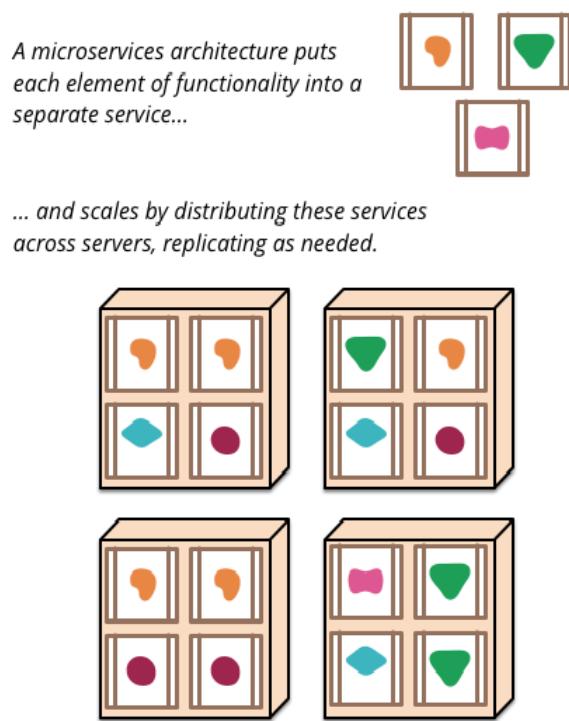


FIGURE 3.2 – Architecture de microservices [12]

Une architecture événementielle utilise des événements pour communiquer entre ces différents services qui sont totalement découpés. Un événement est un changement d'état, ou une mise à jour, comme un article placé dans un panier d'achats sur un site e-commerce. Les événements peuvent soit porter l'état (l'article acheté, son prix et une adresse de livraison), soit être des identificateurs (une notification qu'une commande a été expédiée).

Les architectures événementielles ont trois composants clés : les producteurs d'événements, les bus d'événements et les consommateurs d'événements. Un producteur publie un événement sur le bus, qui filtre et met à disposition sous forme de topic les événements aux consommateurs. Les consommateurs quant à eux souscrivent au topic où il reçoit l'information. Les services producteurs et les services consommateurs sont entièrement découpés, ce qui leur permet d'être mis à l'échelle, mis à jour et déployés indépendamment.

## Avantages

Un microservice s'adapte en fonction de la demande d'une certaine fonctionnalité. De cette façon, les ressources sont utilisées plus efficacement et l'architecture peut mieux prendre en charge l'évolution rapide du nombre d'appareils connectés dans le cas de Shayp.

Une architecture de microservices facilite le travail collaboratif et le test de fonctionnalités uniques, car chaque microservice peut être géré indépendamment. Chaque microservice pourrait être programmé dans un langage de programmation selon les préférences de l'équipe de développement ou les exigences d'un microservice. De plus, de nouvelles fonctionnalités supplémentaires peuvent facilement être ajoutées à l'architecture en créant un nouveau microservice.

Un autre avantage d'une architecture de microservices est l'augmentation de la tolérance aux pannes car une caractéristique importante d'un microservice est qu'ils sont très faiblement couplés. Donc la panne n'affecte pas les autres microservices.

Un microservice doit contenir tous les éléments nécessaires à l'exécution de sa fonction et cela va dans le même sens des avantages à l'utilisation des conteneurs. Cela contraste fortement avec les applications monolithiques qui existent comme un seul bloc de code et chaque module étant fortement interdépendant les uns des autres. De plus la maintenabilité devient extrêmement facile car chaque microservice respecte le principe de la responsabilité unique.

## Désavantages

Le plus grand inconvénient d'une architecture microservices est sa complexité accrue par rapport à une application monolithique. La complexité d'une application basée sur les microservices est directement corrélée au nombre de services impliqués. Ce type d'architecture comporte beaucoup plus d'éléments indépendants que les autres types d'architectures, ce qui nécessite des efforts considérables, une planification minutieuse et surtout une bonne documentation ainsi qu'une automatisation pour gérer la communication interservices, la surveillance, les tests et le déploiement.

Les microservices posent des défis en matière de sécurité. En effet en raison de l'augmentation des communications interservices sur le réseau, toutes ces interactions permettent à des entités extérieures d'avoir accès au système.

Une initiative de microservices exigera un changement de culture dans les organisations qui cherchent à les adopter. Ils exigent une culture agile et mature de développement. Avec une application basée sur les microservices, les équipes doivent être en mesure de gérer le cycle de vie complet d'un service. Cela nécessite souvent de transférer les compétences et la prise de décision des managers et des architectes aux équipes individuelles voire aux développeurs individuels. Ce changement de hiérarchie peut être difficile à accepter pour certaines personnes au sein de l'organisation. Par conséquent, il est important de s'assurer que tout le monde a adhéré à l'initiative. De plus, la communication entre les individus et les équipes devient beaucoup plus difficile, car les équipes n'ont pas toujours une vue d'ensemble et ne savent pas comment les différents services doivent travailler ensemble pour créer une application fonctionnelle.

Une organisation devra également déterminer si ses employés possèdent les compétences et l'expérience nécessaires pour prendre en charge une application basée sur les microservices. Comme une équipe peut être responsable d'un seul service, les développeurs doivent être bien informés sur le développement, le déploiement, les tests et la surveillance d'une application. Il sera également nécessaire de s'assurer que chaque équipe possède des compétences en matière de DevOps et d'automatisation des versions.

Ce désavantage est aussi valable pour l'architecture serverless expliquée ci-dessous.

### 3.3 Architecture serverless

”Serverless computing refers to the concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment.” *Cloud Native Computing Foundation*

L’architecture serverless ne signifie pas que nous n’utilisons plus de serveurs pour héberger et exécuter le code; elle ne signifie pas non plus que les ingénieurs sysadmin ne sont plus nécessaires. Il s’agit plutôt de l’idée que les utilisateurs n’ont plus besoin de consacrer du temps et des ressources à l’approvisionnement, à la maintenance, aux mises à jour, à la mise à l’échelle et à la planification de la capacité des serveurs. Au lieu de cela, toutes ces tâches et capacités sont gérées par une plateforme serverless et sont complètement absentes des développeurs ainsi que des équipes informatiques/opérationnelles. Par conséquent, les développeurs se concentrent sur l’écriture de la logique de leurs applications.

Une plateforme serverless peut fournir l’un des éléments suivants, ou les deux : FaaS et BaaS.

#### Avantages

- Coût opérationnel réduit : L’architecture serverless change radicalement le modèle de coût de l’exécution des applications logicielles en éliminant les frais généraux liés à la maintenance des ressources du serveur et en réduisant les coûts des employés (opérations / développement).
- Coût de mise à l’échelle : L’architecture serverless offre une mise à l’échelle horizontale automatique, élastique et surtout gérée par le fournisseur. Cela peut se traduire par plusieurs avantages, principalement au niveau infrastructure, mais surtout cela permet d’avoir une facturation très fine et de ne payer que des ressources consommées. En fonction du cas d’utilisation, cela peut engendrer une énorme économie sur la facture.

## Désavantages

- Verrouillage chez le fournisseur cloud : Création d'une dépendance forte avec le fournisseur de service. À ce jour, aucune spécification n'est sortie afin d'adopter un langage commun pour les fonctions entre les fournisseurs. Même si certains frameworks (Serverless.io) essaient de briser ces limitations, lors de la conception de votre solution et du choix des fonctionnalités, vous devrez faire le choix d'un fournisseur unique afin de garantir une certaine homogénéité de communication entre les différentes couches et pour pallier au verrouillage que les fournisseurs font de leurs services.
- Contrôle par le fournisseur cloud : Avec toute stratégie d'externalisation, vous abandonnez le contrôle d'une partie de votre système à un fournisseur tiers. Un tel manque de contrôle peut se manifester par des temps d'arrêt du système, des limites imprévues, des changements de coûts, la perte de fonctionnalités, des mises à niveau forcées de l'API, et plus encore.

Après avoir vu les différents types d'architecture, nous allons discuter du design conceptuel retenu.

### 3.4 Design conceptuel

La figure 3.3 ci-dessous reprend le design conceptuel de l'application d'ingestion de données provenant d'objets connectés. Shayp a actuellement des objets connectés avec deux types de protocole de communication, Sigfox et NB-IoT. L'architecture est divisée en deux parties individuelles pour chaque protocole. Pour maintenir ce découplage et cette distribution, une communication orientée événements est utilisée entre les différentes entités. Un bus d'événement permet de les relier tout en maintenant la distribution, car un événement peut apparaître à tout endroit et à tout moment sur le bus. Quant au découplage lui aussi continue d'exister, car l'événement lui-même ne connaît pas les conséquences de son apparition, une fois émis par une entité ce dernier ne sait pas ce qu'il va engendrer sur les autres. Ce qui fait qu'il n'existe aucun couplage logique entre les parties communicantes.

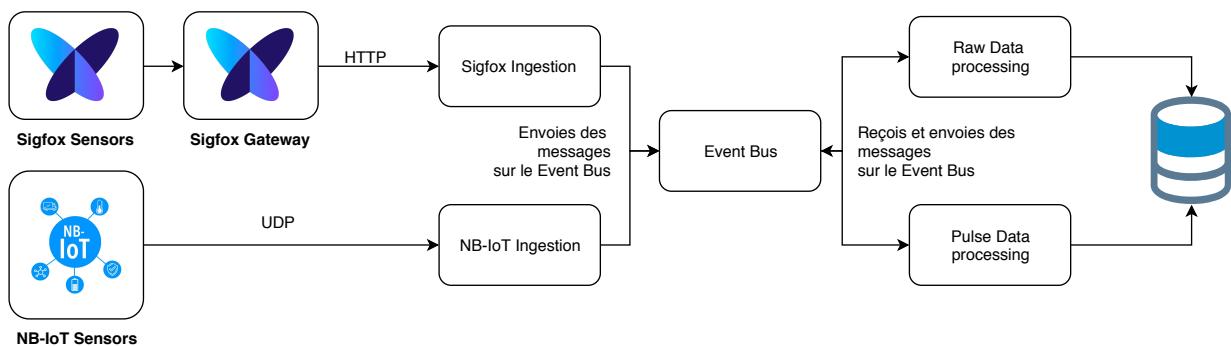


FIGURE 3.3 – Design conceptuel des différents composants de l'architecture

Ce design d'architecture permet aussi de maintenir une bonne évolutivité, car il rend l'architecture modulaire. En d'autres termes, il est possible d'ajouter de nouveaux systèmes à l'architecture de manière aisée, juste en les connectant au bus d'évènements et créer une API spécifique au système pour gérer les évènements envoyés et reçus sur le bus.

L'architecture peut être implémentée sous forme d'un ensemble de microservices ou de fonctions serverless. L'architecture Microservice - Serverless est implantée sur des services gérés par Amazon Web Service et l'architecture microservice est construite sur des machines virtuelles fournies par Scaleway. Ces deux fournisseurs cloud ont été choisis afin de pouvoir fournir aussi une comparaison entre plusieurs plateformes de tailles et types différents et notamment, car les deux plateformes sont utilisées au sein de Shayp. Chez Scaleway se trouve l'application backend contenant toute la logique business et chez AWS, nous avons le site web permettant aux utilisateurs d'accéder à leur consommation d'eau.

Les architectures sont évaluées en fonction des objectifs et contraintes définis au début de la thèse. A cela, s'ajoute une estimation des coûts pour les implémentations sur les différents fournisseurs de cloud computing. En outre, les inconvénients des architectures implémentées sont exposés et les améliorations possibles sont présentées.

Nous allons commencer par l'architecture mixte Microservice - Serverless chez AWS.

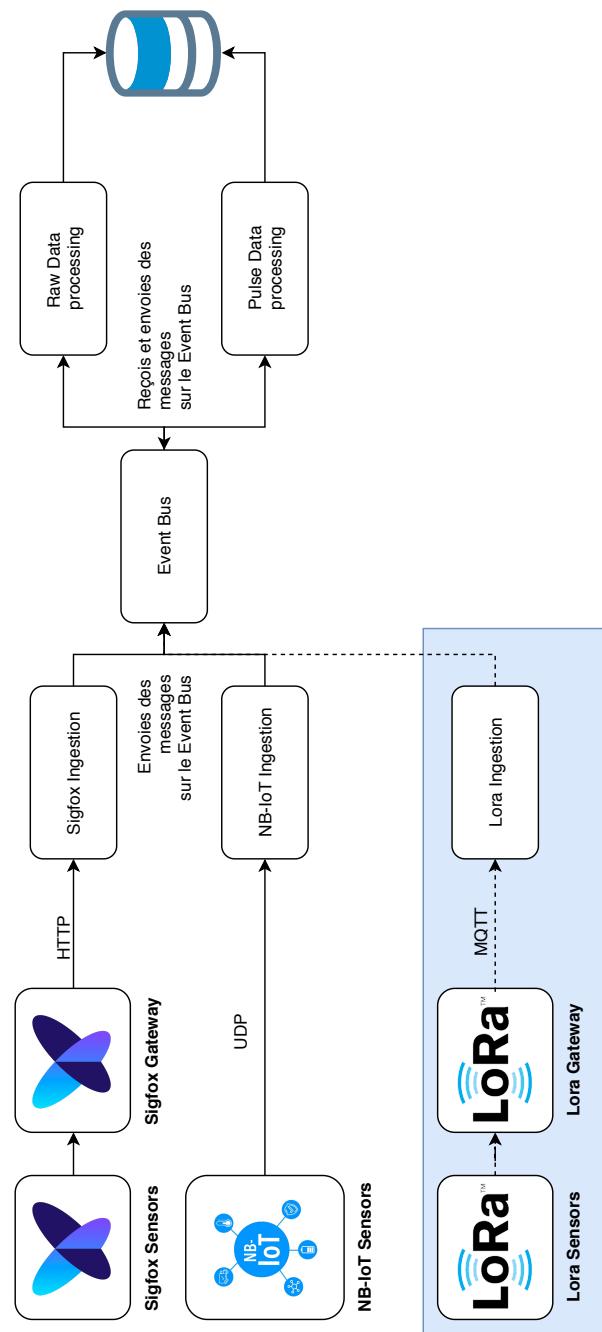


FIGURE 3.4 – Design conceptuel des différents composants de l'architecture avec un nouveau protocol

# Architecture Serverless - Microservice chez AWS

---

4.1	Composants de l'architecture . . . . .	30
4.2	Fonctionnement . . . . .	33
4.3	Performances . . . . .	37
4.4	Bilan . . . . .	39
4.5	Estimation des coûts . . . . .	41
4.6	Inconvénients et améliorations possibles . . . . .	45

---

L'implémentation proposée n'est pas totalement serverless à cause des objets connectés en NB-IoT qui imposent une communication en UDP qui à l'heure actuelle n'est pas possible avec ce paradigme.

La figure 3.3 peut être implémentée au niveau d'AWS suivant les deux variantes présentées ci-dessous. Dans les deux implémentations, la partie processing de l'architecture est totalement serverless et se base sur des événements. Pour la partie ingestion de l'architecture, deux options se posent. Rendre serverless l'ingestion des données Sigfox (figure 4.1) ou utiliser un microservice et les serveurs déjà provisionnés (figure 4.2). Les deux options se valent du point de vue du design mais se déparent en terme de coût. Avant tout, le rôle des différents composants de l'implémentation va être détaillé et une vue complète de l'architecture est disponible en annexe A.2.

Variant 1 - Ingestion mixte (serverless et microservice) & data processing serverless

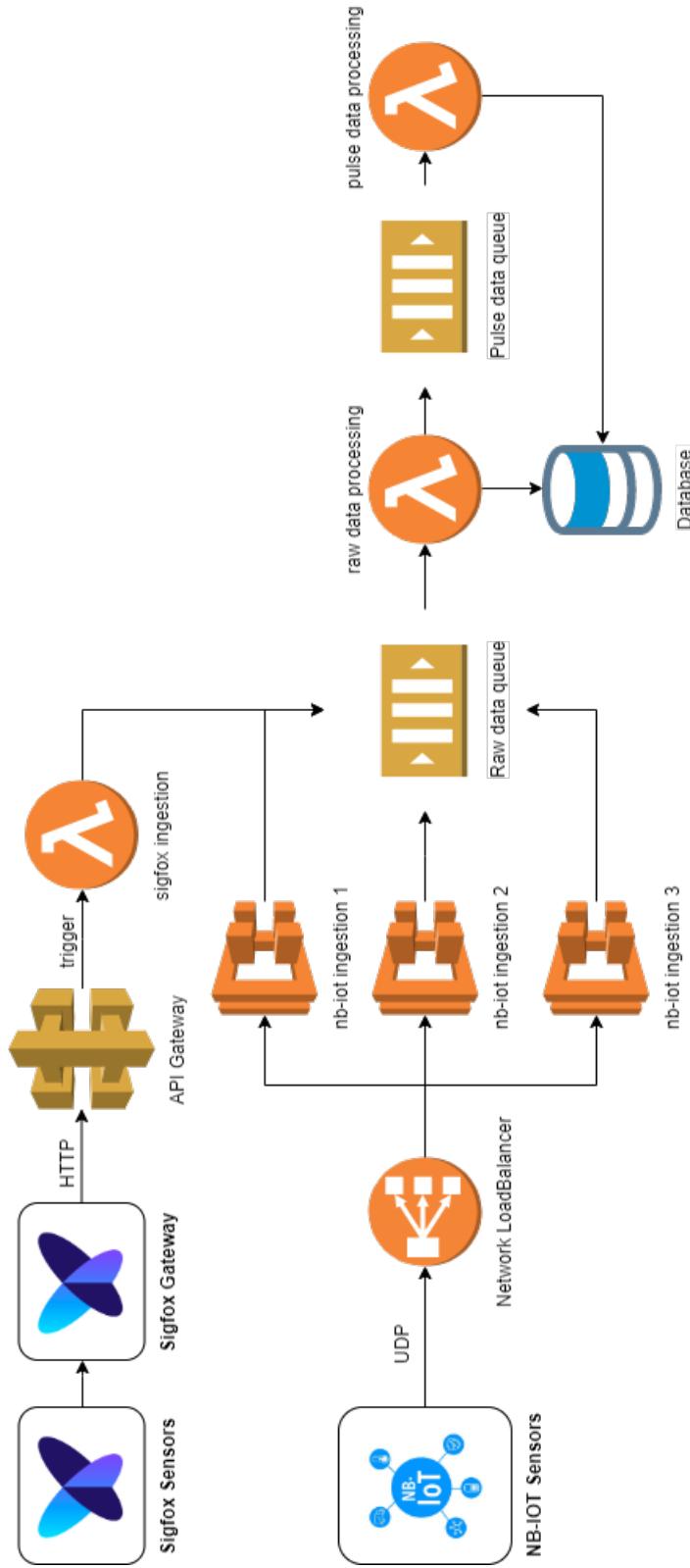


FIGURE 4.1 – Variante 1 - Ingestion mixte (serverless et microservice) & data processing serverless

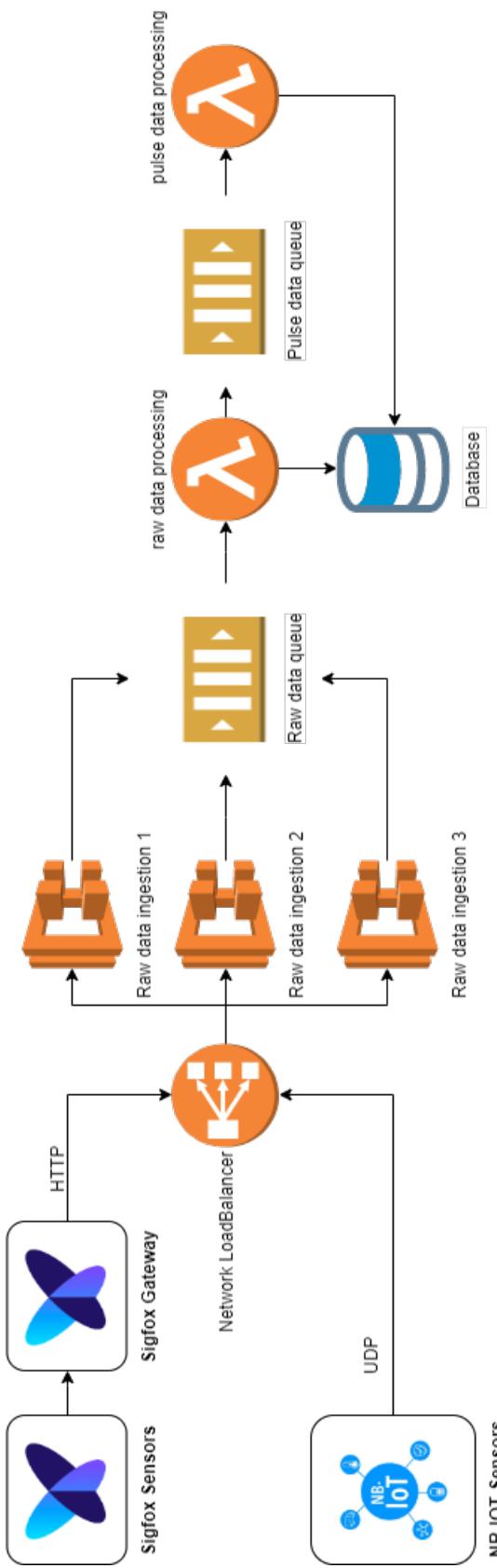
**Variant 2 - Ingestion microservice & data processing serverless**

FIGURE 4.2 – Variante 2 - Ingestion microservice &amp; data processing serverlesse

## 4.1 Composants de l'architecture

### Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) est un service web qui permet d'acheter de la capacité de calcul dans le cloud, qui est redimensionnable quand l'utilisateur le souhaite. Pour provisionner les instances virtuelles, une Amazon Machine Image (AMI) est nécessaire. Cette image est un pack comprenant un système d'exploitation, les paramètres de configuration associés et possiblement un set de logiciels préinstallés. Les instances peuvent être lancées dans une ou plusieurs régions géographiques. Chaque région contient plusieurs "zones de disponibilité (A-Z)" avec des emplacements distincts. Si une panne survient dans une zone de disponibilité spécifique, une autre zone de disponibilité dans la même région peut fournir des services via une connectivité réseau à faible latence.

### Amazon Elastic Load Balancing (ELB)

Lorsque le trafic augmente pour une application, il est préférable de ne pas allouer tout le trafic à une seule instance. Au lieu de cela, le trafic entrant devrait être réparti par un équilibrEUR de charge. Le trafic entrant est automatiquement réparti sur plusieurs instances grâce à l'équilibrage élastique de la charge. Cet équilibrage de charge vous permet également d'obtenir une meilleure tolérance aux pannes dans vos applications en fournissant la capacité d'équilibrage de charge nécessaire en fonction de la charge de trafic actuelle. L'équilibrEUR de charge élastique détecte les instances malsaines et redirige automatiquement le trafic vers les instances saines jusqu'à ce que les instances malsaines aient été restaurées. L'équilibrage de charge élastique peut être activé à l'intérieur d'une même zone de disponibilité ou sur plusieurs zones (pour des performances applicatives encore plus constantes). L'équilibrEUR de charge d'Amazon est divisé en deux sous-catégories, Application Load Balancing (ALB) et Network Load Balancing (NLB).

Pour le cas de notre application, nous utilisons l'équilibrEUR de charge réseau. La communication vers et depuis un équilibrEUR de charge réseau AWS fonctionne sur la couche 4 (Transport) du modèle OSI.

### Amazon Elastic Container Service (ECS)

Amazon ECS est un service d'orchestration de conteneurs propriétaire pour des applications conteneurisées construites avec Docker. Amazon ECS gère et exécute des conteneurs sur un cluster EC2. Les instances de conteneur peuvent être mises à l'échelle horizontalement en ajoutant de nouvelles instances de conteneur au cluster.

ECS est utilisé pour déployer le service qui va gérer l'ingestion des données UDP pour la variante 1 et pour la variante 2 aussi le second service pour l'ingestion des données HTTP.

### Amazon Elastic Container Registry (ECR)

ECR est un registre de conteneurs Docker entièrement géré qui permet aux développeurs de stocker, gérer et déployer facilement des images de conteneur Docker. Amazon ECR s'intègre à ECS, afin de simplifier le flux de travail, du développement à la production. Amazon ECR héberge les images dans une architecture hautement disponible et évolutive, pour assurer un déploiement fiable des conteneurs destinés aux applications.

## AWS Cloud Watch and AWS Auto Scaling Group (ASG)

AWS Cloud Watch peut surveiller différents types de mesures des différents services AWS. Une mesure pourrait être la consommation de mémoire ou l'utilisation du CPU. Des alarmes peuvent être configurées pour certains événements ou franchissements de seuil. Elles peuvent avertir l'utilisateur du service ou prendre une mesure automatisée comme la mise à l'échelle automatique d'AWS. La mise à l'échelle automatique d'AWS met à l'échelle différents services AWS comme ECS avec des politiques et des plans différents en fonction des alarmes fournies par AWS Cloud Watch. La mise à l'échelle automatique AWS doit être configurée pour ajuster correctement les ressources à la demande actuelle. Toutefois, dans la plupart des cas, les ressources doivent être surprovisionnées, car il est difficile de prévoir la charge et donc de réagir à temps avec une quantité correcte de ressources.

## Amazon Virtual Private Cloud (VPC)

VPC est une section logiquement isolée dans le cloud AWS, où le consommateur peut contrôler l'environnement réseau. Différents services proposés par AWS peuvent être mis dans un VPC, il n'y a donc pas de possibilité d'accès direct depuis Internet, ce qui augmente la sécurité d'un service.

## Amazon API Gateway

Amazon API Gateway est un service entièrement opéré, qui permet aux développeurs de créer, publier, gérer, surveiller et sécuriser facilement des API à n'importe quelle échelle. Les API servent de "porte d'entrée" pour que les applications puissent accéder aux données, à la logique métier ou aux fonctionnalités de vos services backend. À l'aide d'API Gateway, vous pouvez créer des API HTTP, API RESTful et des API WebSocket qui permettent de concevoir des applications de communication bidirectionnelle en temps réel. API Gateway prend en charge les charges de travail conteneurisé et sans serveur, ainsi que les applications web.

API Gateway gère toutes les tâches liées à l'acceptation et au traitement de plusieurs centaines de milliers d'appels d'API simultanés, notamment la gestion du trafic, la prise en charge de CORS, le contrôle des autorisations et des accès, la limitation, la surveillance et la gestion de la version de l'API. Aucun frais minimum ou coût initiaux ne s'appliquent à API Gateway. Vous payez pour les appels API que vous recevez et la quantité de données transférées et, avec le modèle de tarification par paliers de l'API Gateway, vous pouvez réduire vos coûts en fonction de l'utilisation de votre API.

## AWS Lambda

AWS Lambda est un FaaS sans serveur qui exécute le code backend du serveur sans configurer et gérer une plateforme ou une infrastructure. AWS Lambda est provisionné sur le temps de calcul consommé. Les fonctions AWS Lambda peuvent être écrites dans plusieurs langages de programmation. Les fonctions AWS Lambda peuvent être déclenchées par plusieurs services AWS différents, par exemple un événement Amazon un appel REST ou SQS.

## Amazon Simple Queue Service (SQS)

La solution Amazon Simple Queue Service (SQS) est un service de file d'attente de messagerie entièrement géré qui permet de découpler et mettre à l'échelle des microservices, des systèmes décentralisés et des applications sans serveur. SQS élimine la complexité et les frais généraux associés à la gestion et à l'utilisation de messages orientés intergiciel, et permet aux développeurs de se concentrer sur la différenciation des tâches.

## 4.2 Fonctionnement

On remarque sur le schéma de l'implémentation, 4.1 et 4.2, qu'il y a une partie de l'architecture exposée à internet et une autre partie qui est privée.

Pour le protocole de communication Sigfox, le service est soit exposé grâce à l'API Gateway et une fonction Lambda pour la variante 1 ou un microservice conteneurisé dans ECS derrière le répartiteur de charge pour la variante 2.

Le microservice s'occupant de l'ingestion des données NB-IoT est exposé par ECS avec un port UDP ouvert et reçoit les requêtes réceptionnées par le répartiteur de charge réseaux.

Pour ce qui est de l'acheminement des données, il reste assez similaire pour chaque protocole.

### SIGFOX

#### Variante 1 - Ingestion mixte (serverless et microservice) & data processing serverless

1. Les capteurs sont configurés pour envoyer des données aux gateways Sigfox les plus proches à un intervalle d'une heure.
2. Le gateway reçoit les paquets et reconnaît les capteurs par leur identifiant. Il génère un JSON avec les données et envoie une requête HTTP vers l'URL indiquée.
3. L'API Gateway réceptionne la requête, la vérifie et exécute une fonction Lambda en lui donnant en paramètre le contenu de la requête en question.
4. Les données utiles sont extraites et envoyées dans un message sur une queue SQS *ProcessRawDataSigfox* par la fonction Lambda.
5. Dès qu'un message est présent sur la queue *ProcessRawDataSigfox*, une fonction Lambda est exécutée pour traiter le message. Elle récupère le message, adapte les données au bon format pour être stocké dans la base de données. Ensuite elle génère un nouveau message contenant l'ID créé dans la base de données qu'elle publie sur la queue *ProcessPulseDataSigfox*.
6. Dès qu'un message est présent sur la queue *ProcessPulseDataSigfox*, une fonction Lambda est exécutée pour traiter le message. Elle récupère le message, adapte les données au bon format pour être stocké dans la base de données.

**Variante 2 - Ingestion microservice & data processing serverless**

1. Les capteurs sont configurés pour envoyer des données aux gateways Sigfox les plus proches à un intervalle d'une heure.
2. La requête est transmise par le répartiteur de charge vers une des machines disponibles.
3. La requête est validée par le microservice (Sigfox Raw data ingestion) et les données utiles sont extraites et envoyées dans un message sur une queue SQS *ProcessRawDataSigfox*.
4. Dès qu'un message est présent sur la queue *ProcessRawDataSigfox*, une fonction Lambda est exécutée pour traiter le message. Elle récupère le message, adapte les données au bon format pour être stocké dans la base de données. Ensuite elle génère un nouveau message contenant l'ID créé dans la base de donnée qu'elle publie sur la queue *ProcessPulseDataSigfox*.
5. Dès qu'un message est présent sur la queue *ProcessPulseDataSigfox*, une fonction Lambda est exécutée pour traiter le message. Elle récupère le message, adapte les données au bon format pour être stocké dans la base de données.

**NB-IoT**

L'acheminement des données est identique pour les deux variantes.

1. Les capteurs sont configurés pour envoyer des données en UDP à un intervalle de une heure vers un nom de domaine donné.
2. La requête est transmise par le répartiteur de charge vers une des machines disponibles.
3. La requête est validée par le microservice (NB-IoT Raw data ingestion) et les données utiles sont extraites et envoyées dans un message sur un topic *ProcessRawDataNBIoT* du bus de communication.
4. Le microservice (NB-IoT Raw data processing) abonné au topic *ProcessRawDataNBIoT* récupère le message, adapte les données au bon format pour être stocké dans la base de données. Ensuite il génère un nouveau message contenant l'ID créé dans la base de donnée qu'il publie sur le topic *ProcessRawDataNBIoT*.
5. Le microservice (NB-IoT Pulse data processing) abonné au topic *ProcessPulseDataNBIoT* récupère le message, adapte les données au bon format pour être stocké dans la base de données.

## Gestion de l'autoscaling par AWS

Un microservice, illustré en détail à la figure 4.3, est un service avec une définition de tâche dans ce que nous appellerons AWS ECS. Une définition de tâche a un type de lancement, qui est dans ce cas-ci AWS ECS. AWS ECS déploie les instances conteneur de la tâche sur notre cluster EC2, qui est configuré au préalable. Le cluster fournit des ressources virtuelles, sur lesquelles les instances de tâche du service s'exécutent. Le provisionnement des ressources pour les instances de tâches est géré par AWS ECS. En outre, le service dans ECS peut être mis à jour avec une nouvelle révision de la définition des tâches. Ainsi que, dans un service, le nombre d'instances de tâches souhaitées, minimum et maximum, peut être défini.

AWS Cloud Watch surveille les instances de tâche d'un service sur différents indicateurs, par exemple le CPU ou l'utilisation de la mémoire. Un service peut avoir des stratégies configurées avec des alarmes, qui déclenchent l'échelle automatique AWS des instances de tâches. Une stratégie doit être configurée de manière à disposer de suffisamment de temps pour ajuster le nombre d'instances de tâches en fonction d'un changement de charge. Par exemple, dans le cas d'une politique d'utilisation de la mémoire supérieure à 50%, une nouvelle instance de tâche est utilisée. La mise à l'échelle et la réduction d'échelle des instances de conteneurs dans l'ECS prennent plusieurs minutes à partir de l'initiation jusqu'à un état sain. La mise à l'échelle des instances se fait par étapes avec une période de refroidissement configurable entre les deux. Cela conduit à un service qui doit généralement être surprovisionné pour s'ajuster à temps pour une éventuelle charge plus élevée à venir. L'implémentation de cette fonctionnalité nous permet de ne plus gérer le provisionnement de nouvelles instances avec la croissance du parc d'objet connecté.

L'équilibrEUR de charge réseau envoie périodiquement des demandes de santé aux machines virtuelles afin de déterminer les cibles saines qui sont donc aptes à recevoir de nouvelles requêtes. Les demandes des objets connectés en UDP arrivent à l'équilibrEUR de charge et il ne les distribue qu'à des cibles saines. L'équilibrEUR de charge réseau sélectionne une instance cible avec un algorithme de hachage de flux, qui dirige tous les appels d'un même client vers la même cible tant que la connexion existe. Une cible malsaine ne reçoit aucune demande jusqu'à ce qu'un bilan de santé réussit. Si une instance cible est plantée ou malsaine pendant une période prolongée, elle est arrêtée et une nouvelle instance est créée pour elle. S'il n'y a pas de cible saine dans un groupe cible, une demande est rejetée avec un code d'erreur.

Pour la variante 2, ECS contient deux services au lieu d'un seul et l'équilibrEUR de charge route aussi les requêtes HTTP. Il reçoit les requêtes via le protocole TCP et envoie les requêtes TCP à l'instance cible. L'équilibrEUR de charge transmet la demande sans ouvrir ou modifier la partie HTTP de la demande.

Pour la partie serverless de l'implémentation, AWS met automatiquement à l'échelle les différents services (Lambda et SQS) en cours d'exécution en fonction de la charge actuelle avec une limite de 1000 exécutions concurrentes pour tous les lambdas du compte.

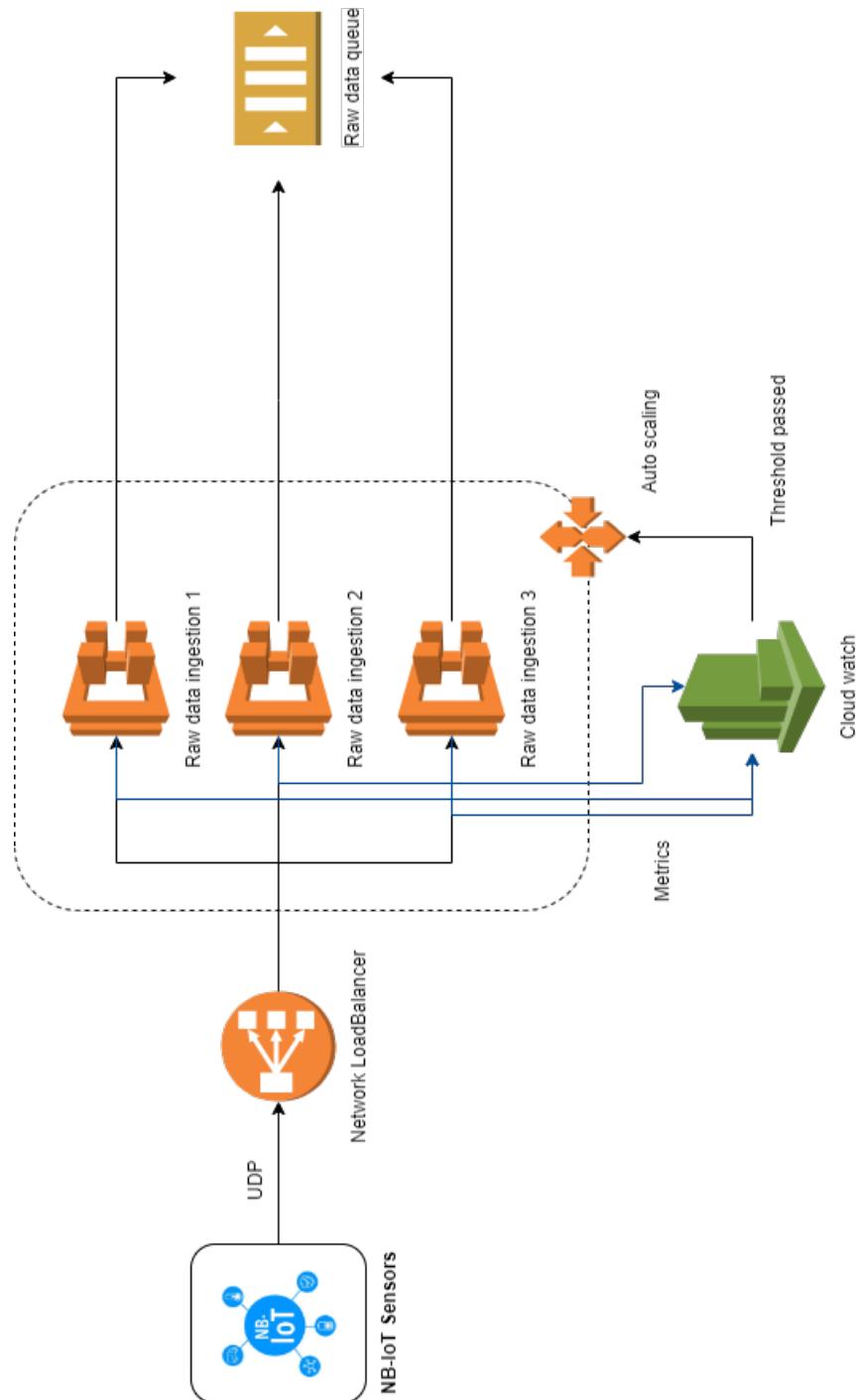


FIGURE 4.3 – Gestion de la mise à échelle des services pour l'ingestion de données UDP

## 4.3 Performances

Les objets connectés envoient les informations vers le serveur une fois par heure. Pour tester les performances de l'implémentation, une série de 3 tests de charge a été effectuée afin de simuler un set d'objets connectés. Les tests de charge ont été effectués sur la variante 2 de l'architecture.

- Simulation de 180 000 / 360 000 / 3 600 000 objets connectés
- Trois machines virtuelles au sein du cluster ECS - 2 vCPUs et 2 Go de mémoire
- Une machine pour simuler le trafic - 8 vCPUs - 32 Go de mémoire
- Répartition des objets : 50% NB-IoT / 50% Sigfox

Quatre métriques ont été surveillées et récupérées pour vérifier le bon fonctionnement de l'implémentation :

- Le nombre total de messages traités par SQS
- Le temps d'exécution moyen des lambdas
- Le nombre total de lambda exécutés
- L'utilisation du processeur par le cluster ECS :

L'utilisation du cluster est mesurée comme le pourcentage de CPU et de mémoire qui est utilisée par toutes les tâches Amazon ECS sur un cluster par rapport à la somme de CPU et de mémoire qui a été enregistrée pour chaque instance de conteneur actif dans le cluster.

Pour mieux comprendre, prenons l'implémentation actuelle. Trois instances s'enregistrent dans le cluster avec 1792 unités (2 vCPUS) de CPU et 1707 Mib de mémoire chacun. Les ressources globales de ce cluster sont de 5376 unités de CPU et 5121 Mio de mémoire. Si dix tâches sont exécutées sur ce cluster et que chaque tâche consomme 256 unités de CPU et 256 Mo de mémoire, un total de 2 560 unités de CPU et 2 560 Mo de mémoire sont utilisés sur le cluster. Cela est rapporté comme une utilisation de 47% du CPU et 50% de la mémoire pour le cluster.

TABLE 4.1 – Performances de l'architecture microservice - serverless

Objets	<b>180000</b>	<b>360000</b>	<b>3600000</b>
CPU Usage by cluster	8%	14%	160%
Total Lambda invocation	360 000	720 000	7 200 000
Average Lambda execution time	150ms	150ms	150ms

Il y a une surutilisation du processeur par le cluster. Cela est dû à la configuration du test de charge. L'envoi des 1000 requêtes par seconde est fait depuis une seule machine. Or le répartiteur de charge envoie toutes les requêtes provenant d'un même hôte vers la même machine cible tant que la session est ouverte. La durée de vie de cette session est gérée par le répartiteur de charge. En observant l'utilisation du CPU général des trois machines virtuelles, figure 4.4, confirme notre hypothèse. Théoriquement si les requêtes sont réparties sur les trois machines, le cluster résiste à la charge sans problème.

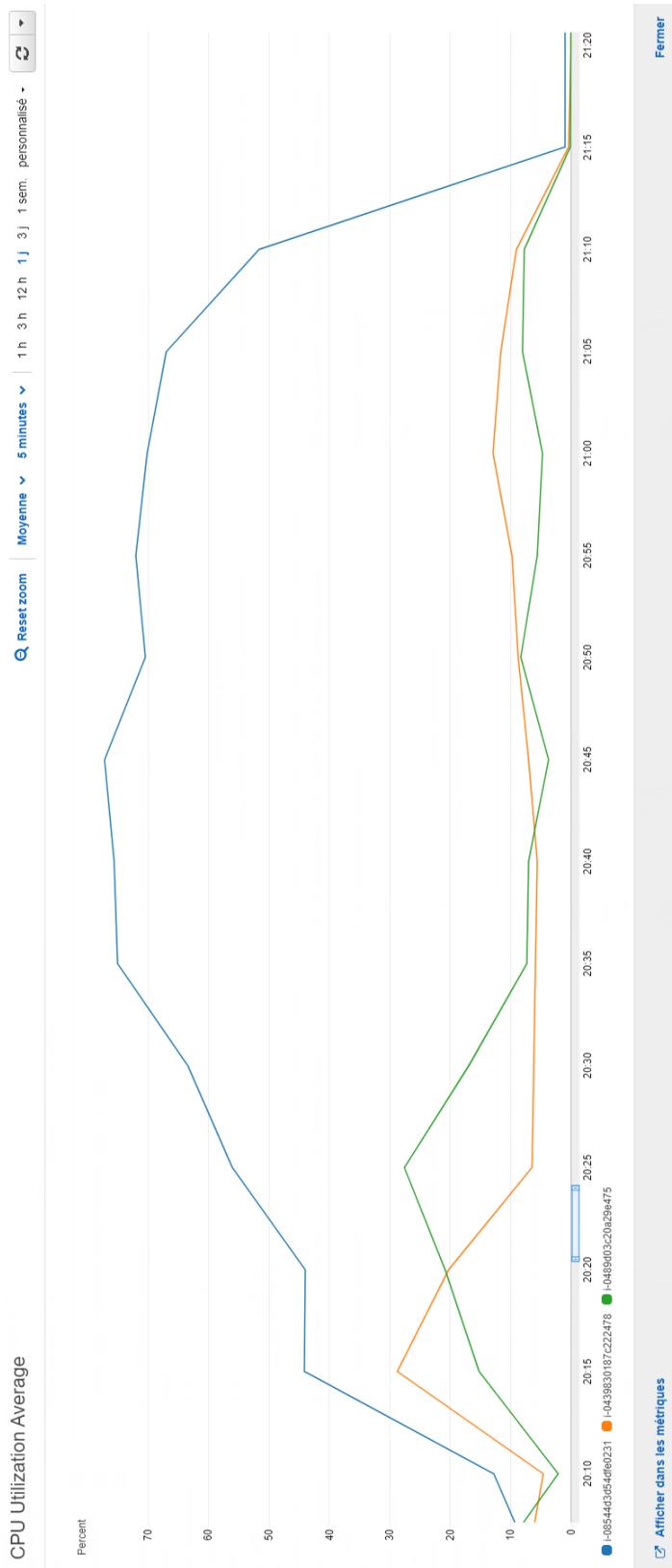


FIGURE 4.4 – Utilisation du CPU par les trois machines virtuelles pour 3M6 objets connectés sur une heure

## 4.4 Bilan

Cette section fait le bilan de l'architecture Serverless - Microservice chez AWS sur base des objectifs définis précédemment au chAPITre 1.1.

### Disponibilité

La disponibilité de l'architecture est évaluée sur plusieurs parties différentes qui pourraient être des points de défaillance uniques dans celle-ci.

AWS garantit pour tous ses services un contrat de niveau de service (SLA) indiquant le pourcentage de disponibilité et de temps de fonctionnement du service en question. Si la disponibilité n'est pas atteinte, des crédits de service sont accordés aux clients.

- ECS et EC2 : AWS garantit pour les instances ECS et EC2 un SLA de 99,99% (AWS, 2019), ce qui correspond à un temps d'arrêt autorisé d'environ 5 minutes par mois. AWS propose différentes zones de disponibilité isolées au sein d'une région afin d'offrir une disponibilité plus élevée en reproduisant les services dans différentes zones. Différentes instances de tâches dans ECS peuvent être déployées dans différentes zones de disponibilité.
- API Gateway : AWS garantit pour l'API gateway un SLA de 99,95% (AWS, 2019), ce qui correspond à un temps d'arrêt autorisé d'environ 22 minutes par mois.
- Network Load Balancer : AWS garantit pour le répartiteur de charge réseau un SLA de 99,99% (AWS, 2019), ce qui correspond à un temps d'arrêt autorisé d'environ cinq minutes par mois.
- SQS : AWS garantit pour SQS un SLA de 99,9% (AWS, 2019), ce qui correspond à un temps d'arrêt autorisé d'environ 44 minutes par mois.
- Lambda : AWS garantit pour les lambda un SLA de 99,95% (AWS, 2019), ce qui correspond à un temps d'arrêt autorisé d'environ 22 minutes par mois.

### Mise à l'échelle

La partie Serverless permet une excellente utilisation des ressources en fournissant automatiquement la bonne quantité de ressources aux différents services. La mise à l'échelle des ressources est rapide, mais elle a certaines limites d'extensibilité par exemple dans le nombre d'invocations simultanées des fonctions Lambda qui est de 1000. C'est la limite imposée par défaut par AWS sur tous les comptes, mais cette limite peut être augmentée après une requête auprès du service clientèle d'AWS.

L'API Gateway et l'équilibrEUR de charge réseau AWS sont mis à l'échelle automatiquement par AWS.

Pour la partie microservice, la mise à l'échelle des instances de tâches du conteneur ECS prend plusieurs minutes après que la quantité de tâches souhaitée ait été modifiée par une alarme Cloud Watch ou une modification manuelle. C'est pourquoi les microservices doivent être surprovisionnés pour servir toutes les requêtes avec au moins une tâche toujours en cours d'exécution. L'utilisation des ressources n'est donc pas optimale, car les ressources provisionnées sont en général toujours supérieures à la demande réelle de ressources.

## Hétérogénéité et flexibilité

L'ajout d'un nouveau type d'appareil connecté est simple grâce à la modularité de l'architecture. Si le nouvel objet communique via HTTP, pour l'ingestion des données en fonction de la variante, il faut créer une nouvelle fonction Lambda ou créer un nouveau microservice. Pour la partie processing des données, mettre à jour les fonctions *Raw data processing* et *Pulse data processing*.

## Optimisation des ressources et des coûts

Les ressources nécessaires dans la partie Serverless de l'architecture sont faibles, car l'environnement Lambda et SQS prêt à l'emploi peuvent être utilisés directement avec une faible charge de travail pour l'installation et la configuration. De plus, le temps nécessaire pour déployer les ajustements et les extensions de la logique applicative est faible. Tandis que la quantité de ressources nécessaires pour la partie microservice de l'architecture est conséquente, car une charge de travail élevée est nécessaire pour la mise en place et la configuration de celle-ci. De plus, le développement et l'extension d'un microservice sont très longs, car le service doit être construit comme un conteneur Docker et déployé dans l'Amazon Elastic Container Registry pour chaque changement.

## 4.5 Estimation des coûts

L'estimation des coûts est effectuée en fonction des tests de charge effectués précédemment.  
Notes :

- La taille d'un message HTTP est de 750 Bytes et est considérée comme la taille moyenne générale.
- Taille message SQS : 224 Bytes
- Trois machines virtuelles t3a.small avec 2 vCPUs et 2 Go de mémoire.
- Un répartiteur de charge réseau.
- Répartition des objets : 50% NB-IoT / 50% Sigfox

### 4.5.1 Variante 1 - Ingestion mixte (serverless et microservice) & data processing serverless

TABLE 4.2 – Estimation des coûts pour la variante 1 - Ingestion mixte (serverless et microservice) & data processing serverless

Objets	1 000	10 000	180 000	360 000	3 600 000
SQS (en \$)	0.58	5.84	105.12	210.24	2,102.40
NLB (en \$)	0.00	0.02	0.30	0.59	5.91
Lambda (en \$)	0.17	4.39	195.71	398.28	4,044.64
API Gateway (en \$)	0.00	2.94	71.82	144.74	1,457.43
EC2 (en \$)	43.57	43.57	43.57	43.57	43.57
<b>Total par mois (en \$)</b>	<b>44.32</b>	<b>56.76</b>	<b>416.51</b>	<b>797.43</b>	<b>7,653.95</b>
<b>Coût par mois et par objet (en \$)</b>	<b>0.0443</b>	<b>0.0057</b>	<b>0.0023</b>	<b>0.0022</b>	<b>0.0021</b>

### Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et service

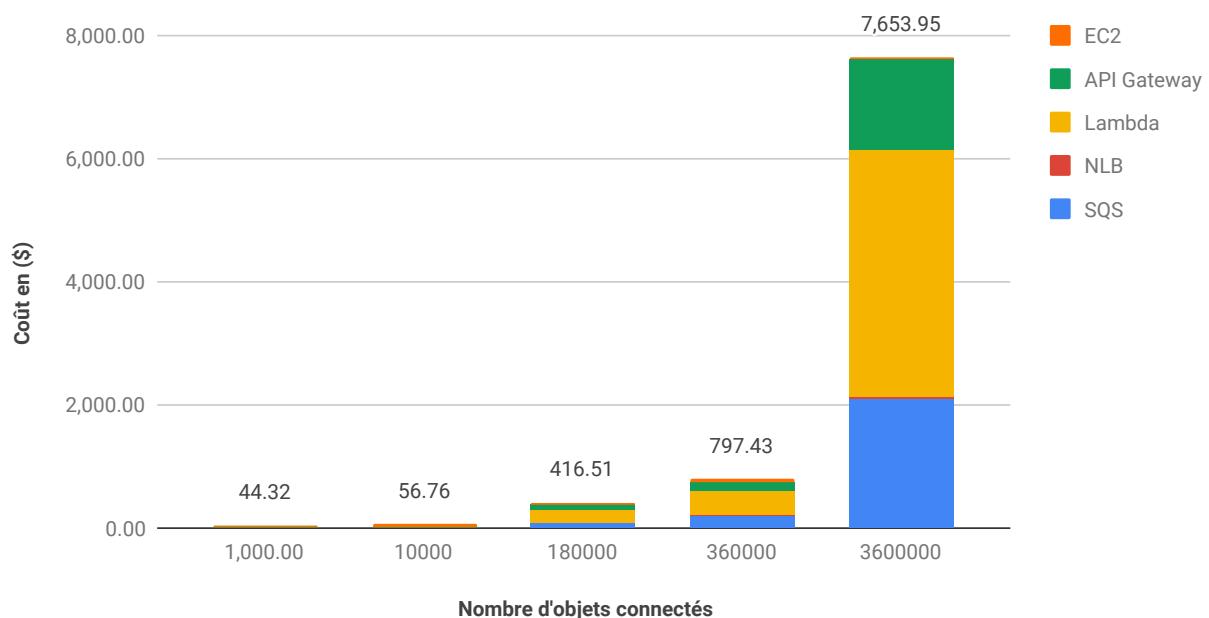


FIGURE 4.5 – Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services

#### 4.5.2 Variante 2 - Ingestion microservice & data processing serverless

TABLE 4.3 – Estimation des coûts pour la variante 2 - Ingestion microservice & data processing serverless

Objets	1 000	10 000	180 000	360 000	3 600 000
SQS (en \$)	0.58	5.84	105.12	210.24	2,102.40
NLB (en \$)	0.00	0.03	0.59	1.18	11.83
Lambda (en \$)	0.09	2.72	155.19	317.25	3,234.34
EC2 (en \$)	43.57	43.57	43.57	43.57	43.57
<b>Total par mois(en \$)</b>	<b>44.25</b>	<b>52.16</b>	<b>304.48</b>	<b>572.25</b>	<b>5,392.14</b>
<b>Coût par mois et par objet(en \$)</b>	<b>0.04425</b>	<b>0.00522</b>	<b>0.00169</b>	<b>0.00159</b>	<b>0.00150</b>

On constate que le plus gros coût de l'implémentation est la partie Serverless avec les lambda et SQS. Si on regarde la variante 1 avec l'API Gateway, le coût est encore plus élevé, car nous ajoutons une fonction lambda qui s'occupe des requêtes HTTP alors que coût des serveurs EC2 ne change pas. Réduire la capacité des instances EC2 ne contre balancera pas le coût de l'API Gateway et de la fonction Lambda.

### Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et service

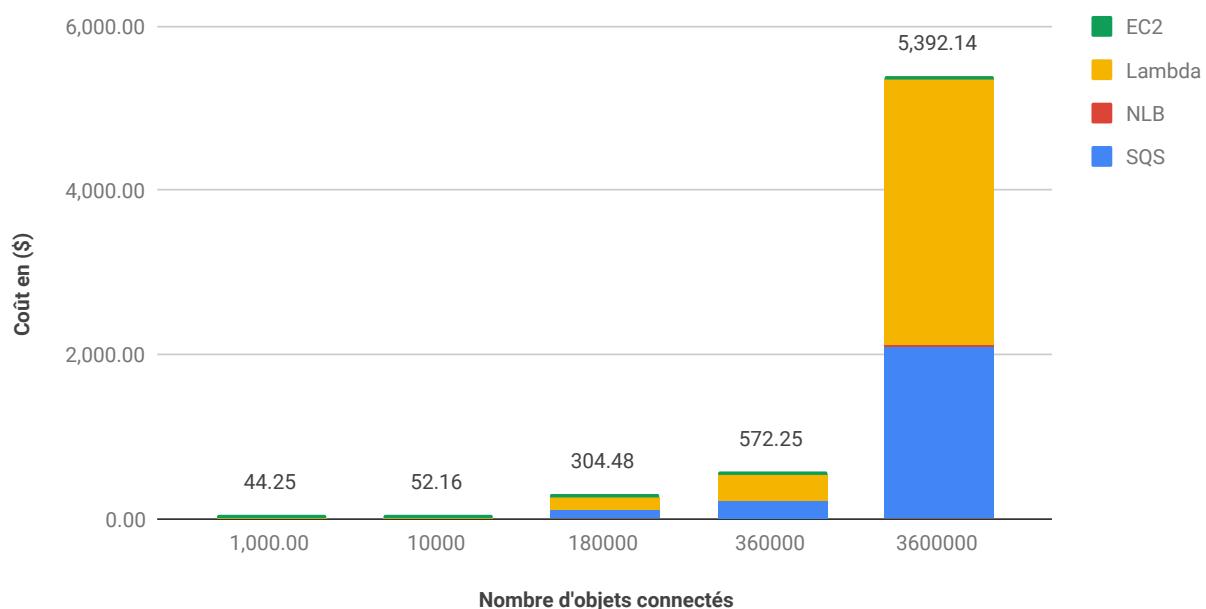


FIGURE 4.6 – Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services

Voici une liste de quelques pièges courants lorsqu'il s'agit de comprendre le coût de l'AWS Lambda.

#### Blocs de 100 ms

Avec AWS Lambda, l'utilisateur est facturé pour les demandes d'invocation ainsi que la durée des invocations.

Lorsque la fonction est exécutée, une invocation qui dure 2 secondes coûterait deux fois plus cher qu'une invocation qui dure 1 seconde. Mais il y a une petite nuance - les durées d'exécutions sont facturées par blocs de 100 ms. Une invocation qui dure 150 ms serait arrondie à 200 ms et coûterait donc autant qu'une invocation qui ne dure que 105 ms. Cela signifie également qu'il n'y a aucun avantage financier à optimiser les fonctions avec un temps d'exécution moyen déjà inférieur à 100 ms.

### Plus de mémoire ne signifie pas toujours plus cher

Avec AWS Lambda, le coût par invocation est également influencé proportionnellement par la taille de la mémoire de la fonction. Une fonction avec 256MB de mémoire coûterait deux fois plus par seconde qu'une fonction avec 128MB de mémoire. De plus, les ressources CPU sont également allouées proportionnellement - plus de mémoire égale plus de CPU. En combinaison avec les blocs de charge de 100 ms, cela ouvre une possibilité d'optimisation intéressante. Considérons une fonction de 128 Mo avec un temps d'exécution moyen de 110 ms. Comme la durée est arrondie aux 100 ms les plus proches, cela signifie que l'utilisateur sera facturé en moyenne 200 ms de temps d'exécution, soit \$0,000000416. Vous pouvez donner à la fonction plus de mémoire pour accélérer la partie du code qui n'attend pas la fin d'une opération d'E/S, et réduire le temps d'exécution moyen. Avec 192 Mo de mémoire, il est possible de réduire le temps d'exécution moyen à moins de 100ms, et réduire le coût moyen d'invocation à \$0.000000313. Cela représente une économie de 25% par invocation.

Déterminer la taille de la mémoire à utiliser nécessite des beaucoup d'essais.

### Services périphériques

Un autre coût souvent négligé de l'utilisation de AWS Lambda réside dans les services utilisés pour surveiller les fonctions. CloudWatch, par exemple, est un service obligatoire. Tout ce qui est écrit par la fonction sur stdout sera capturé et envoyé dans les Logs CloudWatch. Même si rien n'est écrit par la fonction, le service AWS Lambda écrira toujours trois messages système pour START, END et REPORT.

CloudWatch Logs facture \$0,5 par Go ingéré ainsi que \$0,03 par Go par mois pour le stockage. Il est très courant que les utilisateurs dépensent plus pour les logs CloudWatch que pour le AWS Lambda dans leur compte AWS de production. C'est pourquoi il est conseillé de n'utiliser que des échantillons de logs de débogage en production.

Le service Cloudwatch n'apparaît pas dans le tableau ci-dessus, car il est très difficile de déterminer son coût à cause de sa présence obligatoire dans plein de service AWS.

## 4.6 Inconvénients et améliorations possibles

### 4.6.1 Le coût peut être sensible à l'échelle

Les Lambda sont souvent utilisés en conjonction avec SQS ou Kinesis (alternative à Kafka) pour effectuer des traitements en arrière-plan. SQS est facturé uniquement par requête, tandis que Kinesis facture les heures de partition en plus des requêtes d'ajout. Cela fait de Kinesis une source d'événements relativement coûteuse lorsque le débit est faible. Mais avec la quantité de messages reçus par notre architecture, regardons si Kinesis n'est pas une meilleure option.

La différence de coût présentée est due au fait que Kinesis a un coût par million de demandes beaucoup plus faible, soit \$0,016 et \$0,4 pour SQS respectivement. Cela permet au coût de Kinesis d'augmenter à un rythme beaucoup plus lent à mesure que le débit augmente. Cela fait de Kinesis une option très intéressante pour les systèmes qui doivent fonctionner à l'échelle.

Nous obtenons donc une nouvelle estimation de coût, figure 4.7 et figure 4.4. Il serait donc intéressant pour une implémentation en production de remplacer SQS par Kinesis.

### 4.6.2 Traitement par lots

Lors de la configuration de l'intégration des événements SQS avec Lambda, il est possible de configurer une propriété batchSize. Celle-ci spécifie le nombre maximum de messages SQS que AWS enverra à la fonction Lambda sur un seul déclenchement. C'est une propriété intéressante et puissante, mais il faut faire attention à ce qu'elle soit bien réglée pour répondre aux besoins.

Avec l'intégration SQS / Lambda, un lot de messages réussit ou échoue ensemble. C'est un point important. Supposons que le batchSize est réglé à 10 messages, ce qui est la valeur par défaut. Si la fonction est invoquée avec 10 messages, et qu'elle retourne une erreur lors du traitement du 7e message, les 10 messages resteront dans la file d'attente pour être traités par une autre fonction Lambda. AWS ne supprimera les messages de la file d'attente que si la fonction s'est exécutée avec succès sans aucune erreur.

S'il est possible qu'un des messages échoue alors que d'autres réussissent, il faut prévoir la résilience de l'architecture. Il est possible de pallier ce problème de plusieurs manières :

- Utilisation d'un batchSize de 1, afin que les messages réussissent ou échouent d'eux-mêmes. C'est la méthode utilisée dans l'implémentation, mais elle implique donc un coût plus élevé.
- S'assurer que le traitement est idempotent, afin que le retraitement d'un message ne soit pas nuisible, en dehors du coût de traitement supplémentaire.
- Gérer les erreurs dans le code de la fonction, peut-être en les attrapant et en envoyant le message à une file d'attente d'erreurs pour un traitement ultérieur.
- Appeler manuellement l'API DeleteMessage dans la fonction après avoir traité un message avec succès.

L'approche à choisir dépend des besoins de l'architecture.

En appliquant ce traitement par lots de 10 et en ayant ajouté aussi Kinesis, pour la variante 2 de l'architecture nous obtenons une estimation de coût tel que présenté dans la table 4.5 et la figure 4.8.

Il est tout à fait possible d'augmenter le nombre de messages dans le traitement par lot afin de

réduire encore plus le coût des Lambdas.

Après avoir fini l'architecture serverless, nous allons présenter une architecture microservice basé sur un set d'outils open-source.

TABLE 4.4 – Estimation des coûts pour la variante 2 - Ingestion microservice & data processing serverless avec Kinesis comme Event Bus

Objets	1 000	10 000	180 000	360 000	3 600 000
Kinesis (en \$)	0.02	12.65	16.75	21.08	111.54
NLB (en \$)	0.00	0.03	0.59	1.18	11.83
Lambda (en \$)	0.09	2.72	155.19	317.25	3,234.34
EC2 (en \$)	43.57	43.57	43.57	43.57	43.57
<b>Total par mois (en \$)</b>	<b>43.69</b>	<b>58.98</b>	<b>216.10</b>	<b>383.09</b>	<b>3,401.28</b>
<b>Coût par mois et par objet (en \$)</b>	<b>0.0437</b>	<b>0.0059</b>	<b>0.0012</b>	<b>0.0011</b>	<b>0.0009</b>

### Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et service

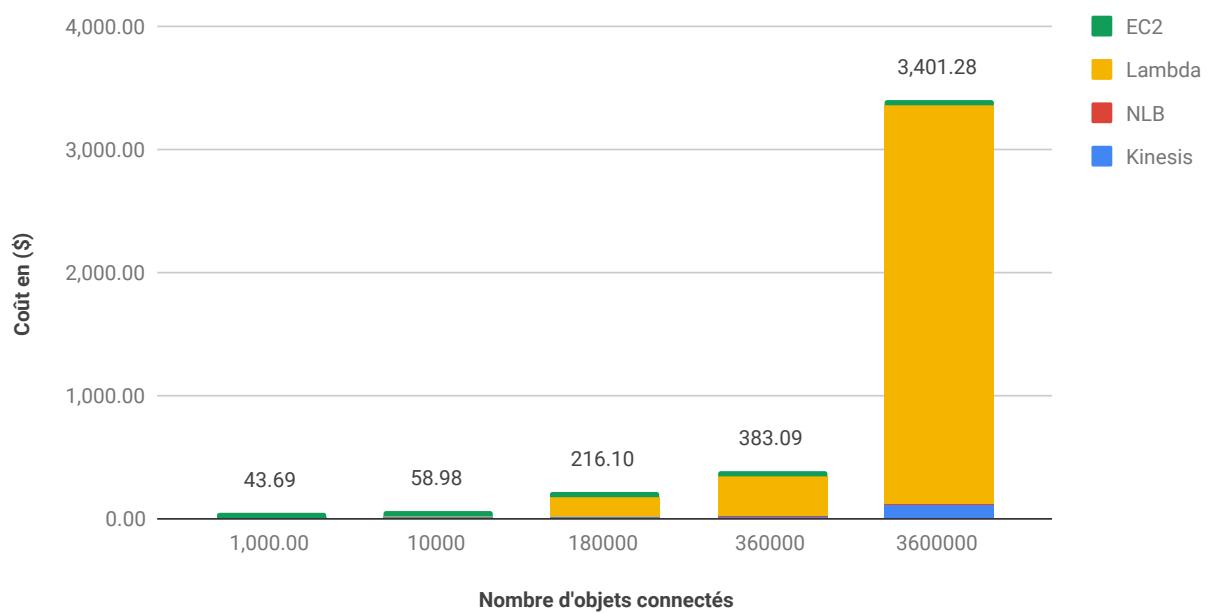


FIGURE 4.7 – Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services avec Kinesis comme Event Bus

TABLE 4.5 – Estimation des coûts pour la variante 2 - Ingestion microservice & data processing serverless avec Kinesis et traitement par lots de 10

Objets	1 000	10 000	180 000	360 000	3 600 000
<b>Kinesis (en \$)</b>	0.02	12.65	16.75	21.08	111.54
<b>NLB (en \$)</b>	0.00	0.03	0.59	1.18	11.83
<b>Lambda (en \$)</b>	0.00	1.41	82.27	171.40	1,775.80
<b>EC2 (en \$)</b>	43.57	43.57	43.57	43.57	43.57
<b>Total par mois (en \$)</b>	43.60	57.66	143.18	237.24	1,942.74
<b>Coût par mois et par objet (en \$)</b>	0.0436	0.0058	0.0008	0.0007	0.0005

### Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et service

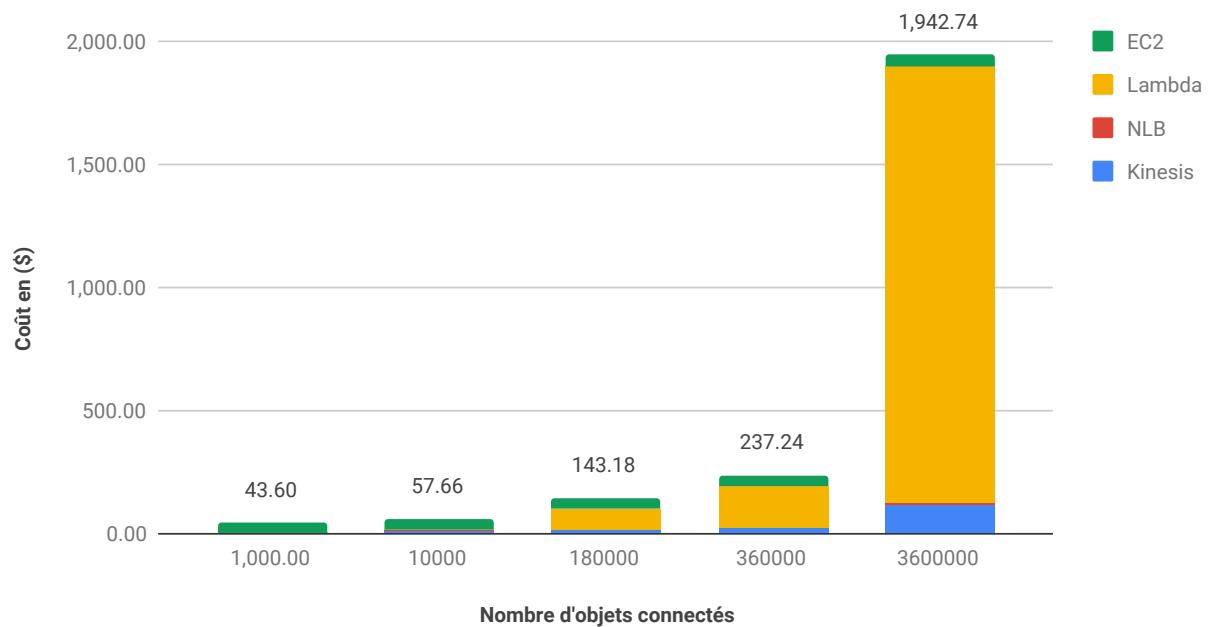


FIGURE 4.8 – Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services avec Kinesis comme Event Bus et traitement par lot de 10

# Architecture microservices chez Scaleway

---

5.1	Composants de l'architecture . . . . .	50
5.2	Fonctionnement . . . . .	55
5.3	Performances . . . . .	57
5.4	Bilan . . . . .	58
5.5	Estimation des coûts . . . . .	60

---

La figure 5.1 représente l'implémentation du design conceptuel. Une vue détaillée de tous les microservices et leur répartition sur les différentes machines est disponible en annexe A.3.

On retrouve des composants similaires à l'architecture implémentée chez AWS tels que le microservice s'occupant de l'ingestion des données Sigfox et NB-IoT, celui qui s'occupe du processing des raw data, celui qui s'occupe du processing des pulses data ainsi qu'une queue pour agir en tant que bus de communication. À ces composants fort similaires à la première architecture s'ajoutent un certain nombre de nouveaux microservices, ces microservices ont pour objectif de surveiller et gérer l'état du cluster Docker Swarm.

Ces microservices ont été implémentés, car notre système doit être constamment surveillé afin de s'assurer du bon fonctionnement de l'application et aussi, car Scaleway comparé à AWS n'offre pas un service de monitoring et logging intégré à leur plateforme (Cloudwatch pour AWS). Dans la section suivante, le rôle de chaque composant va être détaillé.

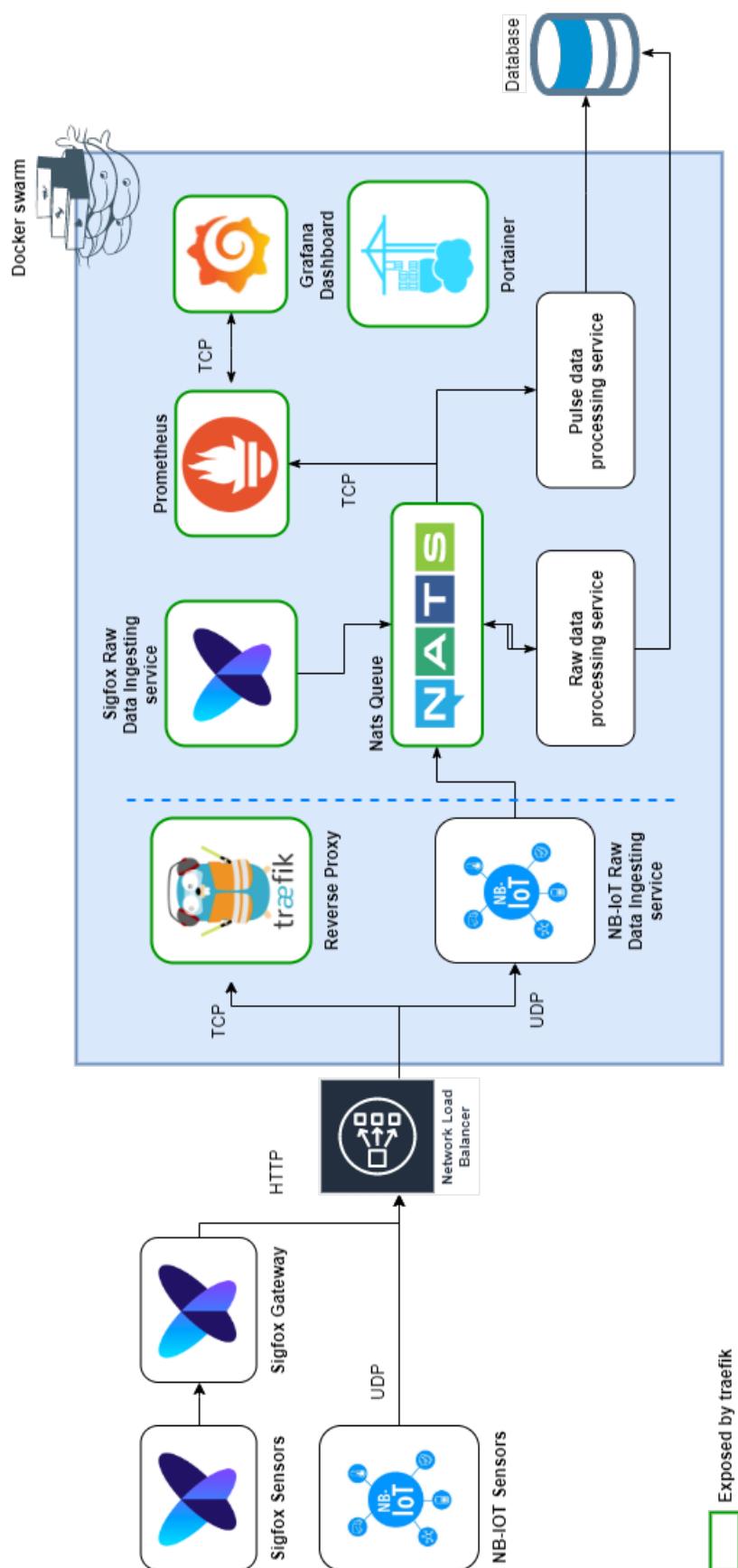


FIGURE 5.1 – Architecture microservices basée sur les événements

## 5.1 Composants de l'architecture

### Docker Swarm

Docker Swarm est l'outil d'orchestration et de regroupement de conteneurs par défaut de Docker. Il permet de déployer et d'orchestrer des conteneurs sur un grand nombre d'hôtes. Entre Docker Swarm et ECS, Docker Swarm est le plus léger et le plus facile à utiliser, car il est déjà intégré à Docker Engine.

Un swarm est un groupe de noeuds qui comprend :

- Nœuds de gestion (managers) : Contrôle l'orchestration, la gestion des clusters et la distribution des tâches.
- Nœuds de travail (workers) : Le seul but des workers est d'exécuter les conteneurs et les services tels qu'ils sont assignés par un noeud manager.
- Services : Un service décrit comment un conteneur individuel se distribue sur les noeuds. Pour créer un service, spécifiez les informations exactes comme dans une "exécution de docker" ordinaire, plus de nouveaux paramètres (c'est-à-dire, le nombre de copies de conteneurs).
- Tâches : Les conteneurs individuels placent le travail dans ces "slots" selon le gestionnaire Swarm.

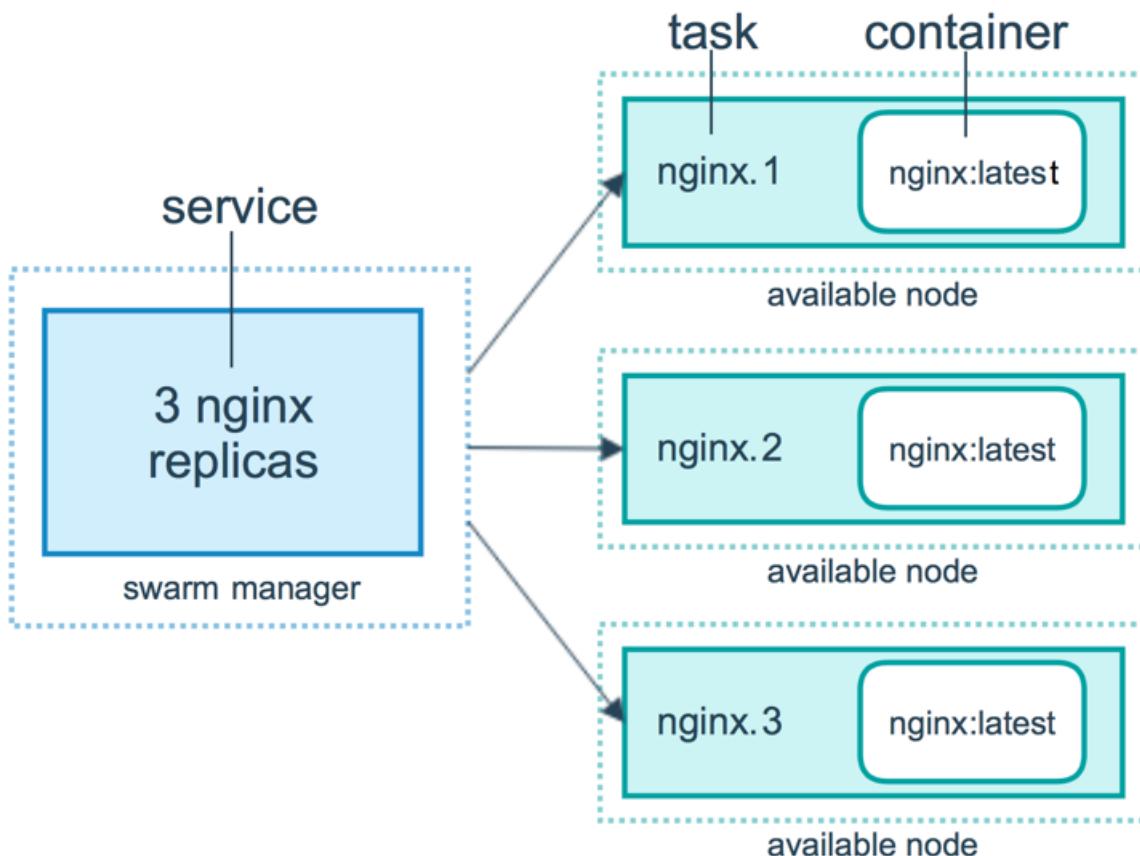


FIGURE 5.2 – Déploiement d'un service [21]

Le cluster est constitué de six machines au total, 3 managers et 3 workers. Nous avons 3 managers

pour que le cluster soit hautement disponible. Swarm utilise le protocole de consensus Raft. Le cluster Swarm ne peut conserver toutes ses fonctionnalités que si plus de la moitié de tous les nœuds managers sont encore disponibles. Par conséquent, si nous pouvons tolérer la perte d'un nœud manager, alors nous sommes obligés d'avoir 3 managers. Si nous pouvons tolérer la perte de 2 nœuds managers, nous devons en avoir 5 au total. Et ainsi de suite.

Si nous comparons ECS à Docker Swarm, les noeuds managers existent aussi, mais ils sont gérés par AWS. Toutes les machines créées dans un cluster ECS sont des noeuds workers.

## Traefik

Un proxy inverse (reverse proxy) est un type de serveur habituellement placé en amont des serveurs web. Contrairement au serveur proxy qui permet à un utilisateur interne d'accéder au réseau Internet. Le proxy inverse permet à un utilisateur d'Internet d'accéder à des serveurs ou services internes. Une des applications courantes du proxy inverse est la répartition de charge.

Traefik est un reverse proxy open source écrit en Go qui permet de déployer facilement une infrastructure microservices et la rendre publiquement accessible sur internet. Traefik fait le lien entre les requêtes venant d'internet (entrypoints) et les services déployés en interne. Il fait aussi office de service discovery et de load balancer.

Dans l'implémentation, Traefik est placé en amont et gère toutes les requêtes entrantes HTTP(S).

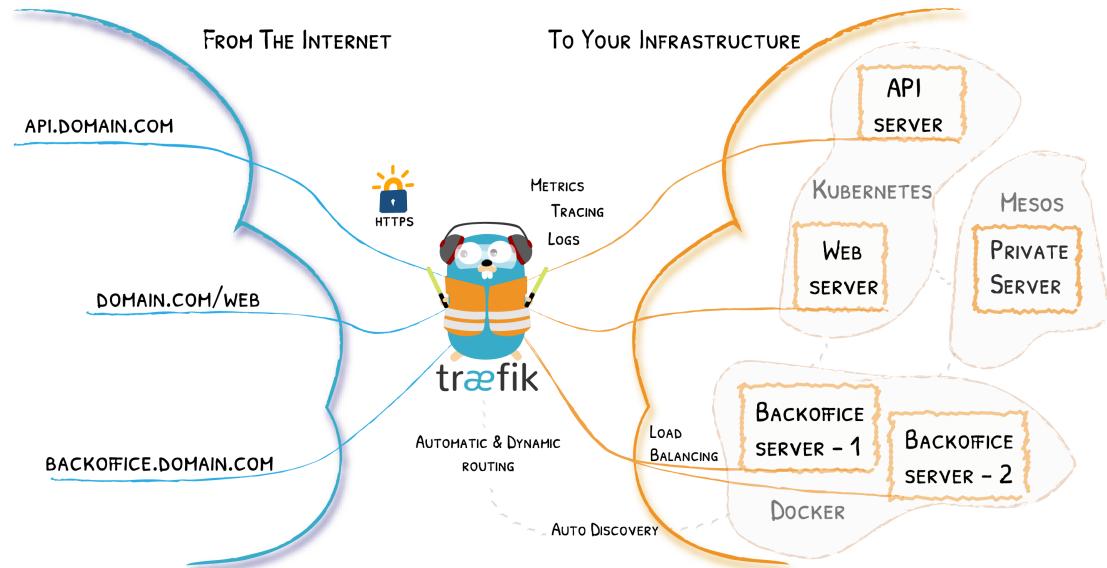


FIGURE 5.3 – Architecture Traefik [22]

## Répartiteur de charge (CenturyLink)

Scaleway n'offre pas un répartiteur de charge réseau sur sa plateforme comme AWS avec Elastic Load balancer. Il n'est pas possible aussi d'utiliser celui-ci pour faire de la répartition de charge externe. Il a donc fallu trouver une plateforme qui offre des LoadBalancer as a service (LBaaS). Notre choix s'est porté sur CenturyLink.

## Prometheus

Prometheus est un logiciel libre utilisé pour la surveillance et l'alerte d'événements. Il enregistre des métriques en temps réel dans une base de données de séries temporelles construite en utilisant un modèle HTTP pull avec des requêtes flexibles et une alerte en temps réel. Des agents récupèrent les métriques utiles des différents systèmes et les envoient au service Prometheus.

## NATS

NATS Server est un système de messagerie open source simple et performant pour les applications natives du cloud, la messagerie IoT et les architectures microservices. Il s'agit d'un système de messagerie M2M cloud-native basé sur le protocole applicatif NATS. Il offre des performances et une robustesse idéale pour les systèmes de contrôles event driven qui ont besoin d'envoyer et de recevoir des commandes et informations rapidement. Il est utilisé au coeur de l'architecture pour la transmission des événements entre les différents services d'ingestion et de processing.

## Grafana

Il s'agit d'un outil de visualisation graphique pour les données horodatées. Il faut d'abord le connecter à une source de données en l'occurrence Prometheus dans notre cas, et il suffit de créer des graphes en faisant des requêtes avec un langage semblable au SQL. Grafana avec Prometheus remplace donc Cloudwatch et sont donc utilisés pour surveiller l'état des différents hôtes au sein du cluster.

## Portainer

Portainer est une interface de gestion légère qui permet de gérer facilement un hôte Docker ou un cluster Swarm. Il est conçu pour être aussi simple à déployer qu'à utiliser. Il permet de gérer le Docker engine, les conteneurs, les images, les volumes et les réseaux. Portainer permet donc de gérer tout le cluster tout comme l'interface graphique ECS pour AWS.



FIGURE 5.4 – Dashboard Grafana

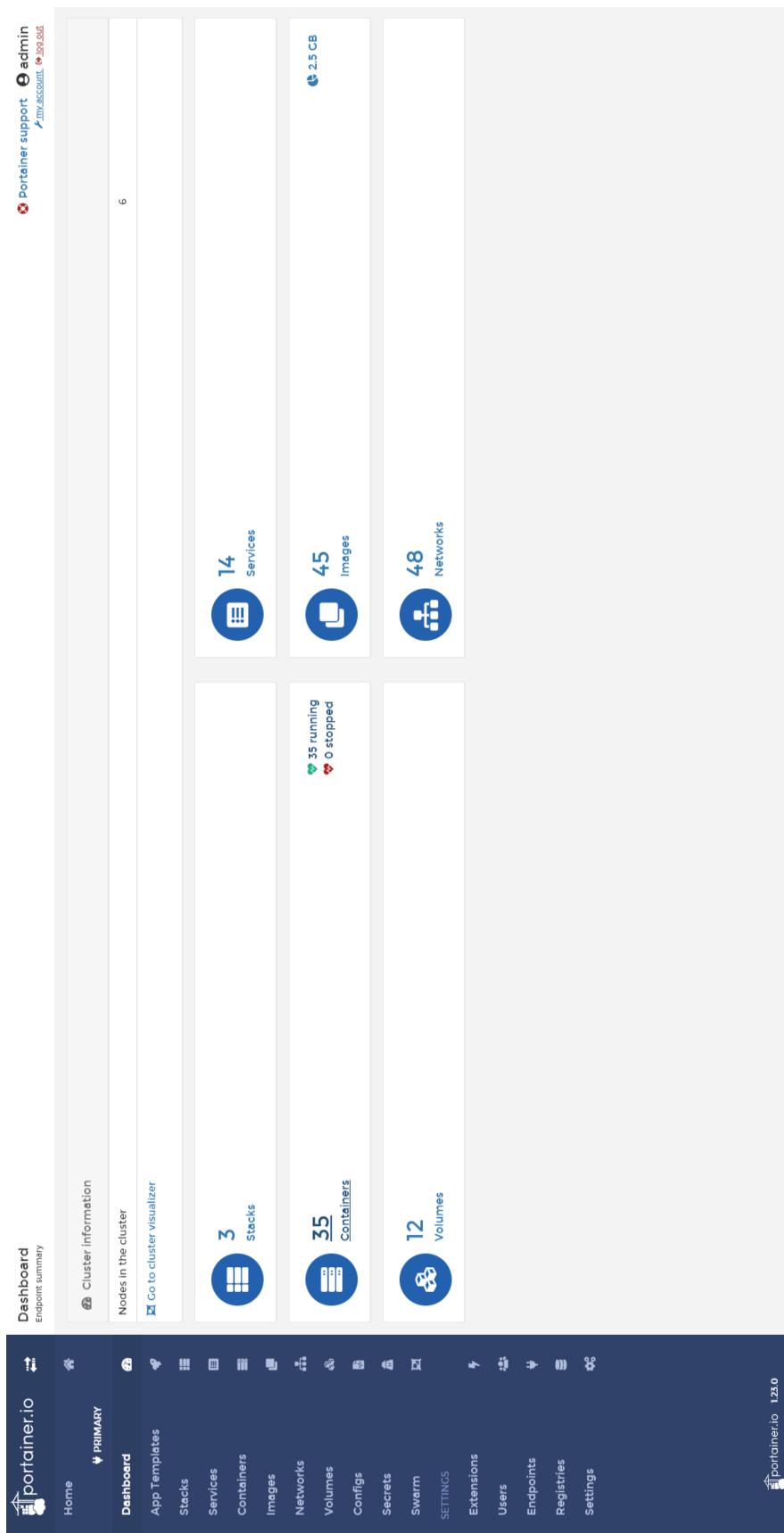


FIGURE 5.5 – Dashboard Portainer

## 5.2 Fonctionnement

On remarque sur le schéma de l'implémentation, figure 5.1, qu'il y a une partie de l'architecture exposée à internet et une autre partie qui est privée, et n'expose aucun port sur le réseau public et elle n'est accessible que par le réseau virtuel de docker.

Néanmoins, certains containers de la partie privée sont exposés par Traefik, ce qui nous permet de gérer l'accès à ces containers et rajouter des middlewares pour l'authentification par exemple. Le microservice s'occupant de l'ingestion des données NB-IoT est directement exposé à Internet, car Traefik ne peut pas gérer le protocole UDP.

Le microservice s'occupant de l'ingestion des données Sigfox est exposé via une API REST qui a un point d'entrée dans la table de Traefik. Grafana, Prometheus et Portainer sont aussi exposés via des points d'entrée, mais un middleware d'authentification a été rajouté pour sécuriser les plateformes et y restreindre l'accès aux personnes non autorisées.

Pour ce qui est de l'acheminement des données, il reste assez similaire pour chaque protocole.

### SIGFOX

1. Les capteurs sont configurés pour envoyer des données aux gateways Sigfox les plus proches à un intervalle d'une heure.
2. Le gateway reçoit les paquets et reconnaît les capteurs par leur identifiant. Il génère un JSON avec les données et envoie une requête HTTP vers l'URL indiquée.
3. La requête est transmise par le répartiteur de charge vers une des machines disponibles.
4. Traefik récupère la requête HTTP et la transmet vers un conteneur du microservice d'ingestion de donnée Sigfox.
5. La requête est validée par le microservice (Sigfox Raw data ingestion) et les données utiles sont extraites et envoyées dans un message sur un topic *ProcessRawDataSigfox* du bus de communication.
6. Le microservice (Sigfox Raw data processing) abonné au topic *ProcessRawDataSigfox* récupère le message, adapte les données au bon format pour être stocké dans la base de données. Ensuite il génère un nouveau message contenant l'ID créé dans la base de données qu'il publie sur le topic *ProcessPulseDataSigfox*.
7. Le microservice (Sigfox Pulse data processing) abonné au topic *ProcessPulseDataSigfox* récupère le message, adapte les données au bon format pour être stocké dans la base de données.

## NB-IoT

1. Les capteurs sont configurés pour envoyer des données en UDP à un intervalle de une heure vers un nom de domaine donné.
2. La requête est transmise par le répartiteur de charge vers une des machines disponibles.
3. La requête est validée par le microservice (NB-IoT Raw data ingestion) et les données utiles sont extraites et envoyées dans un message sur un topic *ProcessRawDataNBIoT* du bus de communication.
4. Le microservice (NB-IoT Raw data processing) abonné au topic *ProcessRawDataNBIoT* récupère le message, adapte les données au bon format pour être stocké dans la base de données. Ensuite il génère un nouveau message contenant l'ID créé dans la base de donnée qu'il publie sur le topic *ProcessRawDataNBIoT*.
5. Le microservice (NB-IoT Pulse data processing) abonné au topic *ProcessPulseDataNBIoT* récupère le message, adapte les données au bon format pour être stocké dans la base de données.

## Autres services

Il existe aussi d'autres services autour des principaux cités précédemment, notamment l'agent qui transmet les métriques du bus de communication NATS vers Prometheus.

Un autre service qui s'occupe de récolter les métriques des différents hôtes et de Docker pour aussi les envoyer vers Prometheus.

Et enfin un service qui permet d'envoyer une alerte en cas de problème au sein du cluster.

## 5.3 Performances

Tout comme pour l'architecture chez AWS, la même série de tests de charge a été effectuée.

- Simulation de 180 000 / 360 000 / 3 600 000 objets connectés
- Six machines virtuelles DEV1-S au sein du cluster Swarm - 2 vCPUs et 2 Go de mémoire
- Une machine pour simuler le trafic - 8 vCPUs - 32 Go de mémoire
- Répartition des objets : 50% NB-IoT / 50% SIGFOX
- Trois conteneurs pour le microservice d'ingestion de données
- Trois conteneurs pour le microservice de processing de données

TABLE 5.1 – Performances du cluster Docker Swarm en fonction des tests de charges

Objets	<b>180000</b>	<b>360000</b>	<b>3600000</b>	<b>3600000 v2</b>
CPU Usage	35.65%	55.40%	140%	75%
Memory Usage	28%	33%	42%	15%
Messages délivrés	180 000	360 000	1 758 356	3 600 000

Pour le troisième test de charge, le nombre de machines et de conteneurs n'étaient pas suffisant pour gérer la charge. Ajouter des machines workers similaires au cluster ne règle pas le problème, car toutes les requêtes sont distribuées par les machines managers et il se fait que les machines de la famille DEV1 sont limitées en terme de performance par Scaleway. En utilisant des machines de niveau supérieur pour les managers du moins tel que les DEV1-L, 4 vCPUs et 8 Go de mémoire, règle le problème et toutes les requêtes sont effectuées avec succès. La dernière colonne correspond à un test de charge effectué sur six machines virtuelles DEV1-L.

## 5.4 Bilan

Cette section fait le bilan de l'architecture microservices chez Scaleway sur base des objectifs définis précédemment au chapitre 1.1.

### Disponibilité

La disponibilité de l'architecture est évaluée sur plusieurs parties différentes qui pourraient être des points de défaillance uniques dans celle-ci.

Scaleway garantit aussi pour tous ses services un contrat de niveau de service (SLA). Si la disponibilité n'est pas atteinte, des crédits de service sont accordés aux clients.

- Compute Instance (machine virtuelle) : Scaleway garantit pour les machines virtuelles un SLA de 99,9% (AWS, 2019), ce qui correspond à un temps d'arrêt autorisé d'environ 44 minutes par mois. Scaleway ne propose actuellement pas de zone de disponibilité. Il faut donc déployer les instances du cluster sur plusieurs régions différentes.

CenturyLink aussi propose un SLA pour son répartiteur de charge qui est de 99,99%.

Pour la disponibilité des différents services existants au sein du cluster, Docker Swarm se charge automatiquement de leur déploiement. En cas de terminaison imprévue d'une tâche d'un service, Docker swarm déploie une nouvelle tâche sur un des machines workers disponibles en quelque seconde.

### Mise à l'échelle

Comparé à ECS, cette architecture-ci n'offre pas de mise à échelle automatiquement. Il est possible de l'implémenter soi-même. Pour ce faire il suffit que Prometheus génère une alerte basée sur des règles définies aux préalables qui appelle un service donné. Ce service va se charger de créer une nouvelle machine virtuelle, l'ajouter au sein du cluster et déployer les nouveaux services sur la machine. Mais tout comme pour l'architecture AWS, cette mise à l'échelle des instances prend plusieurs minutes. C'est pourquoi les microservices doivent être surprovisionnés pour servir toutes les requêtes.

On remarque aussi suite au test de charge qu'à partir d'un certain nombre d'appareils connectés, il faut migrer vers une nouvelle catégorie de machines virtuelles. Comparé encore une fois à AWS, EC2 donne la possibilité d'améliorer la configuration d'une machine sans avoir à en acheter une nouvelle et déplacer son application. Sur Scaleway ce n'est pas possible et la migration peut être résumée en quelques étapes.

1. Acheter les nouvelles machines virtuelles.
2. Ajouter les machines virtuelles au cluster Docker Swarm (managers/workers).
3. Supprimer un noeud worker à la fois afin que les microservices se propagent sur les nouvelles machines.
4. Nommer un des nouveaux noeuds manager comme "Leader" et supprimer les anciens noeuds.

## Hétérogénéité et flexibilité

L'ajout d'un nouveau type d'appareil connecté est simple grâce à la modularité de l'architecture.

Peu importe le protocole de communication supporté par l'appareil connecté, il faut créer un nouveau microservice pour l'ingestion de ses données. Pour la partie processing des données, créer un nouveau microservice pour faire son traitement.

## Optimisation des ressources et des coûts

La quantité de ressources nécessaire à la mise en place de cette architecture est élevée. Le travail effectué peut être illustré en quelques chiffres de la manière suivante :

- 6 machines virtuelles
- 10 microservices
- 35 containers d'applications

C'est un ensemble d'assets à déployer, configurer, monitorer et mettre à jour. Et tout comme pour la partie microservice d'AWS, les microservices doivent être construits comme conteneur Docker et déployés sur un registre d'images. Ensuite seulement les microservices peuvent être mis à jour.

## 5.5 Estimation des coûts

L'estimation des coûts est effectuée en fonction des tests de charge effectués précédemment.

TABLE 5.2 – Estimation des coûts pour l'architecture de microservices chez Scaleway

Objets (en €)	1 000	10 000	180 000	360 000	3 600 000
<b>DEV1-S Instance x 6 (en €)</b>	18.00	18.00	18.00	18.00	0.00
<b>Load balancer (en €)</b>	20.00	20.00	20.00	20.00	20.00
<b>DEV1-L Instance x 6 (en €)</b>					96.00
<b>Total par mois (en €)</b>	38.00	38.00	38.00	38.00	116.00
<b>Coût par mois et par objet (en €)</b>	0.0380	0.0038	0.0021	0.0001	0.00003

Pour cette architecture, il est difficile d'optimiser le nombre de machines ainsi que leurs caractéristiques, car Scaleway limite les performances des machines qu'ils proposent en fonction du prix et de la catégorie. De ce fait malgré que théoriquement les instances DEV1-S peuvent tenir la charge, nous sommes obligés de passer à des instances de catégorie supérieure.

Donc comparés à AWS, nous avons des coûts fortement réduits, mais les serveurs ne sont pas utilisés à plein rendement.

De plus, il est possible de réduire le nombre de microservices à gérer en externalisant le monitoring du cluster. Si un service de monitoring externe (MaaS) tel que Datadog est utilisé, l'ensemble des microservices qui récoltent et stockent les métriques est supprimé et il suffit simplement d'ajouter le petit logiciel fourni par Datadog. Évidemment les coûts mensuels augmentent, mais le nombre de microservices à gérer diminue.

TABLE 5.3 – Estimation des coûts pour l'architecture de microservices chez Scaleway avec monitoring gérée par un service tier

Objets	1 000	10 000	180 000	360 000	3 600 000
<b>DEV1-S Instance x 6 (en €)</b>	18.00	18.00	18.00	18.00	0.00
<b>Load balancer (en €)</b>	20.00	20.00	20.00	20.00	20.00
<b>DEV1-L Instance x 6 (en €)</b>					96.00
<b>Datadog 6 Host (en €)</b>	90.00	90.00	90.00	90.00	90.00
<b>Total par mois (en €)</b>	128.00	128.00	128.00	128.00	206.00
<b>Coût par mois et par objet (en €)</b>	0.1280	0.0128	0.0071	0.0004	0.00006

# Conclusion

*”Part of the journey is the end”*  
Tony Stark

---

6.1	Retour sur les objectifs . . . . .	62
6.2	Réflexion . . . . .	62
6.3	Tests sur d'autres fournisseurs de cloud . . . . .	62
6.4	Serverless vs Microservices . . . . .	63
6.5	Conclusion . . . . .	63

---

## 6.1 Retour sur les objectifs

Les objectifs fixés pour cette thèse ont bien été atteints avec succès et les deux architectures ont bien été implémentées et testées.

### Objectifs atteints :

- ✓ Études et choix de deux architectures.
- ✓ Implémentation architecture serverless chez AWS.
- ✓ Implémentation architecture microservice chez Scaleway.
- ✓ Test de performance architecture serverless chez AWS.
- ✓ Test de performance architecture microservice chez Scaleway.
- ✓ Estimation de coûts architecture serverless chez AWS.
- ✓ Estimation de coûts architecture microservice chez Scaleway.

### Objectifs secondaires :

- ✓ Automatisation du provisionnement de l'infrastructure.
- ✓ Automatisation de la création des images Docker de chaque microservice.

## 6.2 Réflexion

Ce document se concentre sur la comparaison du coût de développement et de déploiement d'une même application en utilisant deux modèles d'architecture, dans le but d'identifier comment différentes architectures peuvent affecter les coûts d'infrastructure pour l'exécution et la mise à l'échelle d'une application dans le cloud. Pour estimer les coûts de fonctionnement de chaque architecture, nous avons expérimenté différents tests de performance. Nous avons aussi défini le coût par appareil connecté pour pouvoir comparer le coût des différentes architectures.

## 6.3 Tests sur d'autres fournisseurs de cloud

Bien que la thèse soit axée sur la comparaison des coûts sur AWS et Scaleway, le processus et l'architecture utilisés peuvent être reproduits sur d'autres fournisseurs cloud. Les architectures, les expériences, les mesures et les défis pourraient servir de guide pour tester les mêmes architectures sur d'autres fournisseurs cloud. Par exemple, les deux architectures présentées dans ce document, ainsi que les tests de performance et la comparaison des coûts correspondants, pourraient être mises en œuvre sur d'autres fournisseurs tels que Microsoft Azure ou Google Cloud. Les résultats d'un tel test pourraient faciliter la comparaison des coûts de fonctionnement de chaque architecture chez différents fournisseurs cloud et aider à déterminer les avantages et les inconvénients à prendre en compte pour déployer et faire évoluer chaque architecture. Une telle comparaison exige évidemment que les architectures présentées dans ce document soient adaptées en fonction des exigences, du cadre de développement et des restrictions de chaque fournisseur.

## 6.4 Serverless vs Microservices

Les deux architectures répondent aux exigences, mais ils se déparent au niveau des coûts qu'il faut nuancer. En effet, comme vu plus haut dans la partie inconvénients et améliorations possibles, pour l'architecture serverless, il est possible d'optimiser l'implémentation pour réduire encore plus les coûts. Par ailleurs, le focus des développeurs doit être mis plutôt sur l'optimisation de l'application business que sur l'infrastructure et le monitoring.

Si nous suivons aveuglément et uniquement les estimations de coût présentées plus tôt, l'architecture microservice déployée sur Scaleway remporte la manche. Mais le focus a été mis sur les coûts de l'infrastructure. Cependant il y a un coût sous-jacent non négligeable à tout cela qui est le coût opérationnel et humain. Donc si nous prenons juste le coût de l'infrastructure, l'architecture sur Scaleway a coût beaucoup moins cher, mais pour gérer cette infrastructure et la maintenir en condition opérationnelle (mise à jour des services, sécurité, monitoring) nous devons rajouter de nouvelles ressources humaines qui devront se focaliser exclusivement à cela par conséquent le coût réel de cette architecture peut drastiquement augmenter en fonction du nombre de personnes nécessaires à ce maintien. Ce coût n'a pas été mis dans les estimations de coût, car il est difficile à estimer. Pour AWS, il faut aussi attribuer du temps à ce maintien opérationnel, mais celui-ci est très minime par la nature serverless de l'architecture et l'écosystème d'AWS.

En résumé, lorsque l'on compare les approches serverless et les microservices, la différence la plus évidente est le développement supplémentaire et les frais généraux d'exploitation que les microservices nécessitent, notamment :

- Installation et soutien du système d'exploitation
- Maintenance et support (par exemple, mises à jour du système d'exploitation, correctifs de sécurité)
- Surveillance du système d'exploitation
- Déploiement et configuration de l'application
- Gestion de l'infrastructure

Une approche serverless offre l'avantage immédiat d'éviter toutes ces complexités et de permettre aux développeurs de se concentrer sur le problème à résoudre.

Pour réussir avec l'un ou l'autre, il faut comprendre les exigences opérationnelles et les attentes des intervenants à l'égard du système que nous concevons.

## 6.5 Conclusion

Ce travail a permis de correctement conceptualiser et implémenter de manière technique, deux architectures pour l'ingestion de données IoT au sein de la plateforme Shayp. Cette expérience a été extrêmement enrichissante car elle m'a permis de découvrir un nouveau domaine technique et de développer mes compétences personnelles.

# A

## Annexes

---

A.1 Infrastructure as Code . . . . .	65
A.2 Design complet architecture AWS . . . . .	69
A.3 Design complet architecture Scaleway . . . . .	71
A.4 Dashboards Portainer . . . . .	72
A.5 Dashboards Grafana . . . . .	76

---

## A.1 Infrastructure as Code

Infrastructure as Code (IaC) utilise un langage de codage descriptif de haut niveau pour automatiser l'approvisionnement de l'infrastructure informatique. Cette automatisation élimine la nécessité pour les développeurs de provisionner et de gérer manuellement les serveurs, les systèmes d'exploitation, les connexions aux bases de données, le stockage et les autres éléments de l'infrastructure chaque fois qu'ils souhaitent développer, tester ou déployer une application logicielle.

À une époque où il n'est pas rare qu'une entreprise déploie des centaines d'applications en production chaque jour, et où l'infrastructure est constamment mise à niveau et mise à l'échelle en réponse aux demandes des développeurs et des utilisateurs, il est essentiel pour une entreprise d'automatiser l'infrastructure afin de contrôler les coûts, de réduire les risques et de répondre rapidement aux nouvelles opportunités commerciales et aux menaces concurrentielles. IaC rend cette automatisation possible.

### A.1.1 Les outils

Bien que de nombreux outils IaC open-source soient disponibles, les outils les plus couramment adoptés et utilisés à travers cette thèse sont Ansible et Terraform :

#### Ansible

Ansible est un projet communautaire open source sponsorisé par Red Hat qui est conçu pour aider les organisations à automatiser le provisionnement, la gestion de configuration et le déploiement d'applications. Ansible permet de créer des "playbooks" (écrits dans le langage de configuration YAML) pour spécifier l'état souhaité pour l'infrastructure et ensuite effectue le provisionnement. Ansible est un choix populaire pour automatiser le provisionnement des conteneurs Docker et des déploiements Kubernetes.

#### Terraform

Terraform est un autre outil de provisionnement déclaratif et d'orchestration d'infrastructure qui permet aux ingénieurs d'automatiser le provisionnement de tous les aspects de leur infrastructure d'entreprise dans le cloud et on-premise.

Terraform fonctionne avec tous les principaux fournisseurs de cloud computing et permet d'automatiser l'accumulation de ressources sur plusieurs fournisseurs en parallèle, quel que soit l'endroit où se trouvent les serveurs physiques, les serveurs DNS ou les bases de données. Il peut également fournir des applications écrites dans n'importe quelle langue.

Contrairement à Ansible, Terraform n'offre pas de capacités de gestion de configuration, mais il travaille main dans la main avec des outils de gestion de configuration (par exemple, Cloud Formation) pour provisionner automatiquement l'infrastructure dans l'état décrit par les fichiers de configuration et pour modifier automatiquement le provisionnement des mises à jour lorsque nécessaire en réponse aux changements de configuration.

### A.1.2 Utilisation au sein du projet

#### Ansible

Ansible est utilisé pour la gestion de configuration et le déploiement du cluster Docker Swarm dans l'architecture microservices chez Scaleway, figure A.2.

Il a pour objectif :

- Mettre à jour les librairies sur la machine hôte
- Installer Docker et créer le cluster
- Ajouter ou supprimer un noeud du cluster
- Déployer les différents microservices sur le cluster

#### Terraform

Terraform est utilisé pour le provisionnement de l'infrastructure sur Scaleway et AWS.

Il crée les éléments suivants :

- Les machines virtuelles sur Scaleway
- Les machines virtuelles sur EC2
- Le cluster ECS
- Les queues SQS
- Les VPC
- Le registre de container (ECR)
- Les différents IAM pour les autorisations
- Le répartiteur de charge (NLB)

Les Lambdas et l'API Gateway est géré par Serverless Framework qui simplifie le déploiement d'application Serverless sur AWS.

```
---
```

```
- hosts: shayp-manager-0
  remote_user: root
  gather_facts: True
  tasks:
    - name: init swarm on primary master
      docker_swarm:
        state: present
        advertise_addr: "{{ansible_default_ipv4.address}}"
      register: swarm_init_result

- hosts: swarm-secondary-managers
  remote_user: root
  tasks:
    - name: add secondary managers to swarm
      docker_swarm:
        state: join
        advertise_addr: "{{ansible_default_ipv4.address}}"
        remote_addrs: [ "{{hostvars['shayp-manager-0'].ansible_default_ipv4.address}}" ]
        join_token: "{{hostvars['shayp-manager-0'].swarm_init_result.swarm_facts.JoinTokens.Manager}}"

- hosts: swarm-workers
  remote_user: root
  tasks:
    - name: add workers to swarm
      docker_swarm:
        state: join
        advertise_addr: "{{ansible_default_ipv4.address}}"
        remote_addrs: [ "{{hostvars['shayp-manager-0'].ansible_default_ipv4.address}}" ]
        join_token: "{{hostvars['shayp-manager-0'].swarm_init_result.swarm_facts.JoinTokens.Worker}}"

- hosts: all
  remote_user: root
  gather_facts: no
  tasks:
    - name: Log into private registry and force re-authorization
      docker_login:
        registry: registry.gitlab.com
        username: scaleway
        password: JT4mCGWwJjLyxGjKfoxN
        reauthorize: yes
```

FIGURE A.1 – Exemple de playbook Ansible pour la création du cluster Docker Swarm

```

terraform {
  backend "s3" {
    bucket = "shayp-terraform-aws"
    key    = "terraform-state/shayp"
    region = "eu-west-1"
  }
}

provider "aws" {
  profile = "default"
  region  = "${var.aws_region}"
  version = "~> 2.34"
}

module "vpc" {
  source = "./vpc"

  aws_region    = "${var.aws_region}"
  project-name = "${var.project-name}"
}

module "ec2" {
  source = "./ec2"

  vpc-id          = "${module.vpc.id}"
  security-group-id = "${module.vpc.security-group-id}"
  subnet-id-1     = "${module.vpc.subnet1-id}"
  subnet-id-2     = "${module.vpc.subnet2-id}"
  subnet-id-3     = "${module.vpc.subnet3-id}"
  ecs-instance-role-name = "${module.iam.ecs-instance-role-name}"
  ecs-instance-profile-name = "${module.iam.ecs-instance-profile-name}"
  ecs-cluster-name      = "${var.ecs-cluster-name}"
  ecs-key-pair-name     = "${var.ecs-key-pair-name}"
}

module "ecs" {
  source = "./ecs"

  ecs-cluster-name      = "${var.ecs-cluster-name}"
  ecs-load-balancer-name = "${module.ec2.ecs-load-balancer-name}"
  ecs-target-group-http-arn = "${module.ec2.ecs-target-group-http-arn}"
  ecs-target-group-udp-arn = "${module.ec2.ecs-target-group-udp-arn}"
  task-definitions       = "${module.container.json}"
  container_name         = "${var.container_name}"
}

```

FIGURE A.2 – Exemple de fichier Terraform pour la création de l'infrastructure AWS

## A.2 Design complet architecture AWS

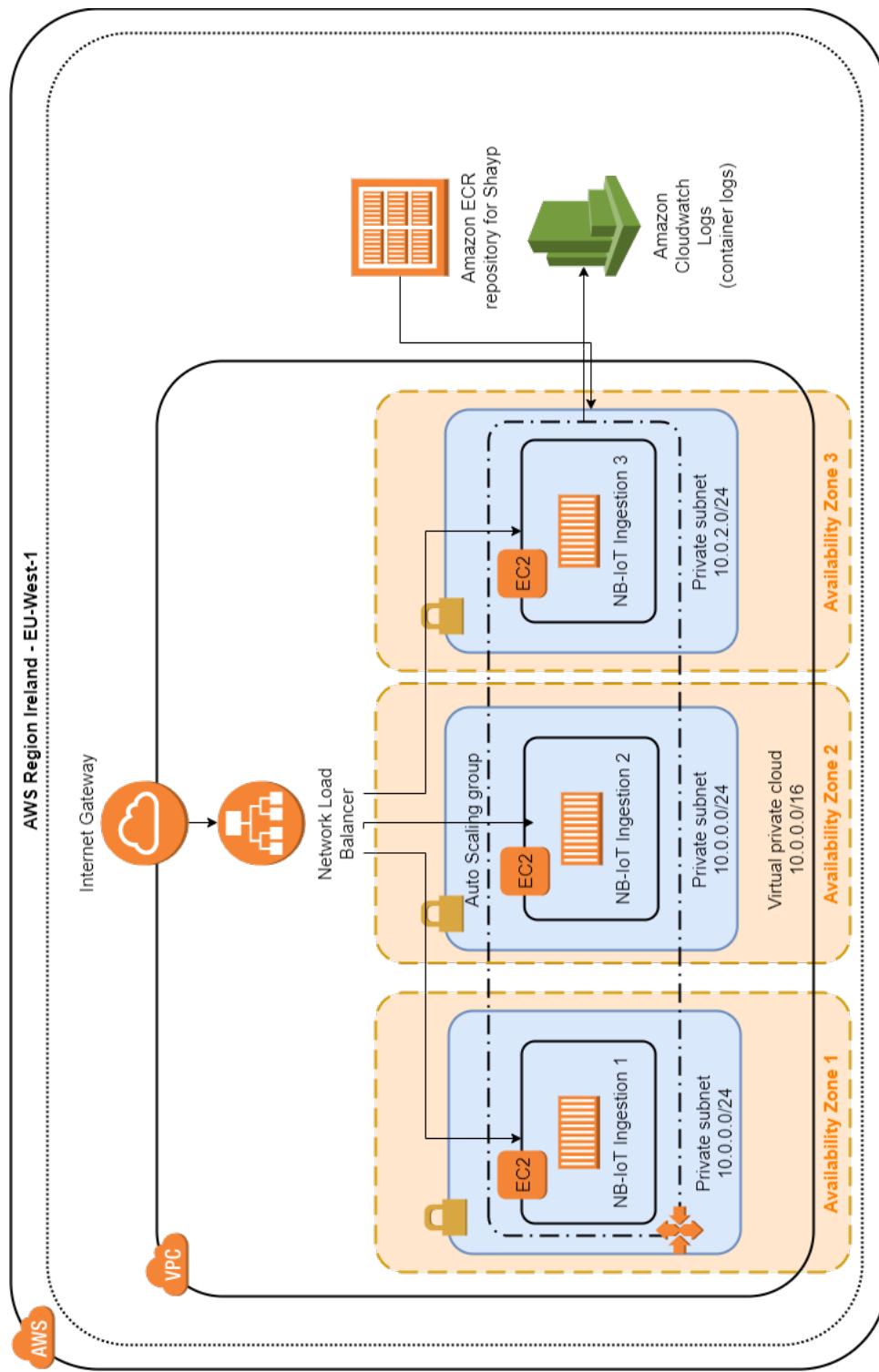


FIGURE A.3 – Design détaillé de l'ingestion de données NB-IoT dans l'architecture chez AWS

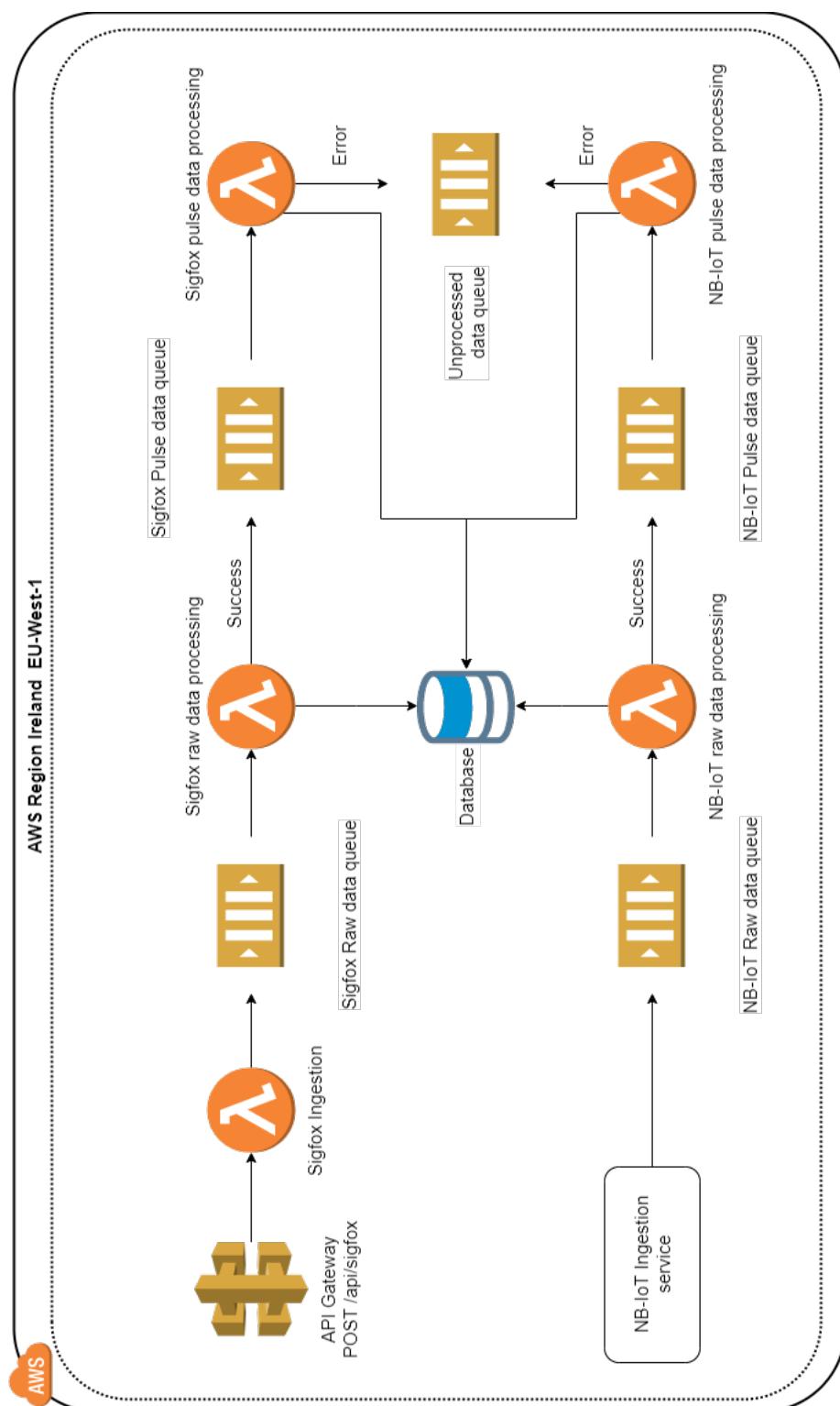


FIGURE A.4 – Design détaillé de la partie serverless dans l'architecture chez AWS

### A.3 Design complet architecture Scaleway

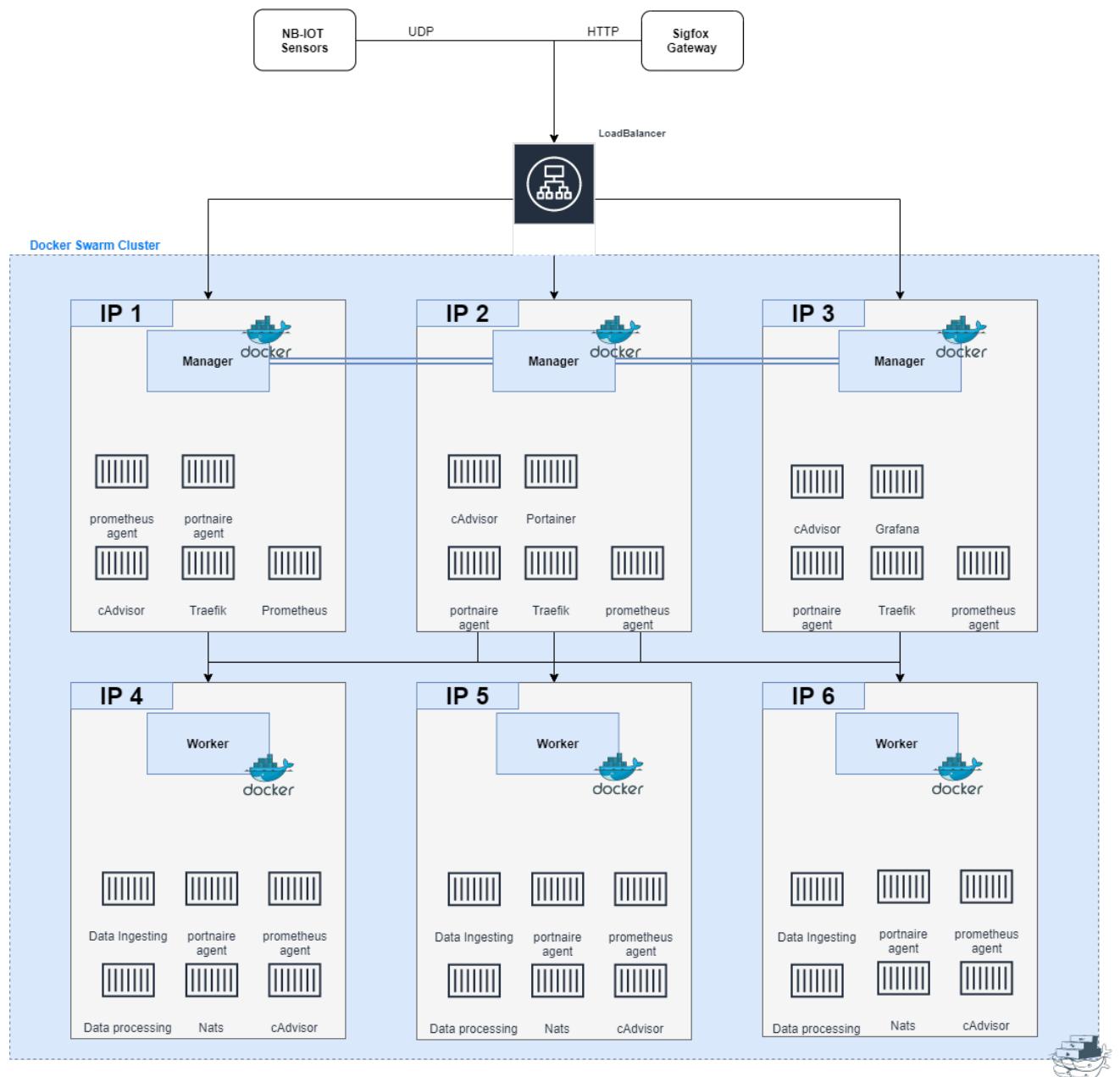


FIGURE A.5 – Vue complète de l'architecture microservice chez Scaleway

## A.4 Dashboards

### Portainer

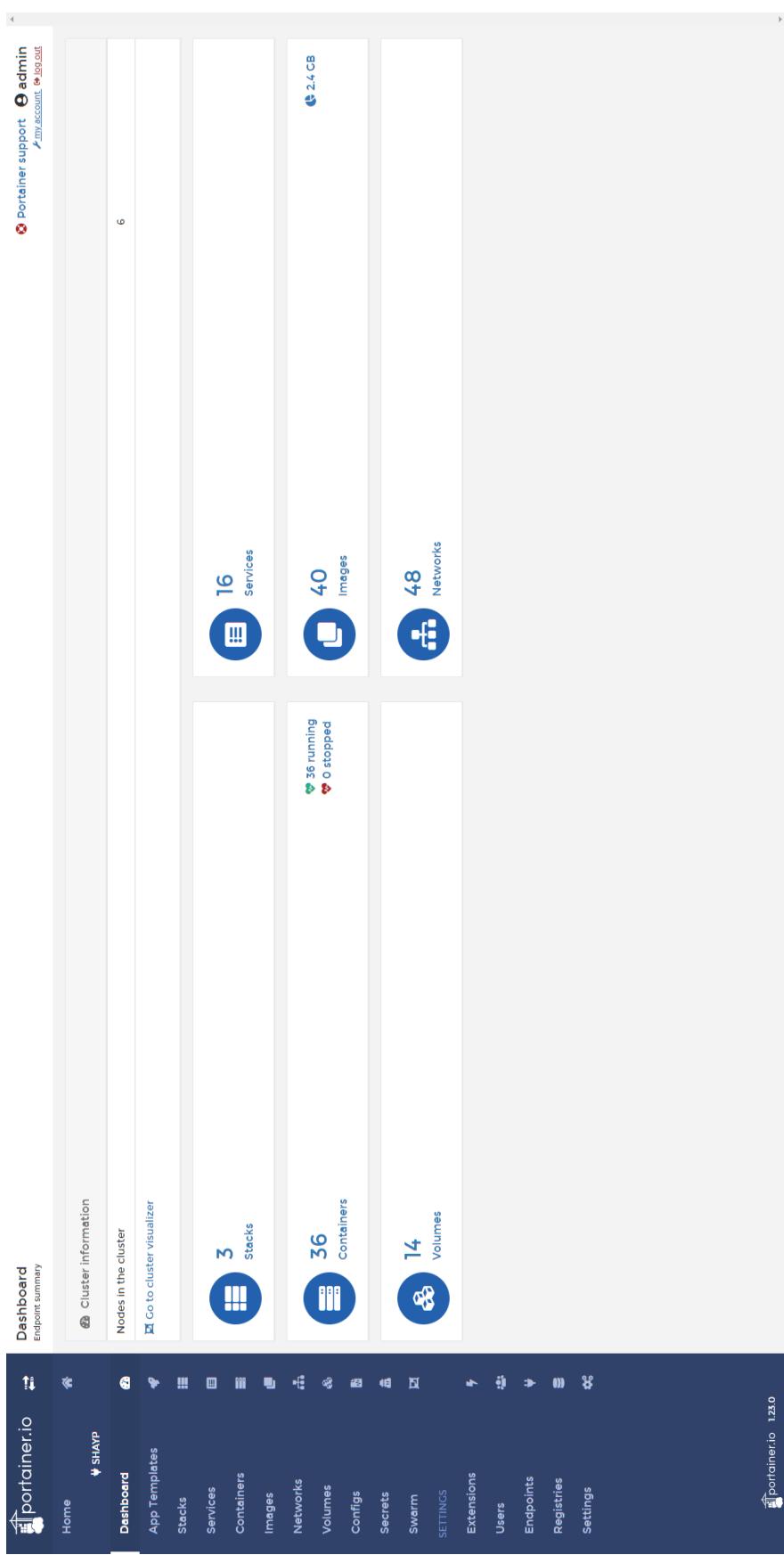


FIGURE A.6 – Portainer - Accueil

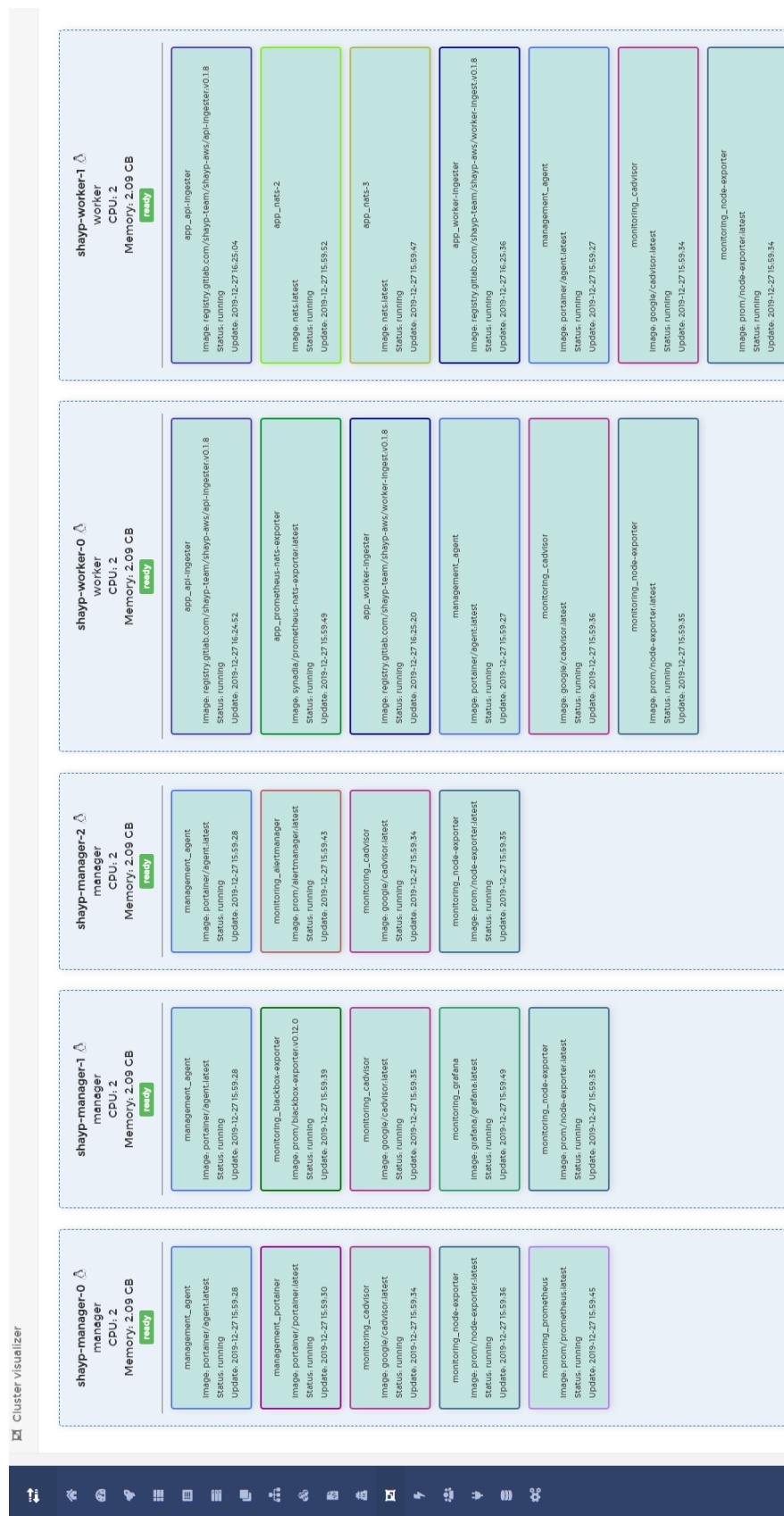


FIGURE A.7 – Portainer - Visualisation du cluster

Stack	Name	Image	Scheduling Mode	Published Ports	Last Update	Ownership
	app-api-ingester	app	replicated 3 / 3 Scale	<a href="#">2204:2204</a> <a href="#">7070:7070</a>	2019-12-27 16:25:09	administrators
Containers	app_nats_1	app	replicated 1 / 1 Scale	<a href="#">1422:4222</a>	2019-12-27 16:24:50	administrators
Images	app_nats_2	app	replicated 1 / 1 Scale	<a href="#">2422:4222</a>	2019-12-27 16:24:40	administrators
Networks	app_nats_3	app	replicated 1 / 1 Scale	<a href="#">3422:4222</a>	2019-12-27 16:24:41	administrators
Volumes	app_prometheus-nats-exporter	app	replicated 1 / 1 Scale	<a href="#">7777:7777</a>	2019-12-27 16:24:43	administrators
Configs	app_worker-ingester	app	replicated 3 / 3 Scale	-	2019-12-27 16:25:41	administrators
Secrets	management_agent	management	global 6 / 6	-	2019-12-27 15:59:24	administrators
Swarm	management_portainer	management	replicated 1 / 1 Scale	<a href="#">9000:9000</a>	2019-12-27 15:59:26	administrators
SETTINGS	monitoring_alertmanager	monitoring	replicated 1 / 1 Scale	<a href="#">9093:9093</a>	2019-12-27 15:59:34	administrators
Extensions	monitoring_blackbox-exporter	monitoring	replicated 1 / 1 Scale	-	2019-12-27 15:59:32	administrators
Users	monitoring_cadvisor	monitoring	global 6 / 6	<a href="#">8080:8080</a>	2019-12-27 15:59:29	administrators
Endpoints	monitoring_grafana	monitoring	replicated 1 / 1 Scale	<a href="#">3000:3000</a>	2019-12-27 15:59:37	administrators
Registries	monitoring_node-exporter	monitoring	global 6 / 6	<a href="#">9100:9100</a>	2019-12-27 15:59:31	administrators
Settings	monitoring_prometheus	monitoring	replicated 1 / 1 Scale	<a href="#">9090:9090</a>	2019-12-27 15:59:35	administrators

Items per page: 25 ▶

FIGURE A.8 – Portainer - Liste des différents services

Name	Role	CPU	Memory	Engine	IP Address	Status	Availability
shayp-manager-0	manager	2	2.1 GB	19.03.5	10.68.40.37	ready	active
shayp-manager-1	manager	2	2.1 GB	19.03.5	10.64.10.65	ready	active
shayp-manager-2	manager	2	2.1 GB	19.03.5	10.64.54.81	ready	active
shayp-worker-0	worker	2	2.1 GB	19.03.5	10.65.32.09	ready	active
shayp-worker-1	worker	2	2.1 GB	19.03.5	10.68.46.31	ready	active
shayp-worker-2	worker	2	2.1 GB	19.03.5	10.64.138.85	ready	active

FIGURE A.9 – Portainer - Overview du cluster.

## A.5 Dashboards

### Grafana



FIGURE A.10 – Grafana - Docker Swarm Service

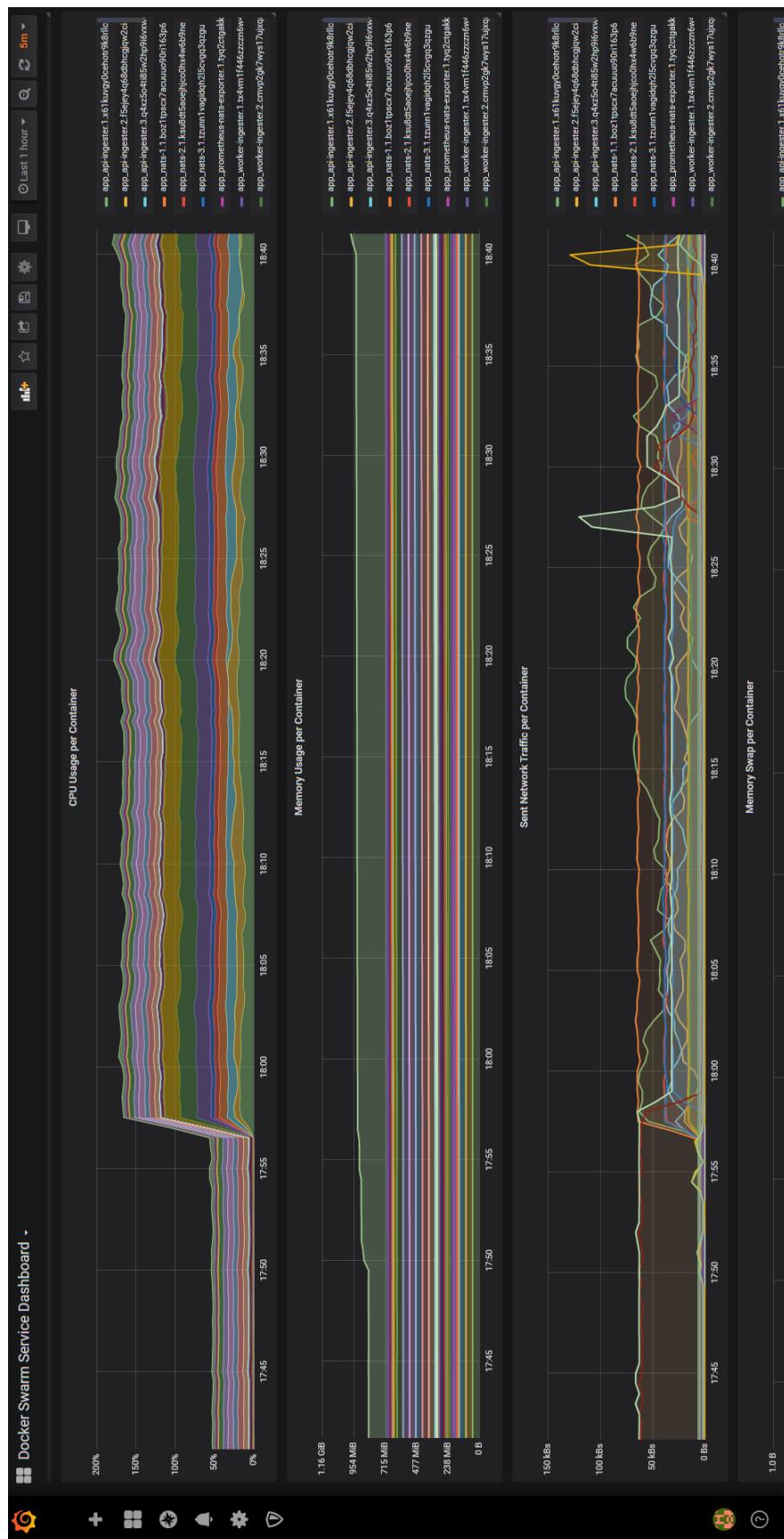


FIGURE A.11 – Grafana - Docker Swarm Service

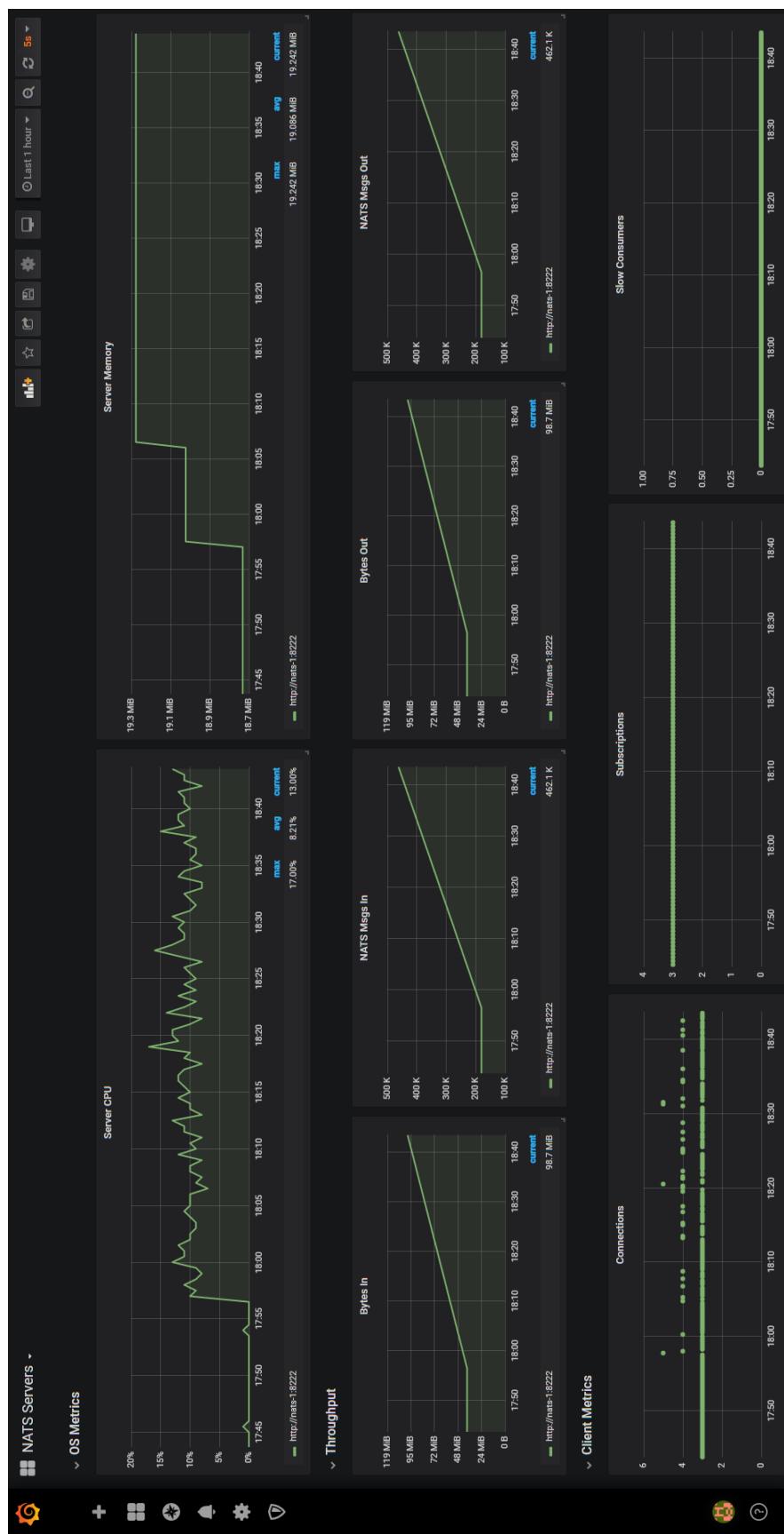


FIGURE A.12 – Grafana - NATS monitoring

# Table des figures

2.1	Les 4 modèles de déploiement du cloud computing selon NIST [8] . . . . .	16
3.1	Architecture monolithique [12] . . . . .	18
3.2	Architecture de microservices [12] . . . . .	20
3.3	Design conceptuel des différents composants de l'architecture . . . . .	25
3.4	Design conceptuel des différents composants de l'architecture avec un nouveau protocol . . . . .	26
4.1	Variante 1 - Ingestion mixte (serverless et microservice) & data processing serverless	28
4.2	Variante 2 - Ingestion microservice & data processing serverlesse . . . . .	29
4.3	Gestion de la mise à échelle des services pour l'ingestion de données UDP . . . . .	36
4.4	Utilisation du CPU par les trois machines virtuelles pour 3M6 objets connectés sur une heure . . . . .	38
4.5	Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services . . . . .	42
4.6	Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services . . . . .	43
4.7	Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services avec Kinesis comme Event Bus . . . . .	46
4.8	Répartition des coûts de l'architecture en fonction du nombre d'objets connectés et services avec Kinesis comme Event Bus et traitement par lot de 10 . . . . .	47
5.1	Architecture microservices basée sur les évènements . . . . .	49
5.2	Déploiement d'un service [21] . . . . .	50
5.3	Architecture Traefik [22] . . . . .	51
5.4	Dashboard Grafana . . . . .	53
5.5	Dashboard Portainer . . . . .	54
A.1	Exemple de playbook Ansible pour la création du cluster Docker Swarm . . . . .	67
A.2	Exemple de fichier Terraform pour la création de l'infrastructure AWS . . . . .	68
A.3	Design détaillé de l'ingestion de données NB-IoT dans l'architecture chez AWS . . . . .	69
A.4	Design détaillé de la partie serverless dans l'architecture chez AWS . . . . .	70
A.5	Vue complète de l'architecture microservice chez Scaleway . . . . .	71
A.6	Portainer - Accueil . . . . .	72
A.7	Portainer - Visualisation du cluster . . . . .	73
A.8	Portainer - Liste des différents services . . . . .	74
A.9	Portainer - Overview du cluster. . . . .	75
A.10	Grafana - Docker Swarm Service . . . . .	76
A.11	Grafana - Docker Swarm Service . . . . .	77
A.12	Grafana - NATS monitoring . . . . .	78

# Liste des tableaux

4.1	Performances de l'architecture microservice - serverless . . . . .	37
4.2	Estimation des coûts pour la variante 1 - Ingestion mixte (serverless et microservice) & data processing serverless . . . . .	41
4.3	Estimation des coûts pour la variante 2 - Ingestion microservice & data processing serverless . . . . .	42
4.4	Estimation des coûts pour la variante 2 - Ingestion microservice & data processing serverless avec Kinesis comme Event Bus . . . . .	46
4.5	Estimation des coûts pour la variante 2 - Ingestion microservice & data processing serverless avec Kinesis et traitement par lots de 10 . . . . .	47
5.1	Performances du cluster Docker Swarm en fonction des tests de charges . . . . .	57
5.2	Estimation des coûts pour l'architecture de microservices chez Scaleway . . . . .	60
5.3	Estimation des coûts pour l'architecture de microservices chez Scaleway avec monitoring gérée par un service tier . . . . .	60

# Acronyms

ALB	Application Load Balancing.
AMI	Amazon Machine Image.
API	Application Programming Interface.
ASG	Auto Scaling Group.
AWS	Amazon Web Service.
BaaS	Backend as a Service.
CaaS	Container as a Service.
CORS	Cross-Origin Resource Sharing.
CPU	Central Processing Unit.
DBaaS	Database as a Service.
EC2	Elastic Cloud Computing.
ECR	Elastic Container Registry.
ECS	Elastic Container Service.
ELB	Elastic Load Balancing.
FaaS	Function as a Service.
HTTP	Hypertext Transfer Protocol.
IaaS	Infrastructure as a Service.
IoT	Internet of Things.
JSON	JavaScript Object Notation.
LBaaS	LoadBalancer as a Service.
LPWAN	Low-Power Wide-Area Network.
M2M	Machine To Machine.
MaaS	Monitoring as a Service.
NB-IoT	Narrow Band Internet Of Things.
NIST	National Institute of Standards and Technology.
NLB	Network Load Balancing.
OSI	Open Systems Interconnection.
PaaS	Platform as a Service.

RAM	Random Access Memory.
SaaS	Software as a Service.
SECaas	Security as a Service.
SLA	Service Level Agreement.
SQL	Structured Query Language.
SQS	Simple Queue Service.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
URL	Uniform Resource Locator.
VM	Virtual Machine.
VPC	Virtual Private Cloud.
XaaS	Everything as a Service.

# Bibliographie

- [1] *Amazon ECR — Amazon Web Services*. URL : <https://aws.amazon.com/fr/ecr/> (visité le 05/01/2020).
- [2] *AWS — Amazon EC2 — Service d'hébergement cloud évolutif*. URL : <https://aws.amazon.com/fr/ec2/> (visité le 05/01/2020).
- [3] *AWS — Amazon VPC — VPN dans le cloud*. URL : <https://aws.amazon.com/fr/vpc/> (visité le 05/01/2020).
- [4] *AWS — Elastic Load Balancing — Répartiteur de charge sur le cloud*. URL : <https://aws.amazon.com/fr/elasticloadbalancing/> (visité le 05/01/2020).
- [5] *AWS — Lambda - Service PaaS de calculs distribués*. URL : <https://aws.amazon.com/fr/lambda/> (visité le 05/01/2020).
- [6] P. CASTRO, V. MUTHUSAMY et A. SLOMINSKI. *The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry*. Rapp. tech. URL : <https://s.cncf.io/>.
- [7] *Conception et gestion des API — Amazon API Gateway*. URL : <https://aws.amazon.com/fr/api-gateway/> (visité le 05/01/2020).
- [8] *Définition de Cloud Computing selon NIST* -. URL : <https://www.hebergeurcloud.com/definition-cloud-computing-selon-nist/> (visité le 10/12/2019).
- [9] *How does it work? Docker! Swarm general architecture — Sebiwi*. URL : <https://sebiwi.github.io/blog/how-does-it-work-docker-1/> (visité le 07/11/2019).
- [10] *InfrastructureAsCode*. URL : <https://martinfowler.com/bliki/InfrastructureAsCode.html> (visité le 13/11/2019).
- [11] *MicroservicePremium*. URL : <https://martinfowler.com/bliki/MicroservicePremium.html> (visité le 14/12/2019).
- [12] *Microservices*. URL : <https://martinfowler.com/articles/microservices.html> (visité le 30/10/2019).
- [13] *Microservices Disadvantages & Advantages — Tiempo Development*. URL : <https://www.tiempodev.com/blog/disadvantages-of-a-microservices-architecture/> (visité le 02/12/2019).
- [14] *Microservices vs Monolith: which architecture is the best choice?* URL : <https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/> (visité le 03/12/2019).
- [15] *MonolithFirst*. URL : <https://martinfowler.com/bliki/MonolithFirst.html> (visité le 19/11/2019).
- [16] *Monolithic vs Microservices Architecture - Why Microservices Win - Tiempo Development*. URL : <https://www.tiempodev.com/blog/monolithic-vs-microservices-architecture/> (visité le 03/12/2019).

- [17] *NIST - The definition of Cloud Computing.* URL : <https://csrc.nist.gov/publications/detail/sp/800-145/final> (visité le 15/11/2019).
- [18] M. SCHUCHMANN. *Designing a cloud architecture for an application with many users.* Rapp. tech. 2018.
- [19] *Serverless Architectures.* URL : <https://martinfowler.com/articles/serverless.html> (visité le 27/10/2019).
- [20] *SQS Service de messagerie — Amazon Simple Queue Service (SQS).* URL : <https://aws.amazon.com/fr/sqs/> (visité le 05/01/2020).
- [21] *Swarm mode overview — Docker Documentation.* URL : <https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/> (visité le 05/01/2020).
- [22] *Traefik - Docker documentation.* URL : <https://docs.traefik.io/providers/docker/> (visité le 15/11/2019).
- [23] M. VILLAMIZAR et al. « Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures ». In : *Service Oriented Computing and Applications* 11 (avr. 2017). DOI : 10.1007/s11761-017-0208-y.