UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Deployment and management of containerized software services at the edge

Diploma Thesis

Isidoros Lamprinos

Supervisor: Christos D. Antonopoulos

October 2023

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Deployment and management of containerized software services at the edge

Diploma Thesis

Isidoros Lamprinos

Supervisor: Christos D. Antonopoulos

October 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ανάπτυξη και διαχείριση υπηρεσιών λογισμικού στο άκρο του δικτύου με τη μορφή containers

Διπλωματική Εργασία

Ισίδωρος Λαμπρινός

Επιβλέπων: Χρήστος Αντωνόπουλος

Οκτώβριος 2023

Approved by the Examination Committee:

Supervisor **Christos D. Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Prof. Christos D. Antonopoulos, for his guidance and support throughout this endeavour. His immense experience and knowledge were indispensable for the completion of this thesis. Finally, I would like to thank my family and friends for their support and love during every step of this six-year journey. I am forever grateful.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Isidoros Lamprinos

Diploma Thesis

Deployment and management of containerized software services at the edge

Isidoros Lamprinos

Abstract

The rapid scaling of cloud computing in recent years has triggered a discussion on the way applications are deployed on the edge of the network. This Thesis investigates the intricate process of deploying and managing containerized software services at the edge of the network, where resource constraints, latency sensitivity and dynamic environments bring to light unique challenges. The objectives of this study are to (i) investigate the technical feasibility of deploying containerized services at the edge using open-source projects like EdgeX Foundry and OpenHorizon, while optimizing resource utilization, and (ii) to investigate the practicality of integrating these projects in order to create a unified methodology of deploying containerized services at the edge. This research, through thorough review of the existing methodologies and strategies and by experimenting on real-world use cases, demonstrates the effectiveness of this integration in optimizing the deployment and orchestration of containerized services at the edge. The insights demonstrated in this Thesis are useful for practitioners and academics considering to leverage the potential of EdgeX Foundry and Open Horizon in delivering and managing containerized software services at the Edge. This study tries to give inspiration for the next generation of Edge computing solutions, encouraging creativity, efficiency and agility in a wide spectrum of cases concerning the deployment of applications on the Edge.

Keywords:

Docker, Containers, Edge Computing, Cloud Edge, OpenHorizon, EdgeX Foundry, IoT

Διπλωματική Εργασία

Ανάπτυξη και διαχείριση υπηρεσιών λογισμικού στο άκρο του δικτύου με τη μορφή containers

Ισίδωρος Λαμπρινός

Περίληψη

Η ραγδαία κλιμάκωση του cloud computing τα τελευταία χρόνια έχει προκαλέσει συζήτηση σχετικά με τον τρόπο που εφαρμογές αναπτύσονται προς άκρο του δικτύου. Αυτή η διπλωματική εργασία εξετάζει την πολύπλοκη διαδικασία υποβολής και διαχείρισης υπηρεσιών λογισμικού στο άκρο του δικτύου, όπου περιορισμοί πόρων, ευαισθησία στην καθυστέρηση και δυναμικά περιβάλλοντα φέρνουν στο φως μοναδικές προκλήσεις. Οι στόχοι αυτής της μελέτης είναι (i) να εξετάσουν την τεχνική εφικτότητα της υποβολής υπηρεσιών στο άκρο του δικτύου χρησιμοποιώντας λογισμικά ανοιχτού κώδικα, όπως το OpenHorizon και το EdgeX Foundry ενώ βελτιστοποιούν τη χρήση πόρων, και (ii) να εξετάσουν τον πρακτικό χαρακτήρα της ένταξης αυτών των έργων με σκοπό τη δημιουργία μιας ενιαίας μεθοδολογίας υποβολής υπηρεσιών στο άκρο του δικτύου. Αυτή η έρευνα, μέσω λεπτομερούς αναθεώρησης των υπαρχουσών μεθοδολογιών και στρατηγικών και μέσω πειραματισμών σε πραγματικές περιπτώσεις χρήσης, δείχνει την αποτελεσματικότητα αυτής της ένταξης στη βελτιστοποίηση της υποβολής και της ενορχήστρωσης των υπηρεσιών λογισμικού στο άκρο του δικτύου. Οι διαδικασίες που παρουσιάζονται σε αυτήν τη εργασία είναι χρήσιμες για επαγγελματίες και ακαδημαϊκούς που σκέπτονται να αξιοποιήσουν τις δυνατότητες του Open Horizon και του EdgeX Foundry για την παράδοση και τη διαχείριση υπηρεσιών λογισμικού μέσω containers στο άκρο του δικτύου. Αυτή η μελέτη προσπαθεί να προσφέρει έμπνευση για την επόμενη γενιά λύσεων στον τομέα της Edge υπολογιστικής, προάγοντας τη δημιουργικότητα, την αποτελεσματικότητα και την ευελιξία σε μια ευρεία γκάμα περιπτώσεων που αφορούν την υποβολή εφαρμογών στο άκρο του δικτύου.

Λέξεις-κλειδιά: Docker, Containers, Edge Computing, Cloud Edge, OpenHorizon, EdgeX Foundry, IoT

Table of contents

List of figures

List of tables

Chapter 1

Introduction

In recent years we observe more and more the use of containerized software in different fields of computer science, such as DevOps, Cloud, virtual network functions and many more. One of the platforms that is mainly used to containerize applications and manage them is Docker. The reason this is happening is that Docker containers make it easy to put new versions of software, with new business features, into production quickly, and to quickly roll back to a previous version if the developer needs to. Containers are easy-to-deploy software packages and containerized applications are easily distributed, making them a natural fit for edge computing solutions.

Containerization is a technology that addresses many of the challenges of operating software systems at the edge. Despite being similar to virtual machines (VMs), containers use the host's kernel and do not virtualize the operating system kernel (typically Linux). Although this method makes containers less portable and less isolated than virtual machines, it does reduce some of the resource overhead associated with virtualization.

1.1 Motivation

The scope of this Thesis is the experimentation with software services and their deployment at the edge. The tools that are going to be used are Open Horizon and EdgeX Foundry, both of which are under the Linux Foundation umbrella. Every day, we utilize edge computing devices like smart speakers, wearables and phones, which collect and process data locally while interacting with the real world. If they interact with the Cloud while computing locally, then they can be considered as Cloud edge devices.

Edge computing moves data processing and storage closer to the points at which people, places and things generate their data. The task of installing the appropriate software and machine learning tools on the appropriate computing platforms and keeping those programs active and updated is made simpler by Open Horizon. Additionally, Open Horizon allows for the autonomous control of up to 10,000 edge devices at once, which is a 20-fold increase over conventional methods. One of the goals of edge computing is to utilize the disciplines developed for hybrid cloud computing to support remote operations of edge computing facilities. Open Horizon is designed with that objective in mind.[1]

In order to reduce deployment risks and fully autonomously control the service software lifecycle on edge nodes, the Open Horizon management hub was specifically created for edge node management. The components of the Open Horizon management hub are installed and maintained by a cloud installer. Edge services are created and published by software developers to the management hub. The deployment policies that determine where edge services are deployed are set by administrators. Everything else is handled by Open Horizon.

1.2 Contribution

In this Thesis, we will try to test the effectiveness of OpenHorizon in distributing containerized application, like the EdgeX framework, at the edge of the cloud:

1. First we deployed and executed Open horizon Management and Agent on different devices and validate proper operation. Open Horizon lacks adequate documentation, which can lead to many problems with the deployment of its components. We used the instructions/tutorials available online in the official page and github repository as a starting point.

2. Next we created a guide on how the tools actually worked, what we changes and what kinds of problems we faced along the way. This is, as far as we can tell, the only up to date guide on how to deploy EdgeX using Open Horizon. There had been another one in 2020 when Open Horizon was first introduced as an open source project [2], but the EdgeX project had a major version update and very few of these guidelines are still functioning.

3. After making sure that everything works properly, we experimented on the actual de-

ployment of containerized applications. EdgeX was the main service deployed. It consists of multiple microservices , it is designed to work on Open Horizon and is part of the same project umbrella.

4. Finally we evaluated Open Horizon and its deployment capabilities by performing experiments firstly with docker-compose.yml, which is the default deployment method, and then with Open Horizon, in order to compare them.

1.3 Thesis Organization

The thesis is organized as follows:

1. The 2nd Chapter analyzes the background knowledge needed to understand the third-party tools we are going to use as an essential part of our process, such as Docker and Containers.

2. The 3rd Chapter focuses on the setup and deployment of Open Horizon and EdgeX.

3. The 4th Chapter is going to be the evaluation of Open Horizon through the experiments we contacted in comparison with the docker-compose.

4. The 5th Chapter concludes the work of this thesis, with potential enhancements and future extensions.

Chapter 2

Background

Two important concepts, that are going to play significant role in the understanding of the implementation in this Thesis, are ”container” and ”docker” .

2.1 Container

Containers are standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment. Containers make it easier to create and deliver distributed applications. They are becoming more popular as enterprises migrate to cloud-native development and hybrid multi-cloud setups.[3]

2.1.1 Why use containers?

Container technology offers all the functionality and benefits of VMs,including application isolation, cost-effective scalability and disposability. Some additional advantages are:

1. Lighter weight than VMs; containers do not carry the load of the entire OS and the hypervisor. They only carry the dependencies and OS processes required to run the program. Container sizes are measured in megabytes (vs. GigaBytes for some VMs).

2. Developer productivity is increased since containerized apps may be created once and run anywhere. Moreover, containers are quicker to deploy, provision and restart than virtual machines. In addition to being a better fit for development teams following Agile and DevOps approaches, this makes them perfect for use in continuous integration

and continuous delivery (CI/CD) pipelines.

3. Increased resource efficiency: Compared to virtual machines (VMs), containers allow developers to execute many more copies of a program on the same hardware. This may result in lower cloud costs.

4. Portability: Containers are isolated, self-contained environments that have all the configurations and dependencies required to run an application. Because of this, it is simple to move a containerized program from one environment to another, such as from a developer's laptop to a testing environment or from an on-premises server to a cloud-based platform. It is simpler to build, test, and deploy programs across many environments as a result of this portability.

5. Consistency: Regardless of the host system or the underlying infrastructure's capabilities, containers offer a consistent runtime environment for programs. This lowers the chance of compatibility problems and other issues that might occur when deploying programs across different systems because applications can be built and tested in a consistent environment.

6. Speed: Containers provide quick application deployment and iteration since they are quick to start and stop. This can shorten the time between development and deployment and help businesses adapt to shifting customer demands.

Overall, using containers to design and deploy applications offers a versatile, effective, and portable method. Developers and operations teams may collaborate more effectively, increase the dependability and consistency of systems and quicken the release of software products to market by adopting containers.

2.1.2 Important concepts on Containers

In the following, we explain some fundamental concepts related with containers.

1. Image Registries: Containers are typically created from images, which are pre-built packages that include all the necessary files, libraries, and dependencies needed to run an application. These images can be stored and shared in image registries, such as Docker Hub, AWS ECR or Google Container Registry. Image registries allow the developers to easily distribute and deploy container images across different systems.

2. Orchestration: Container orchestration tools, such as Kubernetes, Docker Swarm, or Apache Mesos, can help the developers manage and scale containers across multiple hosts or clusters. Orchestration tools automate tasks such as deployment, scaling, and resource allocation, making it easier to manage large-scale container deployments.

3. Networking: Containers can communicate with each other through a network, allowing for inter-container communication and service discovery. The developers can configure container networks to provide isolation, security and load balancing for your applications.

4. Security: Container security is an important consideration, as containers can potentially expose vulnerabilities or allow attackers to gain access to your system. Container security best practices include using trusted images, configuring appropriate permissions and access controls and implementing network segmentation and isolation.

5. Compatibility: While containers provide a high degree of portability and consistency, it is important to ensure that your applications are compatible with the container runtime environment. This includes ensuring that all necessary dependencies are included in the container image and that the application is configured to work within the constraints of the container environment.[4]

2.2 Docker

Docker is an open source platform that enables developers to build, deploy, run, update and manage containers. It is a platform that allows the developers to package and run their applications in an isolated, self-contained environment. It is possible for developers to create containers without Docker, by working directly with capabilities built into Linux and other operating systems. But Docker makes containerization faster, easier and safer.[5]

2.2.1 Why use Docker

Docker allows developers access these native containerization capabilities, using simple commands, and automate them through a work-saving application programming interface (API).Docker offers:

1. Improved and seamless container portability: Docker containers run without modification across any desktop, data center and cloud environment. Docker containers can run on any host that supports Docker, whether it is your local machine, a virtual machine, a cloud server or a Kubernetes cluster.

2. Automated container creation: Docker can automatically build a container based on application source code.

3. Container versioning: Docker allows the developers to track different versions of a container image, go back in time to earlier versions, and find out who created it and how. Even just the differences between an existing version and a new version might be uploaded.

4. Reusing containers: Pre-existing containers may be utilized as base images, serving as sort of templates for new containers.

5. Shared container libraries and Collaboration: Thousands of user-contributed containers are available in an open-source registry that developers can use. Docker makes it easy to share and collaborate on development environments and applications. The developers can share their Docker images with their team or community and run the same environment on their own machine.

6. Security: Docker provides several security features to ensure that your applications and containers are secure. These include namespace isolation, resource constraints, image signing and verification, and network security. However, it is important to be aware of potential security risks, such as container breakout attacks and image vulnerabilities, and take appropriate measures to mitigate them.

2.2.2 Docker Components

1. Dockerfile: A Dockerfile is a script that defines the configuration of a Docker image. It specifies the base image, the commands to install dependencies and build the application and any other configurations needed to run the application. By using a Dockerfile, the developers can automate the process of building and deploying Docker images.

2. Docker Hub: Docker Hub is a public registry where the developers can find and share Docker images. It is like a repository of Docker images that anyone can use. The de-

velopers can also create their own private registry to store and distribute their Docker images within their organization.

3. Docker Compose: Docker Compose is a tool for defining and running multi-container Docker applications. It allows the developers to define a YAML file that describes the services and dependencies of their application, and then start and stop all the containers with a single command. Docker Compose is especially useful for complex applications that require multiple containers to work together.

2.3 Additional important concepts

2.3.1 Cloud Computing

Cloud computing is the delivery of computer services via the Internet (”the cloud”) in order to bring faster innovation, adaptable resources and scalable economies, including servers, storage, databases, networking, software, analytics and intelligence. Typically, the developers only pay for the cloud services they actually use, which lowers operational expenses, improves infrastructure management, and enables them to scale as the company’s needs evolve, which are subsequently the main reasons for its popularity.[6] [7]

2.3.2 Computing at the Edge of the Cloud

Edge computing is a distributed information technology (IT) architecture in which client data is processed as close to the original source as possible at the network’s perimeter. Edge computing is a new computing paradigm that refers to a variety of networks and devices located at or near the user. The Edge is about processing data closer to where it is generated, allowing for faster and larger processing rates and volumes, resulting in more actionable answers in real time. But why is it so important?
Businesses may use Edge to bring the digital world into the real world, bringing web data, algorithms and analytics into physical businesses to improve customer experiences, developing technologies that can instruct employees and scenarios in which workers may learn from machines, creating smart settings that prioritize our safety and comfort. These instances all have one thing in common: Edge computing, which allows businesses to operate applications with the most strict dependability, real-time and data requirements immediately on-site. Fi-

nally, this enables organizations to innovate more swiftly, launch new goods and services more quickly, and create new income streams. [8]

2.3.3 The Internet of Things

IoT is not a new concept, but everyone is talking about it, because it intersects with several other key trends in the tech world, from open source and big data to cybersecurity and software-defined networking. Those who use the Internet of Things may live and work smarter and have complete control over their lives. In addition to delivering smart home automation devices, IoT is critical to business. Organizations may use IoT to examine how their systems work in real time, obtaining insights about anything from equipment performance to supply chain and logistics activity. The Internet of Things has a number of advantages for businesses. Certain advantages are unique to certain businesses, while others apply to many other industries. The following are some typical advantages of IoT for businesses[9]:

1. Monitor their overall business processes.
2. Improve the customer experience.
3. Save time and money.
4. Enhance employee productivity.
5. Integrate and adapt business models.
6. Make better business decisions.

Chapter 3

Deployment of Open Horizon

In this chapter we are going to present a more detailed view on Open Horizon and Edgex Projects and an easy way to deploy the Open Horizon Management Hub, Agent and CLI. We used the official Open Horizon instructions [10] as the baseline for the process described in the following sections.

3.1 The Linux Foundation Edge

LF Edge [11] is a new umbrella organization that aims to establish open, inter-operable frameworks for Edge computing independent of hardware, silicon, cloud, or operating system. Launched in January 2019, LF Edge comprises existing Linux Foundation projects Akraino Edge Stack [12] and EdgeX Foundry [13], as well as the new Project EVE from ZEDEDA [14] and Home Edge Project from Samsung Electronics [15]. The tools that were used in this Thesis were developed or currently ”live” under this umbrella. These tools/projects are:

3.1.1 EdgeX Foundry

It is a very adaptable open-source software framework that supports communication between various IoT Edge apps and devices while providing a stable foundation for security. EdgeX translates and transforms sensor data from devices and delivers it to applications over network-based protocols that meet the needs of the clients. It is capable of sending, as well as receiving, data along with usage for the control and actuation of Edge nodes. By offering modular reference services for device-data ingestion, normalization, analysis and sharing

in support of new IoT data services, advanced Edge computing applications, including AI and automation, the open, vendor-neutral platform shortens the time it takes for technology providers and developers to market. EdgeX translates and transforms data arriving from sensors and devices before sending them across network-based protocols to applications in formats and structures that suit users' needs. Moreover, it transfers data from applications to Edge nodes and devices for updates, control, and actuation. The primary services offered by the EdgeX loosely coupled microservices architecture are discussed in Section [13].

3.1.2 Open Horizon

LF Edge Open Horizon is an open source project that provides a platform for managing the life-cycle of distributed Edge applications. It enables organizations to deploy and manage containerized workloads on Edge devices, such as IoT devices or gateways, and includes features such as dynamic application deployment, policy-based management, and machine learning inference at the Edge.

Some key features and components of Open Horizon include:

Node: The Edge devices that are managed by Open Horizon are referred to as nodes. Nodes can be any device capable of running a Docker container, such as a Raspberry Pi, a smart camera or an industrial gateway.

Pattern: A pattern is a set of instructions that defines how to deploy and manage a set of containers on a node. Patterns can be defined using a JSON file and can include information such as which containers to deploy, how to configure them, and when to start and stop them.

Policies: are rules that govern how containers are deployed to Edge devices. They define which devices are eligible for deployment, what containers should be deployed to each device, and when and how the containers should be deployed. Policies are defined using a policy language that is specific to Open Horizon.

Machine Learning Inference: Open Horizon includes a machine learning inference capability that enables Edge devices to run machine learning models locally. This can reduce latency and bandwidth requirements, and enable real-time decision making at the Edge.

Community: Open Horizon is part of the LF Edge organization, which is an umbrella organization for open source projects focused on Edge computing. The LF Edge community includes developers, vendors, and users who collaborate on developing and promoting open standards and best practices for Edge computing.[1]

3.2 How does Open Horizon Work

Open Horizon intends to facilitate autonomous administration of Edge workloads, such as machine learning models, on a variety of Edge devices, including Raspberry Pi devices, IoT gateways and other sorts.
This is a high-level overview of how Open Horizon works:

1. **Registration of Edge Devices**: In order to use Open Horizon, Edge devices must first be registered with the platform. Each device must have an Open Horizon agent installed on it so that it can communicate with the Open Horizon management center and accept workloads.

2. **Defining Workloads and Policies**: Users can specify workloads and policies for how those workloads should be allocated across the Edge devices once devices have been registered. Workloads may consist of executable programs, containers or machine learning models that can execute on the device. Our primary focus is going to be the deployment of containers.

3. **Execution of Workloads**: The Open Horizon agent on each Edge device is in charge of carrying out the workload when it has been distributed to those devices. The agent keeps track of the device's condition and updates the management hub on the workload's development and status.

4. **Edge Node Governance**: For Edge nodes, Open Horizon offers a governance paradigm. Users can create policies that limit the execution of workloads on particular devices, ensuring that workloads are only run on devices that adhere to predetermined standards.

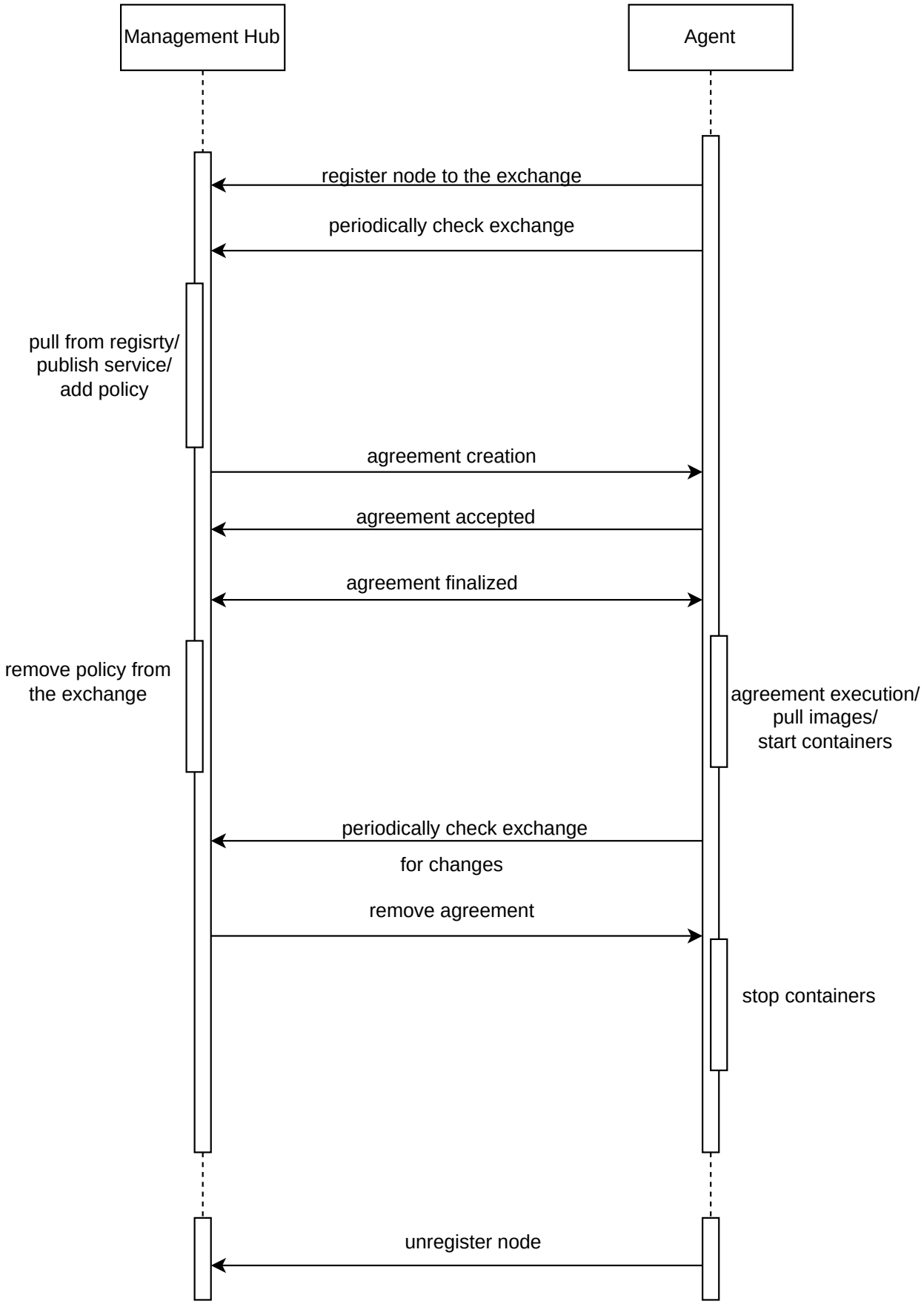

Figure 3.1: Open Horizon Sequence Diagram

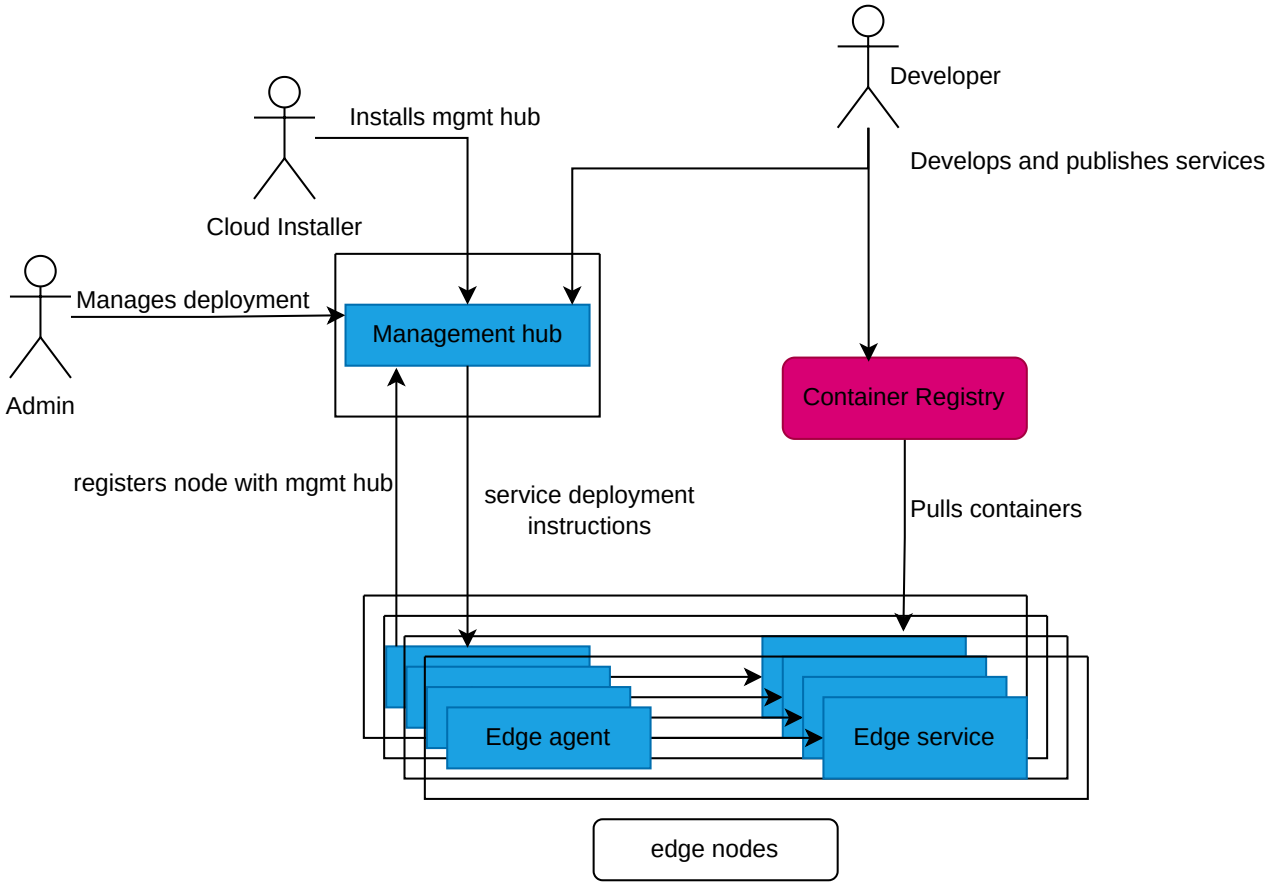

Figure 3.2: Open Horizon Workflow

3.3 How does EdgeX Work

As our primary focus is going to be the deployment of EdgeX, lets see a brief overview of how it works:

3.3.1 Overview

1. **Device Integration:** The EdgeX platform includes an SDK (Software Development Kit) that offers a collection of APIs (Application Programming Interfaces) for integrating various Edge devices, sensors, and services. With the help of this SDK, developers may quickly link their products to the EdgeX platform.

2. **Data Collection**: After devices are linked, the EdgeX platform uses a common data model to collect data from these devices. The data is translated into a standard format that may be used by a variety of applications after being normalized.

3. **Processing of data**: EdgeX offers a flexible rules engine that enables programmers to create rules for data processing, filtering and aggregation. Based on particular data patterns or thresholds, this rules engine can be used to start events or trigger actions.

4. **Data Export**: EdgeX offers a collection of APIs for transferring data to other cloud services, databases or other software programs. This makes it simple for developers to integrate their Edge data with other platforms and programs.

3.4 Pre-requisites

In order to run the All-in-One [10] Horizon Management Hub, Agent and CLI script [16] we are going to use Ubuntu Server 20.04 on Virtualbox [17]. As this tutorial is old, we also need to mention that the script works not only on Ubuntu 18.04, but on 20.04 and 22.04 as well. The Management Hub device running on VM needs:

	Memory(GB RAM)	Storage (GB)	CPU	other Pre-requisites
Management Hub	4	20	1 virtual CPU	root access
Agent	1	10	1 virtual CPU	root access

Table 3.1: Management and Agent Pre-requisites

3.5 Setting up the Management Hub device

Some important changes were made in the deployment of the Management Hub after carefully reading the tutorial and through trial and error. Although some are mentioned later in the tutorial and, as our goal is to connect more than one Edge devices, we need to follow some steps earlier than mentioned on the official documentation. Changes were also made in order ensure to its effective implementation.

- First of all, in order to add more devices, the management hub needs to be listening on an IP address that is reachable by the Edge devices and by using HTTPS. So we need to reconfigure it. Before running the deployment script:

```
        ./deploy-mgmt-hub.sh

```

we need to export some values:

```
export HZN_LISTEN_IP=`ifconfig | egrep 'inet ' | sed 's/addr://' |
   awk '{ print $2 }' | egrep -v '^172.|^10.|^127.' | head -1`
echo "export HZN_LISTEN_IP=${HZN_LISTEN_IP}" >> ~/.bashrc
export HZN_ORG_ID=issy
echo "export HZN_ORG_ID=issy" >> ~/.bashrc
export HZN_EXCHANGE_URL=https://${HZN_LISTEN_IP}:3090/v1/
echo "export HZN_EXCHANGE_URL=https://${HZN_LISTEN_IP}:3090/v1/" >>
   ~/.bashrc
export HZN_FSS_CSSURL=https://${HZN_LISTEN_IP}:9443
echo "export HZN_FSS_CSSURL=https://${HZN_LISTEN_IP}:9443" >> ~/.
   bashrc
export HZN_TRANSPORT=https
echo "export HZN_TRANSPORT=${HZN_TRANSPORT}" >> ~/.bashrc
```

What this script implements is exporting the IP address of the device, setting the parameter HZN_ORG_ID and using the IP in order to set more useful values, like the url of the HORIZON EXCHANGE. It also exports these values on /.bashrc so we do not have to repeat this step every time we login to the device.

- Secondly, we need to make some changes in the script. Most of the images are pulled from Docker Hub with tag: -latest. This is generally a mistake, as this cannot guarantee that these images are compatible with one another, as was the case at the time we tested it. The error that was raised was a connectivity issue between mongodb image and the css image. So the first modification is in line 197:

```
1    export MONGO_IMAGE_TAG=${MONGO_IMAGE_TAG:-5}
2
```

We also modified the name of the Organization in line 146:

```
1 export EXCHANGE_USER_ORG=${EXCHANGE_USER_ORG:-<value_of_choice>}
2
```

This <value_of_choice> has to be the same value that was exported in the /.bashrc as HZN_ORG_ID . In our case this value was set to ”issy”.

- One more thing to consider is the use of the flags -A -E of the deployment script:
  -A —> Do not install the horizon agent package (It will still install the horizon-cli package). Without this flag, it will install and register the horizon agent (as well as all of the management hub services).
  -E —> Skip loading the horizon example services, policies, and patterns.

3.6 Setting up the Agent device

After setting up the Management Hub we can go ahead and setup the Agent on an Edge device. On each of the devices we want to install it, we run the script:

```
1 export HZN_ORG_ID=<value_of_choice> #as mentioned above
2 export HZN_EXCHANGE_USER_AUTH=admin:<password>  # use the pw deploy-mgmt-
    hub.sh displayed
3 export HZN_EXCHANGE_URL=https://<HZN_LISTEN_IP>:3090/v1 #<HZN_LISTEN_IP>
    is the IP the Management Hub listen to.
4 export HZN_FSS_CSSURL=https://<HZN_LISTEN_IP>:9443/
5 curl -sSL https://github.com/open-horizon/anax/releases/latest/download/
    agent-install.sh | bash -s -- -i anax: -k css: -c css: -p IBM/pattern-
    ibm.helloworld -w '*' -T 120
6 # Install agent and run an example, which we do not need to run.
```

3.7 Deployment of Services Using Open Horizon

As we have mentioned earlier, Open Horizon uses two different ways, namely Patterns and Policies, to manage and deploy services on the Edge nodes. Open Horizon Agents manage the applications on their nodes, which can be standalone Linux machines(managed by

Docker) or clusters of machines(managed by Kubernetes). In our experiments we use Docker on Linux. The main differences between patterns and policies are summarized below:

- Patterns are easy to use and therefore recommended for beginners. Developers directly specify services to run, but they are less powerful and flexible, and, as such, they are not suggested for complicated deployments.
- Policies, on the other hand, declare intent with constraint-based resolution and are used to determine whether a policy will resolve in a deployment. Node policies are resolved against service and business policies in order to determine the deployment.

3.7.1 Using Patterns

Open Horizon Pattern is a named collection of services. The developer must specify the services she wants to include and then OH will add their stated dependencies(other services) [18]. After that, the node must be registered by using the name that the pattern was given and all services will be deployed. Only one pattern can be active on any single node.

An simple example of pattern.json follows:

```
{
  ”name”: ”pattern-edgex-amd64”,
  ”label”: ”This text label only for humans”,
  ”description”: ”More analytical text for humans to understand the
   functionality of this deployment”,
  ”services”: [
    {
      ”serviceUrl”: ”my-service.url”,
      ”serviceOrgid”: ”my-org”,
      ”serviceArch”: ”amd64”,
      ”serviceVersions”: [
        {
          ”version”: ”1.0.1”
        }
      ]
    }
  ],
  ”userInput”:[
```

```
18
19    ]
20 }
```

The most important fields are the following:[18]

- name: We can give the pattern any name we like, as is the case for ”label” and ”description” fields, which are only meant to be read by humans, the computer ignores them.
- public: Defines if people outside our organization can use this pattern.
- services: It is an array of services each having four fields:
  - serviceOrgid, the name of the organization.
  - serviceUrl, the name of the service.
  - serviceArch, the hardware architecture of the service.
  - serviceVersions, a list the service versions that the pattern is applicable to.

3.7.2 Using Policies

There are three different kinds of policies [19] (we do not discuss model policy as it focuses on the deployment of ML models, which is out of the scope of this Thesis) :

- Deployment policy: In order to use the policy mechanism developers must define a deployment policy (also called business policy) [20].
- Service policy: The developer can also attach this to the service they are deploying and it is automatically merged into the deployment policy that references this service.
- Node policy, which is used to register Edge nodes [21].

We also need to mention that node and service policies have the same format. They are basically the same type of policy but used in different situations. In the code block below we can see an example of node/service policy.

```
{
    "properties": [
      {
        "name": "edgeX",
        "value": "true"
      },
      {
        "name": "first_batch",
        "value": "true"
      },
      {
        "name": "openhorizon.allowPrivileged",
        "value": true
      }
    ],
    "constraints": [
      "edgeX == \"true\"",
      "first_batch == \"true\""
    ]
  }
```

We use properties to define the characteristics of each node. The agent will cross-reference these in the exchange with the service constraints, which are conditional expressions, in order to define if the particular service can be deployed on that node. Developers can use whatever value they wish, as they are the ones who define them. There are also some defaults that can be used, as shown in the example (openhorizon.allowPrivileged)

```
{
    "label": "This text label only for humans",
    "description": "More analytical text for humans to understand the
   functionality of this deployment",
    "properties": [
    ],

    "constraints": [
    ],

    "service":{
      "org"             :"my-org",
      "name"            :"my-service.url",
      "arch"            :"amd64",
      "serviceVersions": [ {"version": "1.0.0" , "properity": {} } ]
    }
}
```

The code-block above provides an example of a deployment policy. Deployment policies contain properties and constraints, like a pattern, but the difference is that they have a single service section. It has the same functionality as the pattern services field, but it only contains one service and not an array of services, like the pattern does. So the work flow of the Policies mechanism is:

- All policies have the attributes ”properties” and ”constraints”(we do not need do define both).
- The deployment policy is merged, optionally with the service policy that might exist for a specified service.
- Agbots propose to the node Agent that policy.
- Each Agent evaluates this proposal against their node policy, based in their own properties and constraints.
- If the constraints and properties are resolved, the Agent will accept the proposal and the deployment of the service will proceed.

Chapter 4

Experimentation and Evaluation

4.1 Experimental platform

We deployed our implementation on VirtualBoxd. We used two different setups, one for the Management Hub and one for the node (edge device). In accordance with what was described previously about the prerequisites of the Open Horizon project, the Management Hub device is Ubuntu 20.04 LTS with 4 GB of RAM, 20 GB of storage and 1 virtual CPU, on which the developer has root access. The node device is also Ubuntu 20.04, but equipped with 1 GB of RAM, 10 GB of storage and 1 virtual CPU, also assuming root access. We run both the experiments with the deployment using docker-compose and the deployment using Open Horizon on this setup, in order to have comparable data. Of course, the deployment using docker-compose only needed the node device, as it is the one in which the containers are deployed.

4.2 General information

In order to test the deployment of services using Open Horizon we used the EdgeX Foundry Project. As we mentioned before, EdgeX is a Project under the LF Edge umbrella. It comprises many different micro-services that need to be deployed on the edge nodes. This characteristic makes it a very suitable candidate to test the deployment capabilities of Open Horizon. Apart from everything that was mentioned before about policies and pattern, Open horizon needs a definition of the service that is going to be deployed [22]. The respective json files define the docker image that is used as the base for the service, the required services that

need to be already deployed in order for this one to work and any user input it needs. User input is a set/array of variables that govern how the service implementation in the container image behaves. Probably the most important field to mention here is ”deployment”, that has extensive documentation of its own [23]. This field includes the main definition of the service, like the container name, the ports that the containers should map to the host and other characteristics that the containers should have. More than one services can be included in the deployment string. The service definition is a mandatory file, with both Policies and Patterns deployment methods.

4.3 EdgeX deployment

The usual way the EdgeX Foundry Project is deployed is by using the docker_compose.yml from the official repository on GitHub [24]. We used version 2.3 (Levski). In order to deploy EdgeX with Open Horizon we had to ”translate” this docker_compose file into a service definition [24].

4.3.1 Original Idea

Our initial approach, which we expected to be easier and more efficient, was to create a separate service definition for each of the EdgeX microservices. That way we would have different ”userInput” and ”requiredServices” fields for each of the microservices. This would be an accurate interpretation of the fields ”depends_on” (into ”requiredServices”) and ”environment” (into ”userInout”) of each microservice from the docker_compose file into the service_definition file. Moreover, we would have the flexibility to deploy EdgeX micorservices to different node devices for distributed execution. For each service definition there would also be a respective deployment policy, as we mentioned in the previous chapter. Therefore we created a service definition for each individual component of the EdgeX project (for example consul, edgex-core-command etc.). We populated each of the aforementioned fields in the json file we the corresponding values found in the compose file. The following listing is an example of such a service definition (in the particular example for the edgex-core-command microservice).

```
1 {
2   ”org”: ”$HZN_ORG_ID”,
```

```
  "label": "core-command",
  "url": "core-command.isidoros",
  "version": "1.1.5",
  "arch": "amd64",
  "public": true,
  "sharable": "singleton",
  "requiredServices": [
    {
      "url": "core-consul.isidoros",
      "org": "issy",
      "versionRange": "[0.0.0,INFINITY)",
      "arch": "amd64"
    },
    {
      "url": "redis.isidoros",
      "org": "issy",
      "versionRange": "[0.0.0,INFINITY)",
      "arch": "amd64"
    },
    {
      "url": "core-metadata.isidoros",
      "org": "issy",
      "versionRange": "[0.0.0,INFINITY)",
      "arch": "amd64"
    }
    ],
    "userInput": [
        { "name": "CLIENTS_CORE_COMMAND_HOST", "label": "", "type": "
  string", "defaultValue": "edgex-core-command" },
        { "name": "CLIENTS_CORE_DATA_HOST", "label": "", "type": "string"
  , "defaultValue": "edgex-core-data" },
        { "name": "CLIENTS_CORE_METADATA_HOST", "label": "", "type": "
  string", "defaultValue": "edgex-core-metadata" },
        { "name": "CLIENTS_SUPPORT_NOTIFICATIONS_HOST", "label": "", "
  type": "string", "defaultValue": "edgex-support-notifications" },
        { "name": "CLIENTS_SUPPORT_SCHEDULER_HOST", "label": "", "type":
  "string", "defaultValue": "edgex-support-scheduler" },
        { "name": "DATABASES_PRIMARY_HOST", "label": "", "type": "string"
  , "defaultValue": "edgex-redis" },
```

```
36        { "name": "EDGEX_SECURITY_SECRET_STORE", "label": "", "type": "
    boolean", "defaultValue": "false" },
37        { "name": "MESSAGEQUEUE_EXTERNAL_URL", "label": "", "type": "
    string", "defaultValue": "tcp://edgex-mqtt-broker:1883" },
38        { "name": "MESSAGEQUEUE_HOST", "label": "", "type": "string", "
    defaultValue": "edgex-redis" },
39        { "name": "MESSAGEQUEUE_INTERNAL_HOST", "label": "", "type": "
    string", "defaultValue": "edgex-redis" },
40        { "name": "REGISTRY_HOST", "label": "", "type": "string", "
    defaultValue": "edgex-core-consul" },
41        { "name": "SERVICE_HOST", "label": "", "type": "string", "
    defaultValue": "edgex-core-command" }
42     ],
43     "deployment": {
44       "services": {
45         "edgex-core-command": {
46           "image": "edgexfoundry/core-command:2.3.0",
47           "specific_ports": [
48             {
49               "HostPort":"59882:59882/tcp",
50               "HostIP":"127.0.0.1"
51             }
52           ]
53         }
54       }
55     }
56   }
```

As the reader can observe, this service has three dependencies on the services ”core-consul” , ”redis” and ”core-command”. The URLs of each service are defined by us in the service definition and used when we want to reference that specific service, which means that these URLs are also unique in each Open Horizon Exchange. Regarding the attribute ”userInput”, as we have already mentioned, it defines the environment variables of the container that this specific service definition is going to initialize [22]. We also specify the image that will be used for this container and the mappings of ports of the host to ports of the container. Unfortunately, this approach has a significant limitation: the EdgeX microservices’ containers have to be in the same docker network; they can not live on different ones.

4.3.2 Final Implementation

In order to avoid the docker network connectivity issue, we decided to mimic the implementation of [2] in which all the microservices are defined in one service file. That way, only one docker network is created and all containers are connected to this one. A new problem with this approach is that there is only one field for ”userInput”, which by definition cannot envelope all the environment variables of all the containers that we define in that file. That is because the environment variables defined in ”userInput” are going to be applied to every container. The way we solved that is by setting only the variables that have the same values for everyone or are only used by one container, which would not impact the rest of the containers. The second step, was to find a way to set the remaining environment variables in the containers. We decided to modify the original images provided by the EdgeX Project for each of the microservices. So, for each one we created a Dockerfile using the respective image and setting the environment variables, thus creating a new image to use in its place. The following listing provides an example of a Dockerfile:

```
1 FROM edgexfoundry/core-command:2.3.0
2
3 #environment variables from docker compose
4 ENV MESSAGEQUEUE_EXTERNAL_URL=tcp://edgex-mqtt-broker:1883
5 ENV MESSAGEQUEUE_HOST=edgex-redis
6 ENV MESSAGEQUEUE_INTERNAL_HOST=edgex-redis
7 ENV SERVICE_HOST=edgex-core-command
```

The ”edgexfoundry/core-command:2.3.0” is original image of the core-command microservice, version 2.3.0 (Levski) and the four environment variables are the ones defined in the docker compose file, that had different values for each of the containers and needed to be to set individually. The images where created locally and then uploaded on Docker Hub (of course we could have used any other docker repository). The full list of images on our Docker Hub is the following:

- ilamprinos/edgex_images_database:1.0.3
- ilamprinos/edgex_images_notifications:1.0.3

- ilamprinos/edgex_images_app-service-rules:1.0.3
- ilamprinos/edgex_images_app-service-sample:1.0.3
- ilamprinos/edgex_images_command:1.0.3
- ilamprinos/edgex_images_data:1.0.3
- ilamprinos/edgex_images_device-rest:1.0.3
- ilamprinos/edgex_images_device-virtual:1.0.3
- ilamprinos/edgex_images_rulesengine:1.0.3
- ilamprinos/edgex_images_metadata:1.0.3
- ilamprinos/edgex_images_scheduler:1.0.3
- ilamprinos/edgex_images_system:1.0.3
- ilamprinos/docker-simple-filter-xml:1.0.0

The last image is the application service that was used as proof of concept for the functionality of the EdgeX deployed using Open Horizon. It is a very simple example taken from the official examples of the EdgeX Project [25]. We need to mention here that the image for the concul did not need to be altered as it did not have any environment variables that needed to be set.

4.4 Experimental results

After validating the correctness of the setup and the deployment, we experimentally compared deploying a Project like EdgeX using the docker compose method and the Open Horizon method. Of course, Open Horizon has a different purpose altogether, but the comparison of the two can offer a better perspective on the effectiveness of Open Horizon and the latency associated with deploying a service.
The results are summarized in the following figures and tables:

- Figure 4.1 compares the average time (in seconds) needed for the service to be published in the Exchange in two different circumstances, docker images downloaded a

priori and the same images downloaded on demand. This metric gives a better idea of the different latency when deploying an application for the first time and redeploying the application on the same device.

- Figures 4.2 and 4.3 present the average time (in seconds) needed for the individual stages of the agreement negotiation to be completed, with the only thing changing being again whether the images are downloaded a piori or on demand. As we can see, there is no difference in the first two stages as they are contacted between the AgBots. However, the time needed for the service, in our case Edgex, to start its execution increases excessively, mainly because the Agent has to download the images on the device.

- As we have previously mentioned, we also evaluated the time that it takes for the docker-compose method to deploy EdgeX. Figures 4.4 and 4.5 present the data we collected from that experiment.

- We also investigated the case when there is an additional microservice deployed alongside EdgeX, which is something that would happen in a real scenario. Figures 4.6 and 4.7 contain the same information as 4.2 and 4.3 respectively, with only one difference, the addition of that extra microservice, in order to determine the consequences that it would have on the deployment, which are apparently minor.

- Table 4.1 presents the time in seconds that it takes for the entire process, from adding a new policy to the creation of the agreement on the Management Hub, to be complete.

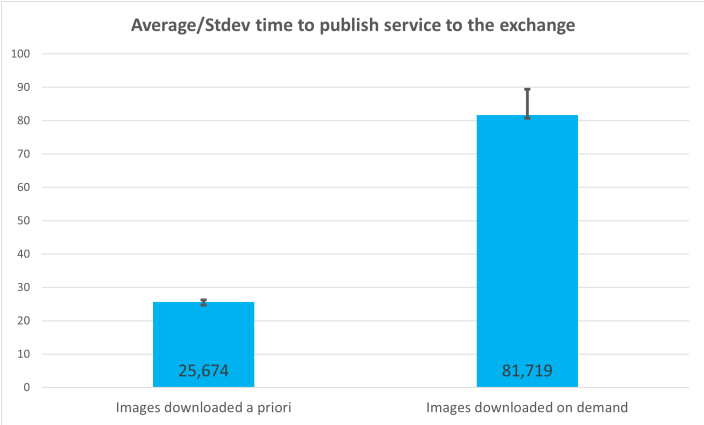

Figure 4.1: Average and Stdev time for publishing a service to the Exchange

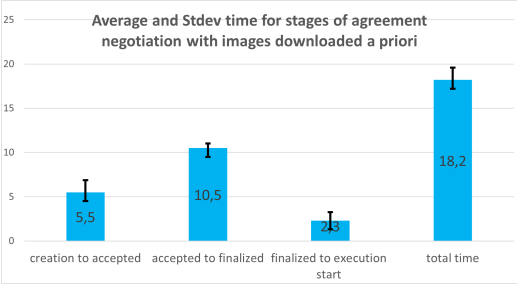

Figure 4.2: Average and Stdev time for stages of agreement negotiation with images downloaded a priori

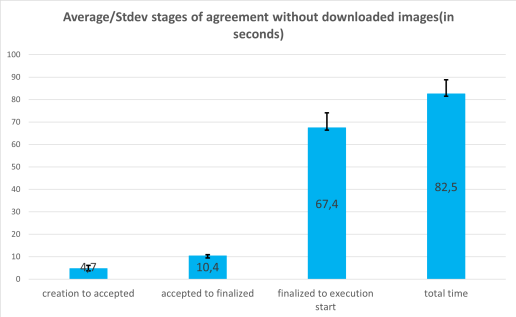

Figure 4.3: Average and Stdev time for stages of agreement negotiation with images downloaded on demand

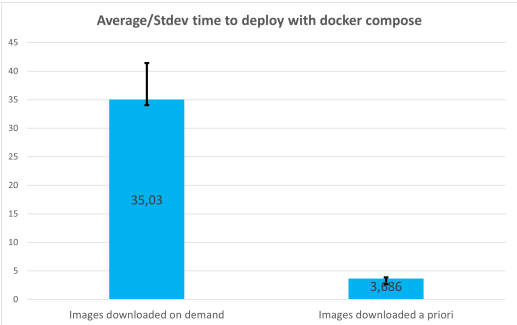

Figure 4.4: Average and Stdev time for containers to reach healthy state with docker compose

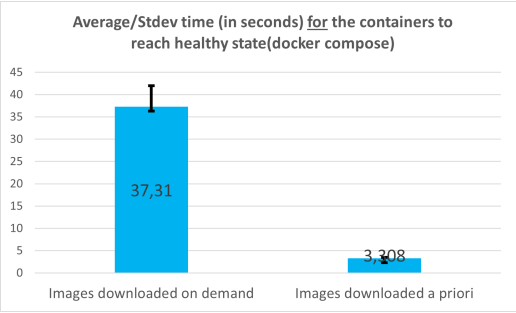

Figure 4.5: Average and Stdev time for containers to reach healthy state with docker compose and extra application service

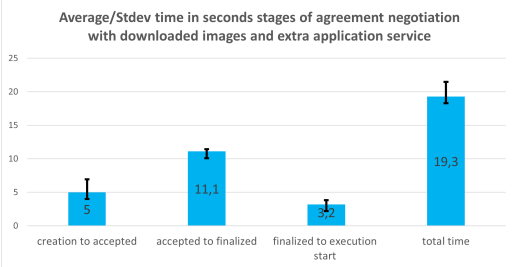

Figure 4.6: Average and Stdev time for the agreement negotiation with docker images downloaded a priori and an extra image for an example application service

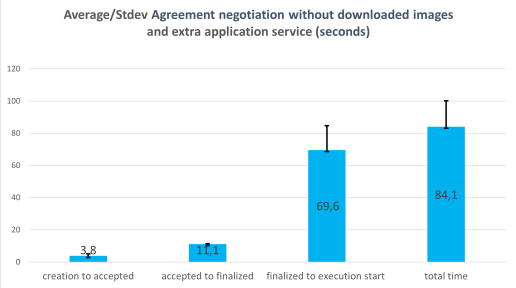

Figure 4.7: Average and Stdev time of the agreement negotiation stages with docker images downloaded on demand and an extra image for an example applications service

	Latency between Adding Policy and Agreement creation
Average time	9.9
Standard Deviation	1.7919

Table 4.1: Latency between adding policy to the Exchange and creation of the agreement negotiation

adding policy to the Exchange:	The creation of the policy using business.json file, also known as the deployment policy
publishing an OH service to the Exchange :	The creation of an Open Horizon service using the service.definition.json and its publishing on the Exchange
agreement_creation_time:	The timestamp that the Agbot has reached out to propose an agreement to the local Agent
agreement_accepted_time:	The timestamp that the Agent accepted the agreement.
agreement_finalized_time:	The timestamp the Agent finalized the agreement with the Agbot.
agreement_execution_start_time:	If the images needed for the service are not downloaded, the Agent downloads them and then the execution starts. If there are already available on the device the execution starts.

Table 4.2: Memo of the components of the above Tables and Figures

4.5 Evaluation of the results

To assess the deployment of EdgeX with Open Horizon, a comparative analysis has been conducted in terms of deployment time against the conventional ”docker-compose.yml” method, as previously discussed. It is important to note that when employing Open Horizon, the image retrieval process occurs twice. Initially, the images are downloaded on the Management Hub device to facilitate the creation of the Open Horizon service and the corresponding service key within the Exchange. Subsequently, a second image download operation is triggered on the node device to initiate the deployment process. In order to publish a service using the service definition file ”service.json” the following command is needed:

```
$ hzn exchange service publish -P -f service.json
```

After that, the policy of that deployment needs to be added to the Exchange, on the Management Hub device, using the deployment policy file business.json and the command:

```
$ hzn exchange business addpolicy -f business.json business.json_policy
```

The name of the policy is defined by the developer. The name that was used in this test case was ”business.json_policy”. After that, the policy is uploaded on the Exchange and the agents on the nodes check, as was discussed previously, if this particular policy is meant for them. This scan, initiated by the nodes, occurs periodically, in order to determine any changes in the Exchange, which means that the same procedure is followed when a policy is deleted from the Exchange and the nodes need to stop and remove the respective containers. Following the verification process where the nodes check the applicability of the policy to their specific node policies, the negotiation of the agreement starts. The explanations of the individual components of the agreement are presented on Table 4.9.

Moving to the comparison between docker compose and Open Horizon methods, the most insightful metric for the docker compose method is the time that transpires for the containers to reach ”healthy state”. We provide that on Table 4.4 and it is on average 35.03 secs if the images are not available on the device and 3.68 if they were. On the other hand, the complexity of Open Horizon means that more than one metric is needed. These are briefly presented on the figures above. These different phases of the negotiation between the Management Hub Agbot and the Agent Agbot need to be explained a little more. Initially, the creation of the agreement takes place on the Exchange on the Management Hub device. Subsequently, the Agbot located on the node’s side is required to approve this proposal, marking the finalization of the agreement. Following this pivotal step, the Agent situated on the node assumes the responsibility for the downloading of the Docker images and the subsequent execution of the resulting containers. The average time that is needed for the Open Horizon to publish EdgeX on the node device is, in total, around 53.7 secs if the images are downloaded on both the Management Hub and on the Agent. If they are not, this goes up to around 174.1 secs. These numbers make sense as the images need to be downloaded twice. As we can see, the difference between Open Horizon and docker compose is noticeable. For the purposes of the Open Horizon this time difference is acceptable and not problematic, as we have discrete goals when we use these two methods. As it was mentioned previously, the goal from the comparison of these was to make an estimate of the increase in the deployment time that the Open Horizon would cause. It was never indented to be a direct comparison

Chapter 5

Conclusion

This Thesis focused on the rapidly evolving landscape of cloud computing and the emerging importance of deploying applications at the edge of the network. The deployment and management of containerized software services in these edge environments, characterized by resource constraints, latency sensitivity, and dynamic conditions, present unique challenges. This study seeks to address these challenges through a two-fold approach: firstly, by exploring the technical feasibility of deploying containerized services at the edge using the open-source projects EdgeX Foundry and OpenHorizon, with a focus on resource optimization; and secondly, by investigating the practicality of integrating these projects to establish a unified methodology for deploying containerized services at the edge.

Our work has proved the practical benefits of merging EdgeX Foundry and OpenHorizon through a thorough evaluation of existing approaches and tactics, as well as through real-world experiments. This integration not only optimizes the deployment and orchestration of containerized services at the edge, but also provides valuable insights and lessons for both practitioners and academics. We expect these findings to be useful and inspirational to practitioners who want to use these technologies to deploy and manage containerized software services at the edge.

5.1 Future work and potential extensions

In order to extent the capabilities of EdgeX through Open Horizon deployment, the future extension of this work could be to find a way to deploy the different components of EdgeX in different nodes. Also, it seems that the development of costume distributed applications

designed for deployment at the Edge of the network would further utilize Open Horizon and prove its efficiency.

Bibliography

[1] Open horizon official lf edge page. https://www.lfedge.org/projects/openhorizon/.

[2] Edgex/open horizon integration. https://github.com/edgexfoundry-holding/open-horizon-integration/blob/main/hub/01-horizon-services-setup.md/.

[3] Kevin Pitstick and Jacob Ratzlaff. Containerization at the edge. Carnegie Mellon University, Software Engineering Institute's Insights (blog), Mar 2022. Accessed: 2023-May-15.

[4] Importance of containers. https://www.ibm.com/topics/containers.

[5] Docker. https://www.ibm.com/topics/docker.

[6] Cloud computing. https://www.ibm.com/topics/cloud-computing.

[7] Benefits cloud computing. https://cloud.google.com/learn/advantages-of-cloud-computing.

[8] Edge cloud computing. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-edge-computing/.

[9] Iot importance. https://www.linkedin.com/pulse/what-iot-why-so-popular-how-do-businesses-benefit-from-.

[10] All-in-one horizon management hub, agent and cli. https://open-horizon.github.io/docs/mgmt-hub/docs/index.html.

[11] Lf edge official page. https://www.lfedge.org/.

[12] Lf edge official page akraino. https://www.lfedge.org/projects/akraino/.

[13] Lf edge official page edgex foundry. https://www.lfedge.org/projects/edgexfoundry/.

[14] Lf edge official page project eve. https://www.lfedge.org/projects/eve/.

[15] Lf edge official page home edge project. https://www.lfedge.org/projects/homeedge/.

[16] Tutorial on how to install ubuntu server. https://open-horizon.github.io/common-requests/install/#pre-requisites.

[17] Virtualbox. https://www.virtualbox.org/wiki/Downloads.

[18] Patterns and policies from the official lfedge yt channel. https://www.youtube.com/watch?v=alcHKc3Upbk&t.

[19] policy.md from the official open horizon documentation. https://github.com/open-horizon/anax/blob/master/docs/policy.md.

[20] deployment_policy.md from the official open horizon documentation. https://github.com/open-horizon/anax/blob/master/docs/deployment_string.md.

[21] node_policy.md from the official open horizon documentation. https://github.com/open-horizon/anax/blob/master/docs/node_policy.md.

[22] Service definition from the official open horizon documentation. https://github.com/open-horizon/anax/blob/master/docs/service_def.md.

[23] Deployment string of the service definition from the official open horizon documentation. https://github.com/open-horizon/anax/blob/master/docs/deployment_string.md.

[24] Docker compose yaml deploying all microservices. https://github.com/edgexfoundry/edgex-compose/blob/levski/docker-compose-no-secty.yml.

[25] Edgex examples. https://github.com/edgexfoundry/edgex-examples.

APPENDICES

	Images downloaded a priori	Images downloaded on demand
	26.85	89.65
	25.83	74.73
	25.14	79.31
	25.46	79.03
	25.86	74.82
	24.86	99.61
	25.25	81.70
	26.17	80.79
	25.86	75.11
	25.46	82.44
Average	25.67	81.71
StDev	0.5717	7.728

Table A.1: Time in seconds for publishing an OH service to the exchange multiple tries

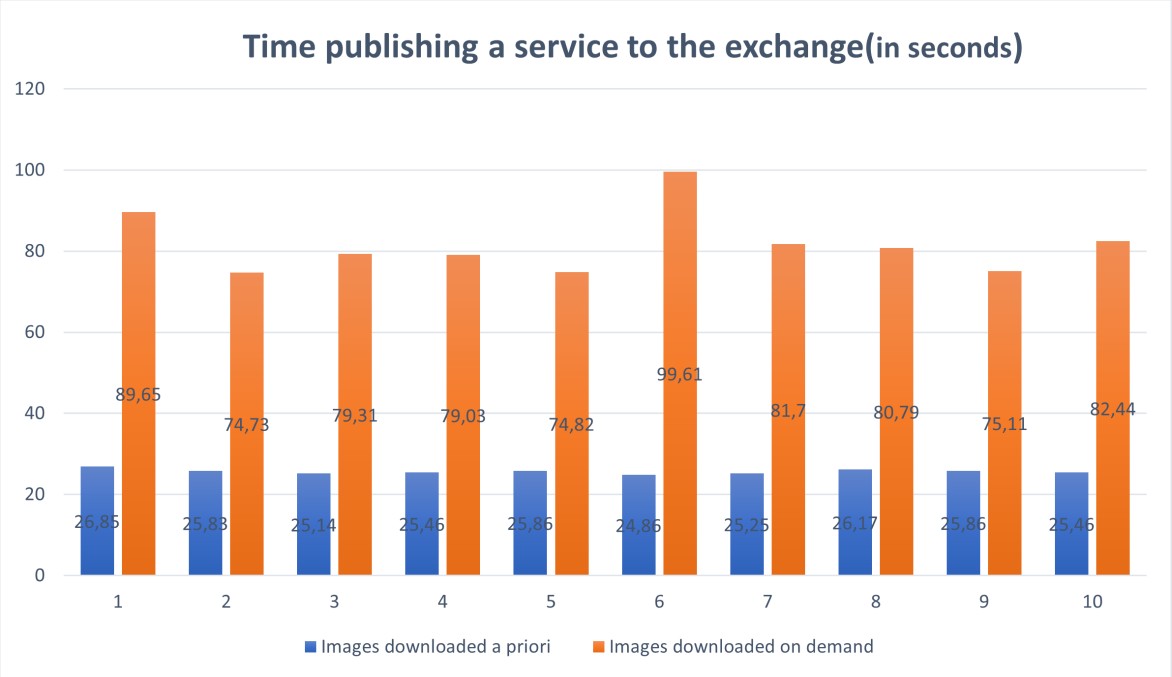

Figure A.1: Bar graph time in seconds for publishing a service to the exchange

	creation to accepted	accepted to finalized	finalized to execution start	total time
	7	11	3	20
	6	11	2	19
	7	10	3	19
	4	10	2	16
	6	11	1	19
	4	10	4	18
	7	10	2	19
	4	11	1	16
	6	11	2	19
	4	10	3	17
Average	5.5	10.5	2.3	18.2
StDev	1.35	0.52	0.94	1.39

Table A.2: Stages of the agreement negotiation with docker images downloaded a priori in seconds

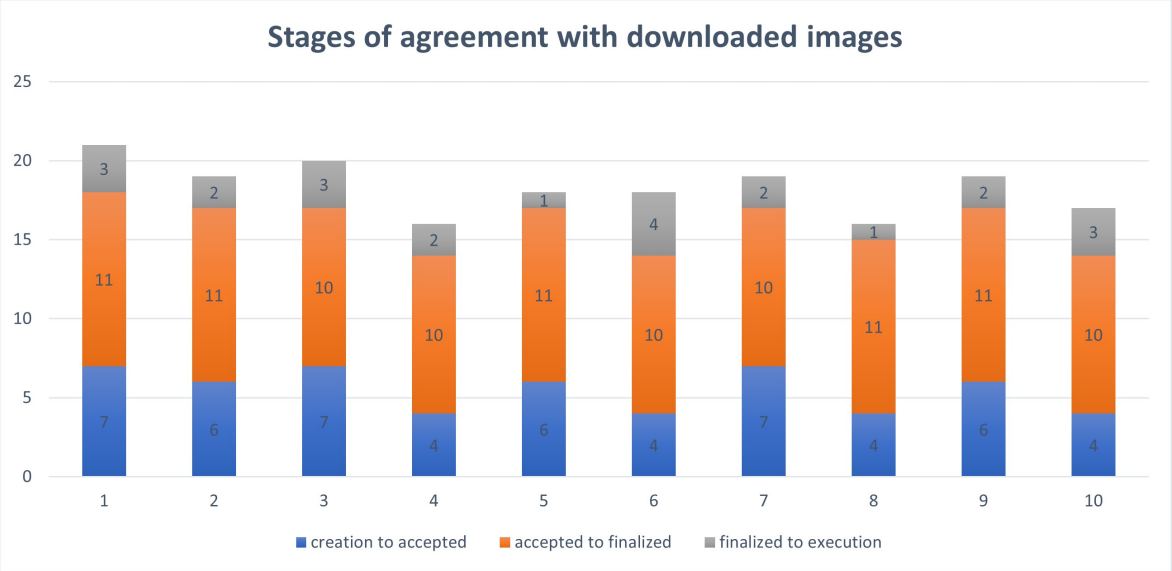

Figure A.2: Stack bar graph for stages of agreement negotiation with downloaded images in seconds

	creation to accepted	accepted to finalized	finalized to start_execution	total time
	3	11	64	78
	7	11	64	82
	4	10	80	94
	4	10	78	92
	4	10	65	79
	6	10	68	84
	4	10	60	74
	7	10	61	78
	4	11	68	83
	4	11	66	81
Average	4.7	10.4	67.4	82.5
StDev	1.41	0.51	6.65	6.25

Table A.3: Stages of the agreement negotiation without downloaded docker images

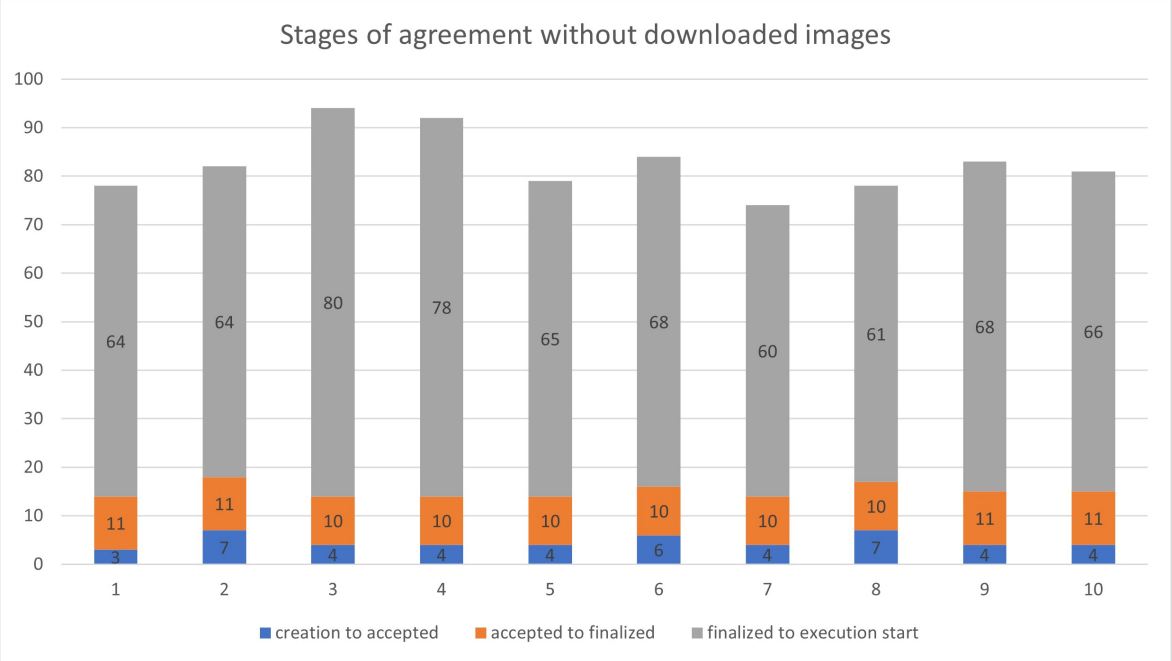

Figure A.3: Stack bar graph for stages of agreement with images downloaded on demand in seconds

	Images downloaded on demand	Images downloaded a priori
	29.68	3.65
	33.10	3.70
	35.34	3.64
	36.64	4.10
	32.63	3.56
	38.82	3.57
	30.53	3.80
	50.76	3.59
	28.65	3.66
	34.15	3.50
Average	35.03	3.68
StDev	6.362	0.165

Table A.4: Time that transpired in seconds for containers to reach healthy state with docker compose

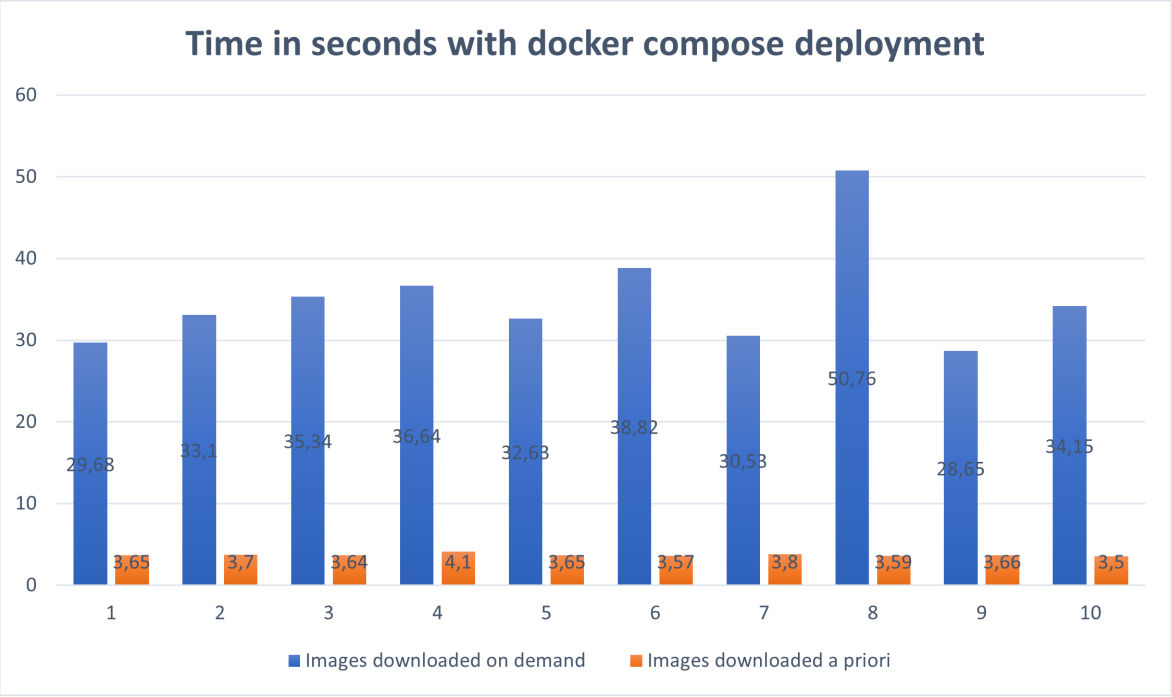

Figure A.4: Stack bar graph for time in seconds with docker compose deployment

	Images downloaded on demand	Images downloaded a priori
	34.85	3.21
	36.79	3.24
	32.91	3.57
	36.64	3.35
	35.91	3.11
	48.46	3.49
	36.89	3.16
	40.92	3.08
	31.68	3.45
	38.05	3.42
Average	37.31	3.3
StDev	4.688	0.1711

Table A.5: Time that transpired in seconds for containers to reach healthy state with docker compose(sample-filter-xml)

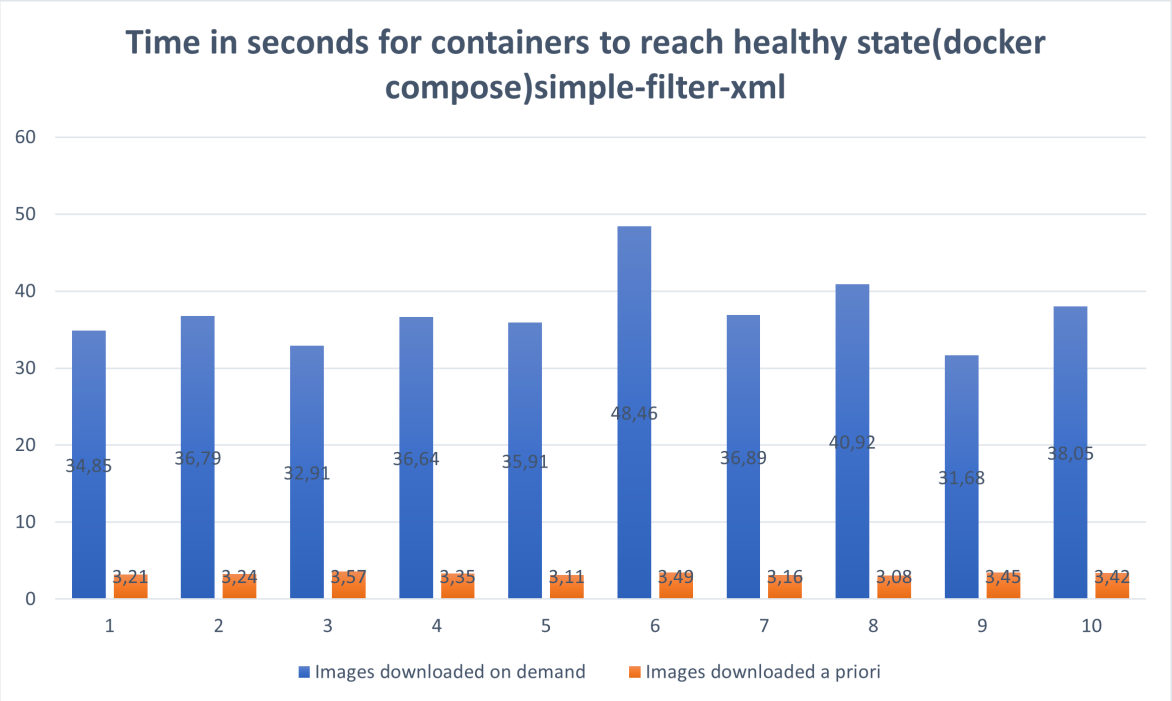

Figure A.5: Bar graph time in seconds for containers to reach healthy state with extra application service

	creation to accepted	accepted to finalized	finalized to start_execution	total time
	3	11	4	16 sec
	7	11	3	20 sec
	6	12	3	21 sec
	3	11	3	17 sec
	4	11	4	18 sec
	3	11	3	16 sec
	7	11	3	21 sec
	3	11	2	17 sec
	7	11	3	20 sec
	7	11	4	21 sec
Average	5	11.1	3.2	19.3 sec
StDev	1.94	0.31	0.63	2.16

Table A.6: Stages of the agreement negotiation with docker images downloaded a priori and an extra image for an example applications service(simple-filter-xml)

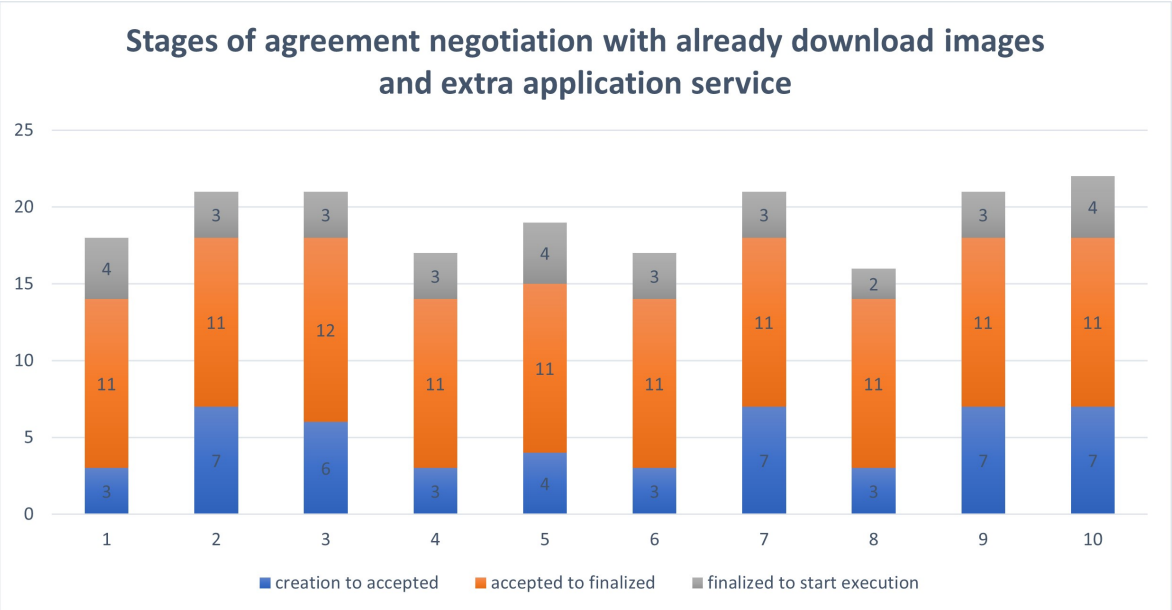

Figure A.6: Bar graph time in seconds oif stages of agreement negotiation with images downloaded a priori and one extra application service

	creation to accepted	accepted to finalized	finalized to execution_start	total time
	4	11	60	75
	3	11	62	76
	4	11	101	116
	3	12	62	77
	3	11	61	65
	4	11	95	110
	3	11	62	76
	3	11	64	84
	7	11	64	82
	4	11	65	80
Average	3.8	11.1	69.6	84.1
StDev	1.229	0.316	15.10	16.12

Table A.7: Stages of the agreement negotiation without downloaded docker images and an extra image for an example applications service(simple-filter-xml)

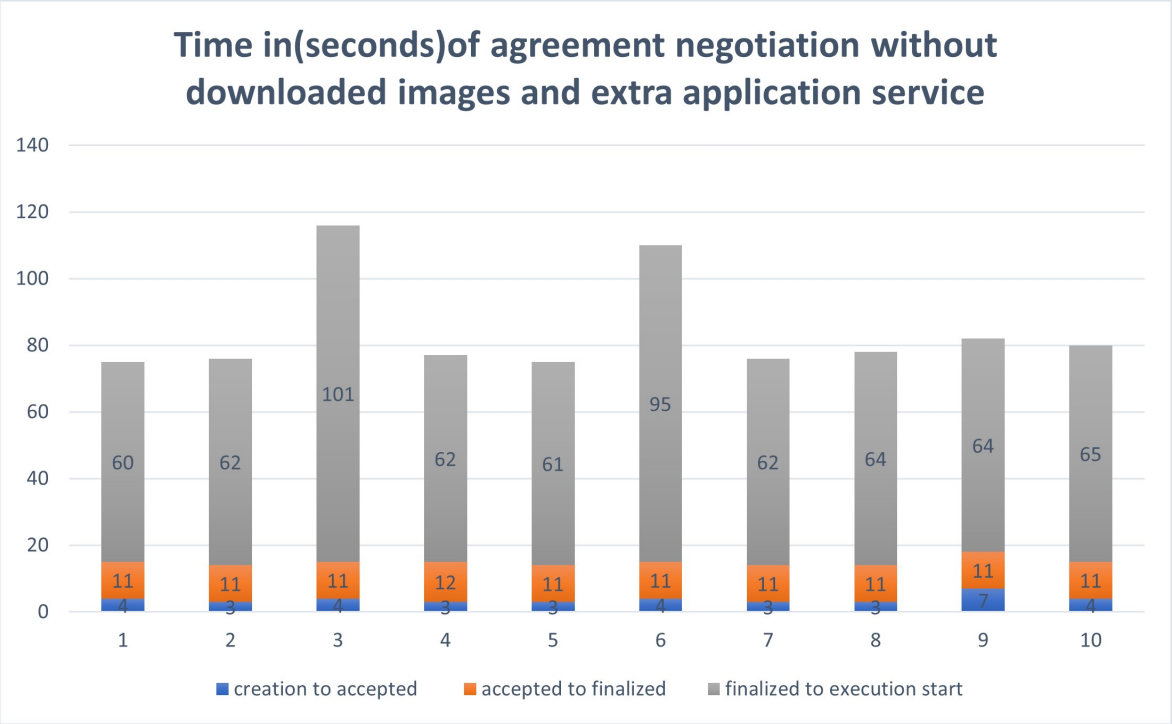

Figure A.7: Bar graph time in seconds of agreement negotiation with images downloaded on demand and an extra application service

	Time transpired between Adding Policy and Agreement creation
	7
	10
	8
	9
	10
	11
	12
	9
	13
	10
Average time	9.9
Standard Deviation	1.7919

Table A.8: Time transpired between adding policy to the exchange and creation of the agreement negotiation

	Images downloaded a priori	Images downloaded on demand
Average	25.67	81.71
StDev	0.5717	7.728

Table A.9: Time in seconds for publishing an OH service to the Exchange

	creation to accepted	accepted to finalized	finalized to start execution	total time
Average	5.5	10.5	2.3	18.2
StDev	1.35	0.52	0.94	1.39

Table A.10: Stages of the agreement negotiation with docker images downloaded a priori (in seconds)

	creation to accepted	accepted to finalized	finalized to start execution	total time
Average	4.7	10.4	67.4	82.5
StDev	1.41	0.51	6.65	6.25

Table A.11: Stages of the agreement negotiation without downloaded docker images (in seconds)

	Images downloaded on demand	Images downloaded a priori
Average	35.03	3.68
StDev	6.362	0.165

Table A.12: Time in seconds for containers to reach healthy state with docker compose

	Images downloaded on demand	Images downloaded a priori
Average	37.31	3.3
StDev	4.688	0.1711

Table A.13: Time in seconds for containers to reach healthy state with docker compose and extra application service

	creation to accepted	accepted to finalized	finalized to start_execution	total time
Average	5	11.1	3.2	19.3 sec
StDev	1.94	0.31	0.63	2.16

Table A.14: Stages of the agreement negotiation with docker images downloaded a priori and an extra image for an example application service

	creation to accepted	accepted to finalized	finalized to execution_start	total time
Average	3.8	11.1	69.6	84.1
StDev	1.229	0.316	15.10	16.12

Table A.15: Stages of the agreement negotiation stages with docker images downloaded on demand and an extra image for an example applications service