UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Deployment and communication infrastructure for modern distributed applications on edge and cloud environments

Diploma Thesis

Triantafyllos Anargyros Giannoukos

Supervisor: Spyros Lalis

Volos 2020

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Deployment and communication infrastructure for modern distributed applications on edge and cloud environments

Diploma Thesis

Triantafyllos Anargyros Giannoukos

Supervisor: Spyros Lalis

Volos 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Υποδομή διαχείρισης και επικοινωνίας για σύγχρονες κατανεμημένες εφαρμογές σε περιβάλλοντα edge και cloud

Διπλωματική Εργασία

Τριαντάφυλλος Ανάργυρος Γιαννούκος

Επιβλέπων: Σπύρος Λάλης

Βόλος 2020

Approved by the Examination Committee:

Supervisor **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

Member **George Thanos**
Laboratory Teaching Staff, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 9-10-2020

Only in the darkness
can you see the stars.

-Martin Luther King Jr.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor Christos D. Antonopoulos for his continuous support during the last years of my studies. His patient guidance and excellent feedback were pivotal for the completion of this work. Furthermore, through his brilliant lectures in his undergraduate courses, he inspired my decision to choose a software development career path. I would also like to show my warmest appreciation to Professor Spyros Lalis for his significant and constructive suggestions during the planning and development of this work. His opinion and invaluable remarks were critical for the completion of this Thesis. I would like to express my greatest acknowledgment to Prof. Nikolaos Bellas for his valuable advice over the last years of my studies and for his participation in the evaluation committee of this Thesis. I would also like to thank Manos Koutsoubelias for his crucial guidance when it came to the experimental evaluation part of my work.

To my dear friends, I am eternally grateful for all the amazing moments we shared over the past five years. Thank you for being always there for me, for supporting me during tough times, and for never letting me down. Finally, I would like to thank my family for their endless love, encouragement, and support throughout my studies.

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 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

Triantafyllos Anargyros Giannoukos
9-10-2020

Abstract

The emergence of cloud and edge computing has changed the way we design applications. It has also created some challenges that we need to handle regarding infrastructure. Some of them are processing large amounts of data in a short amount of time and providing resilience and security. In this thesis, we attempt to address some of these challenges by designing an infrastructure for applications that we want to deploy at the edge or the cloud. We use Apache Kafka to provide communication between our services, allowing for high throughput in messages produced at the edge, while delivering reliability and scalability. Moreover, we make use of Kubernetes to deploy our application. In this manner, we provide fault-tolerance, monitoring, self-healing, and scalability features for our applications hosted on the fog or the cloud. Then, we put our infrastructure to the test, by using a real-world application that we designed, in order to evaluate the performance of our system.

Περίληψη

Η εμφάνιση του cloud και του edge computing έχει αλλάξει τον τρόπο με τον οποίο σχεδιάζονται οι εφαρμογές. Επίσης έχει δημιουργήσει κάποιες προκλήσεις που πρέπει να διαχειριστούμε, οι οποίες αφορούν την υποδομή. Κάποιες από αυτές είναι η επεξεργασία μεγάλης ποσότητας δεδομένων σε σύντομο χρονικό διάστημα και η εξασφάλιση ανθεκτικότητας και ασφάλειας. Σε αυτήν τη διπλωματική εργασία, επιχειρούμε να αντιμετωπίσουμε κάποιες από αυτές τις προκλήσεις σχεδιάζοντας μια υποδομή για εφαρμογές που θέλουμε να διαχειριστούμε στο edge και στο cloud. Χρησιμοποιούμε το Apache Kafka για την επικοινωνία ανάμεσα στα services μας, ώστε να έχουμε υψηλή ρυθμαπόδοση στα μηνύματα που παράγονται στο edge, ενώ ταυτόχρονα εξασφαλίζεται η αξιοπιστία και η επεκτασιμότητα. Επιπρόσθετα, χρησιμοποιούμε το Kubernetes για να διαχειριστούμε την εφαρμογή μας. Με αυτόν τον τρόπο, παρέχουμε ανοχή βλαβών, παρακολούθηση, αυτο-θεράπευση και επεκτασιμότητα στις εφαρμογές μας που φιλοξενούνται στο edge και το cloud. Στη συνέχεια, δοκιμάζουμε την υποδομή μας, χρησιμοποιώντας μια εφαρμογή που χρησιμοποιείται στον πραγματικό κόσμο, η οποία σχεδιάστηκε από εμάς, ώστε να αξιολογήσουμε τις επιδόσεις του συστήματός μας.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Introduction

1.1.1 Motivation

Cloud computing has become more and more popular over the last few years. Many companies use the cloud for backup storage, as a backend infrastructure for their Internet of Things applications, of for big data analytics. One of the main reasons for its wide adoption is the fact that it provides the power of enormous data centers at a significantly lower cost than building and maintaining one's own IT systems. Moreover, the cloud is flexible and it allows scaling up or down on demand, reducing costs for the businesses that use it.

One of the disadvantages of cloud computing is the increased latency when accessing the datacenter from the outside. This can be a problem for applications that demand frequent communication with the cloud. To solve this problem, *edge computing* attempts to bring computation and storage closer to the location where the data are generated and used. To achieve this, it creates an extra layer between the edge devices and the cloud, as illustrated in Figure 1.1. Edge computing is significant for Internet of Things applications since they produce massive amounts of data and require small latency so that they can function properly. Moreover, since we reduce the amount of data sent to the cloud, we reduce the risks related to security and privacy.

Using edge computing, though, introduces some other challenges. First, in order to reduce

latency, we need to process large volumes of data in real-time. Then, we have to aggregate the data and send them to the cloud for further processing. There is also the need for resiliency and fault-tolerance in order to maintain stability. Monitoring is essential for this purpose. One more difficulty of working at the edge is the limited hardware available, accompanied by little to no IT support.

1.1.2 Contribution

This Thesis presents our work on evaluating an infrastructure suitable for edge computing, focusing on the challenges stated above. More specifically, we used *Apache Kafka* [1] for the communication between the services hosted in the edge devices, the fog, and the cloud. We made this choice because Kafka focuses on offering high throughput and high volume real-time message processing while providing reliability and replication features to our data. Also, Kafka gives us the ability to reprocess data even after they have been consumed since it stores the messages to the disk, which is very useful for use at the edge.

In order to deploy an application, our infrastructure makes use of Kubernetes [12]. First of all, this provides our infrastructure with the flexibility to deploy on any kind of hardware, supporting heterogeneous systems. Moreover, we can provide resiliency to our application, by exploiting the fault-tolerance and self-healing features that Kubernetes offers.

1.1.3 Thesis Structure

The rest of the Thesis is structured as follows:

Chapter 2 provides background information, presenting the details of the frameworks, namely Docker, Kubernetes and Apache Kafka, that we simultaneously used in this Thesis.

Chapter 3 elaborates on the details of the communication and deployment infrastructure. It also describes the application that we created for the purpose of testing and evaluating the system.

Chapter 4 discusses the results of the evaluation of our system.

Chapter 5 concludes with a summary of the Thesis and introduces some ideas for future work.

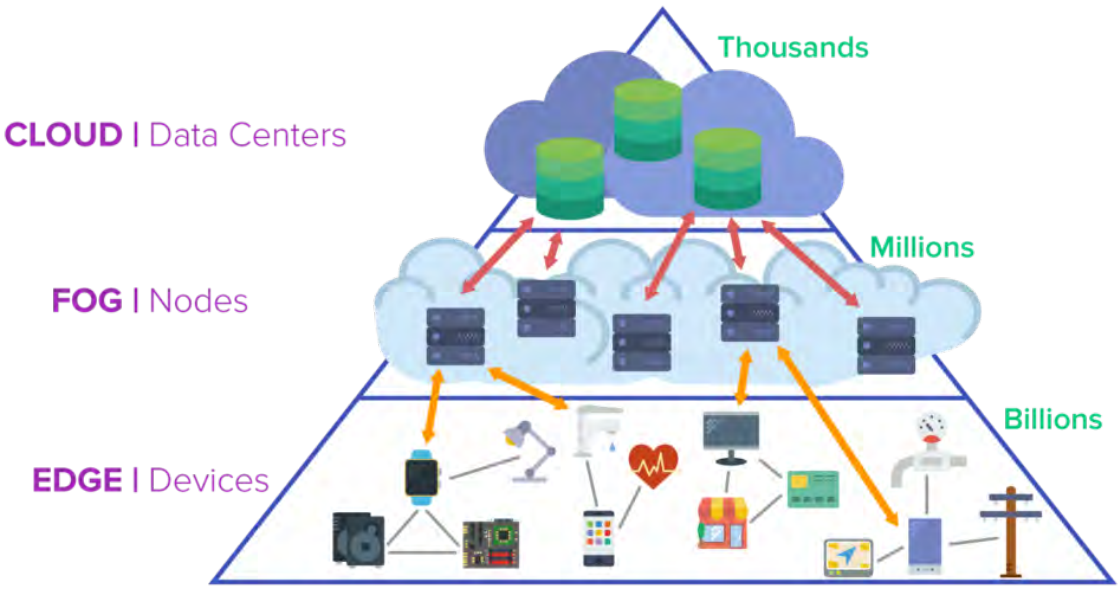


Figure 1.1: Edge computing Architecture [15]

Chapter 2

Background

In this chapter we will introduce the main technologies that we used, elaborating on their features and their architecture.

2.1 Docker

Docker [8] is an open-source containerization platform. It allows developers to create, deploy, and run applications by using containers.

2.1.1 Docker Containers

Containers make use of Operating System (OS) virtualization and process isolation. They allow multiple applications to share the resources of a single instance of an OS kernel. Containers can have their own processes, their own networking interfaces, their own file systems, similar to machine virtualization. However, as shown in Figure 2.1, unlike conventional Virtual Machines (VMs), each container does not have its own Operating System, as they access the kernel of the host Operating System. Thus, containers have significantly reduced overhead compared to Virtual Machines. They use less disk space, take less time to boot, and are much more efficient in CPU usage. In most cases, except for the OS kernel, each container also shares the binaries and libraries with its host. As a result, there is no need to reproduce the OS code and a server can run multiple workloads with a single OS installation.

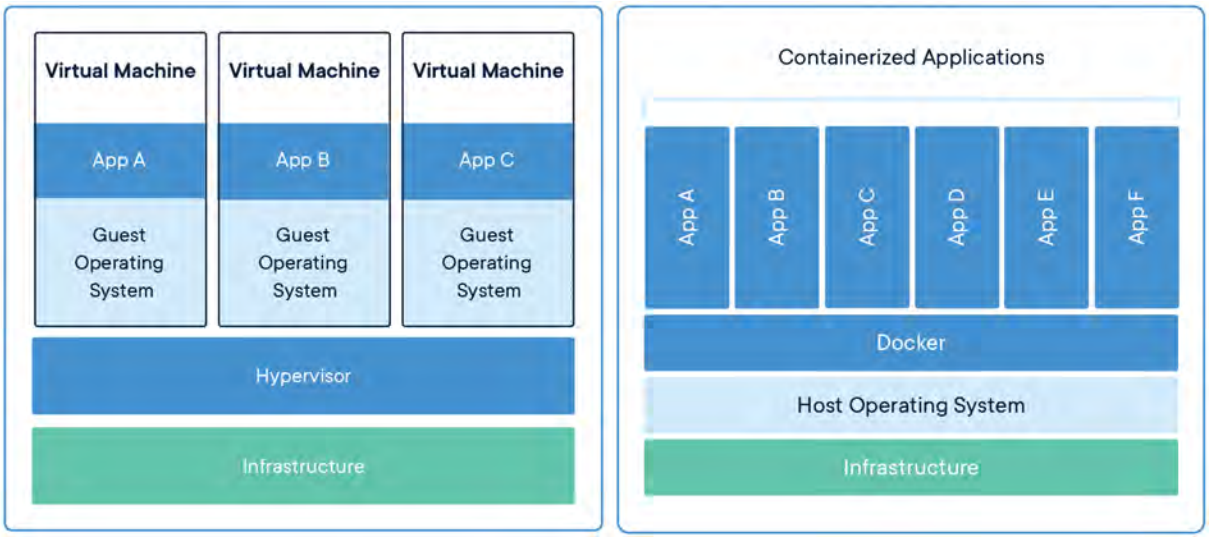


Figure 2.1: Virtual Machines vs. Containers [5]

Building containers would not be possible if it were not for two very important Linux kernel capabilities:

- *Namespaces*, that provide processes with an isolated view of the system. There are 6 types of namespaces. For instance, processes that are part of the same *pid namespace* can only see processes within that namespace. Moreover, processes in the same *net namespace* are provided with their own private network stack, while processes in the same *mnt namespace* have their own root file system.

- *cgroups*, which limit the resources that a process or group of processes can use. These resources can be CPU utilization, memory usage, network I/O, or access to the file system.

2.1.2 Docker Architecture

Docker uses a client-server architecture. As Figure 2.2 shows, it comprises three main components:

- the *Docker Client*, which is the main way of interaction between Docker and the user. When the user enters a command, such as `docker run`, the client sends it to the

Docker daemon.

- the *Docker Host*, which provides a complete environment to execute and run applications. It includes the Docker daemon, images, containers, networks, and storage. The Docker daemon listens for Docker API requests sent from the client and manages all the objects included in the Docker host. For example, if a `docker run` command is received, the daemon will create a container based on the requested image.

- the *Docker Registry*, which contains Docker repositories that host Docker images. There exist private registries as well as public registries, such as *Docker Hub*.

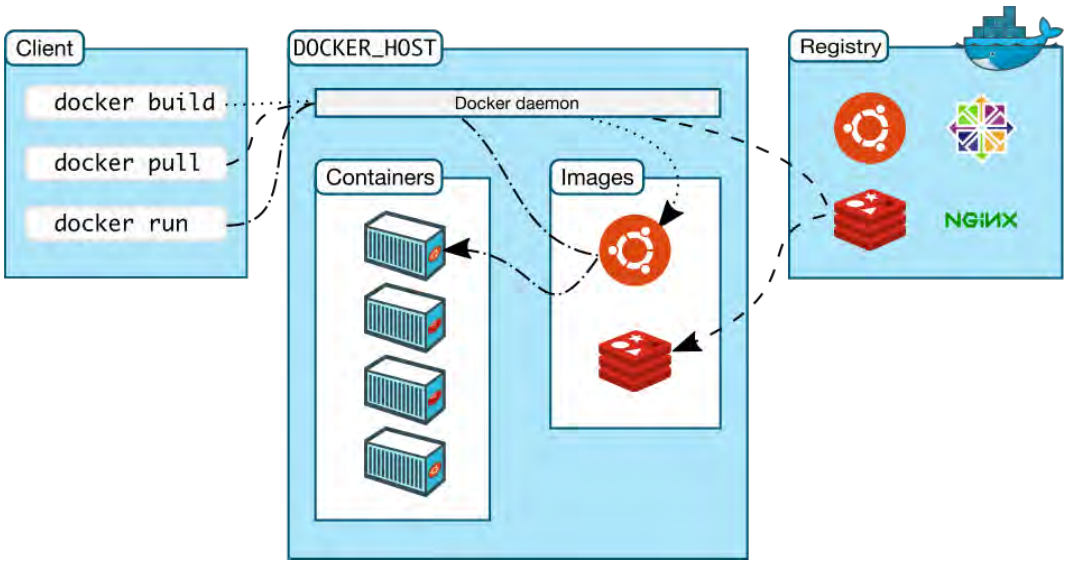


Figure 2.2: Docker architecture [6]

Images are read-only binary blueprints that are used to build containers. They contain application source code along with all the tools, libraries, and dependencies that the application code needs to run as a container. It is possible to create a container image from scratch, though it is more common to use images uploaded on the Docker Hub as a reference. In both cases, when building your own image, you have to create a *Dockerfile* in order to specify the steps needed to create and to run the image. The Dockerfile contains a list of commands, written in simple syntax, that the Docker Engine will execute to assemble the image.

2.2 Kubernetes

Kubernetes [12] is an open-source system for automating deployment, scaling, and management of containerized applications. It provides the orchestration and management capabilities required to deploy containers, at scale, for real production app workloads. Kubernetes orchestration allows us to build application services that make use of multiple containers, schedule those containers across a cluster, scale them, and manage their health over time. [18]. Kubernetes runs on top of an operating system and interacts with groups of containers running on the nodes. The developer interacts with Kubernetes through its CLI, by executing specific commands as well as providing manifests in YAML files, that contain information about the desired architecture of the system. The architecture of Kubernetes is depicted in Figure 2.3.

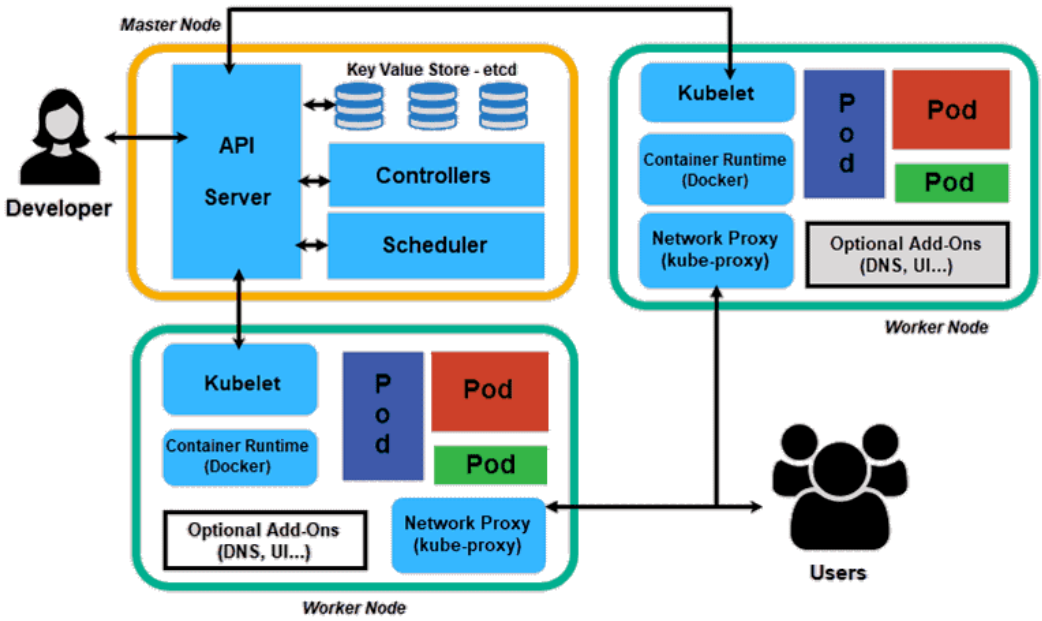


Figure 2.3: Kubernetes Architecture [20]

The highest-level Kubernetes abstraction, the *cluster*, refers to the group of machines running Kubernetes and the containers managed by it. A Kubernetes cluster must contain a *master* node. The Kubernetes master takes the commands from an administrator and relays those commands to the subservient nodes. It performs the scheduling required to spread the

workload across multiple nodes. Also, it is responsible for reconfiguring the workload in cases of node failures. Each cluster contains Kubernetes nodes. Nodes might be physical machines or VMs. These nodes run *pods*, collections of one or more containers. The containers within a pod always run on the same node and they also share the same IP address, memory, and volumes. The pod serves as Kubernetes' core unit of management. However, they are not usually launched directly on a cluster. Instead, they are managed by one more layer of abstraction, the *deployment*. A deployment is responsible for maintaining a sufficient number of replicas of an image at any time. If, for example, a pod dies the deployment is responsible for creating another one. Each node includes two main components, the *kubelet* and the *kube-proxy*. The kubelet manages the communication between the node and its master. The kube-proxy allows network traffic to come in and then be redirected into the various pods. A *service* is defined as a set of pods that work together. Each service has its IP address in the cluster, enabling effective communication between different types of pods (e.g. front-end servers and back-end servers). There are different types of services. For example, the *ClusterIP* service enables communication within the cluster, while the *NodePort* service also allows for external communication. Figure 2.4 displays the use of a ClusterIP service.

An important aspect of services is that they do not live on a specific node, but they belong to the whole cluster. Hence, every container that belongs to the cluster can communicate with the service transparently. It is essential to know that services are implemented by the kube-proxy component, that runs on every node. The kube-proxy creates iptables rules that redirect requests to pods and it receives updates from the Kubernetes API whenever a change in the configuration of a service occurs.

When a container shuts down, all of its data are deleted. This does not cause problems in stateless applications, but it is not compatible with stateful applications. To work this issue around, Kubernetes provides *Persistent Volumes*, a persistent storage mechanism for containers. They allow the developer to mount a file system to the cluster and to share information between nodes. Persistent volumes are hosted in their own pods so that they can remain alive for long periods.

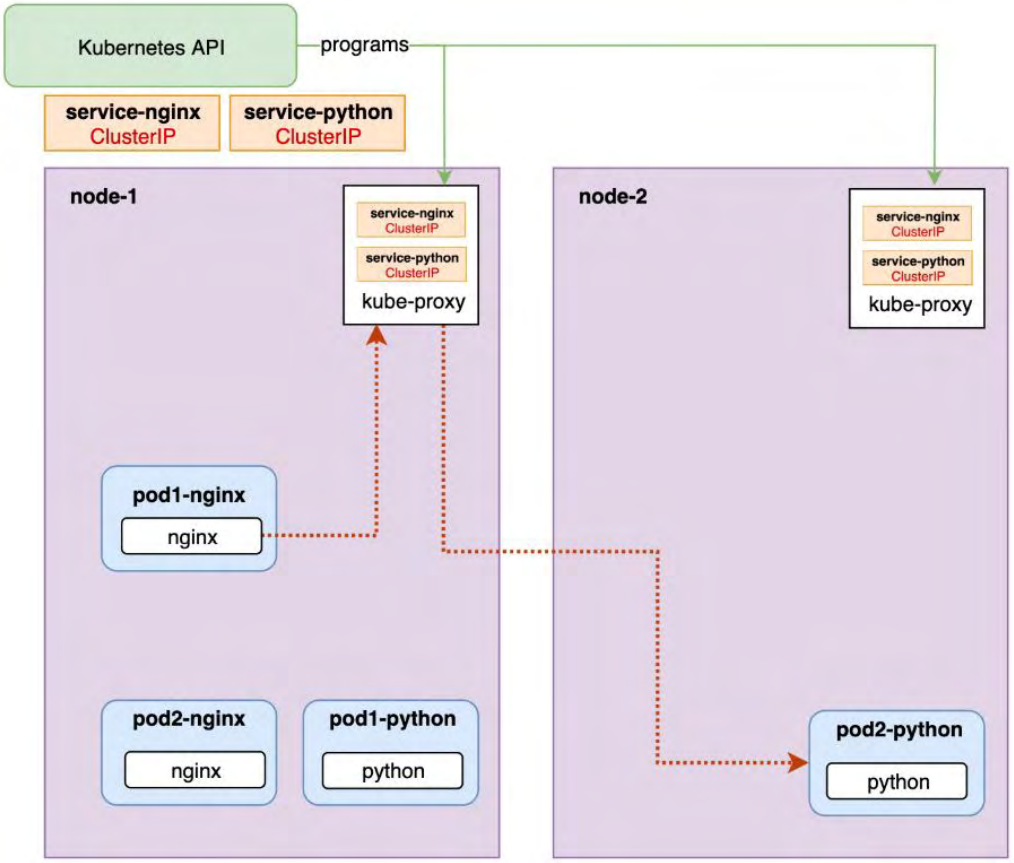


Figure 2.4: Kubernetes Request [10]

2.3 Apache Kafka

Kafka [1] is a distributed messaging system that is used for collecting and delivering high volumes of data with low latency. It was created by LinkedIn in order to manage the large amount of log data generated by user activity as well as operational metrics.

Kafka combines features from existing messaging systems and log aggregators and it is suitable for both offline and online message consumption. However, it also incorporates some unconventional features, in order to improve the performance and the scalability of the system. In contrast to Kafka, conventional messaging systems usually offer strong delivery guarantees, such as allowing each individual message to be acknowledged after it is consumed. Such mechanisms are excessive for collecting log data and they increase the overhead and the complexity of those systems. Moreover, most messaging systems are weak in

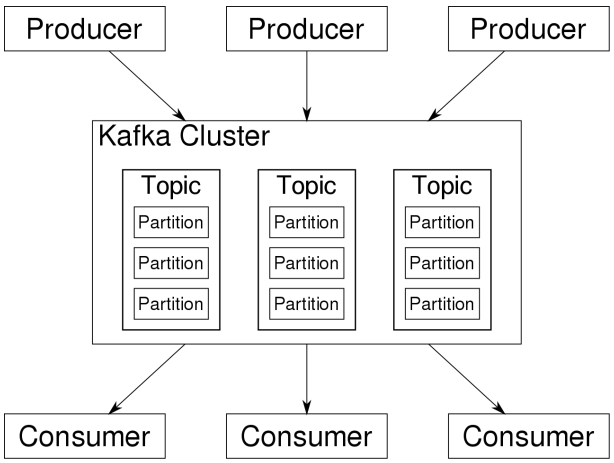


Figure 2.5: Kafka Architecture [21]

distributed support and they assume near-immediate consumption of messages received. [11]

As shown on Figure 2.5 the important structures of Kafka are the producers, brokers, and consumers, and the key concept is the topic. A *topic* is defined as a stream of messages of a particular type and it is labeled with a specific name. The *producer* publishes messages to a topic. Then, the messages are managed by the *broker*. Multiple Kafka brokers form a Kafka *cluster*. The cluster manages the data between producers and consumers. The *consumers* subscribe to one or more topics in order to receive the messages.

In Kafka, producers can submit a set of messages in a single request. Also, consumers receive multiple messages in every request, even when they are processing them one by one. The Kafka broker is stateless and the information about the consumer offsets is maintained by the consumers themselves.

A topic can be divided into several partitions, as can be seen in Figure 2.6. Different partitions can reside in different brokers, enabling for multiple consumers to consume from the same topic, while making sure that each message is only provided to one consumer. It is also possible to create multiple copies of a partition, allowing for fault tolerance.

Kafka provides at-least-once delivery guarantee. It is, however, possible for the consumer to achieve exactly-once delivery by removing duplicates, using the built-in Kafka offsets. A producer can decide how many replicas have to receive the message, in order to consider the send operation complete. By setting the `ACKS` variable to `0`, the producer will not wait for a

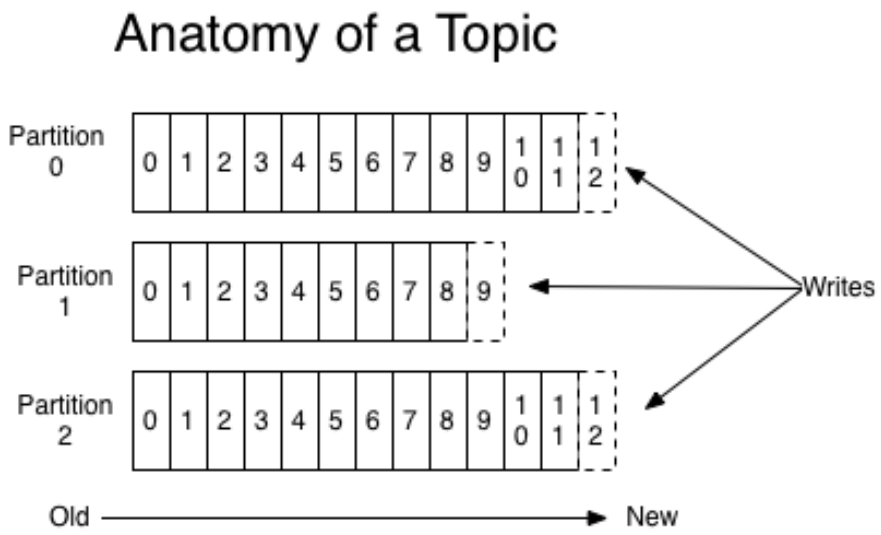


Figure 2.6: Kafka Topics are divided in partitions, which consist of ordered messages. Messages are given a unique sequential id, the *offset*. [2]

reply from the broker. By setting it to `ACKS=1`, it will wait until just one acknowledgment is received, while by setting it to `ACKS=all`, it will wait until all the brokers confirm that they received the message.

ZooKeeper is used for broker synchronization and coordination in a cluster. It updates producers and consumers on information regarding broker availability, load-balancing, and coordination. Kafka also persistently stores messages, which allows the user to index backwards in a topic while still providing high throughput. Instead of flushing the messages to disk, Kafka relies on the operating system to do it in a more efficient manner, improving performance.

Chapter 3

System Architecture and Implementation

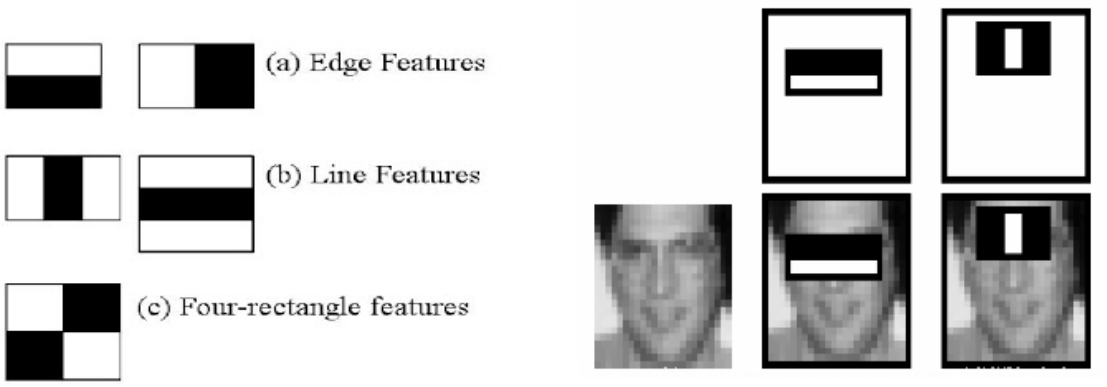
In this chapter, we will present the design of our application. Then, we will discuss the communication and deployment infrastructure that we configured.

3.1 Application design

In order to showcase our work, we decided to create a simple application that would benefit from using an efficient way of communication and cloud deployment. Thus, we needed an application that we could split into multiple microservices. For this reason, we created an app that takes camera frames as input and performs face detection.

In its first phase, the *Camera Service* must use a web camera in order to capture frames. In order to do so, we used the OpenCV [16] library. The application captures a frame every X seconds. At first, X is set at an arbitrary value, which can be modified by the user. Later, we will discuss how this value can change dynamically, in order to maximize efficiency. At the same time, using the *datetime* python library, the current date and time are captured.

In the second phase, the *Processing Service* processes each frame using the *Haar Cascade Classifier*, a machine learning object detection algorithm used to identify objects in an image or video [19]. At first, the algorithm uses a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then, the *Haar features*, demonstrated in Figure 3.1, have to be collected. In order to calculate them, the classifier selects adjacent rectangular regions at a specific location. Subsequently, the algorithm sums



(α) The three basic types of Haar features used for face detection.

(β) The first two features that are selected by the algorithm, which correspond to the regions of the eyes and the nose respectively. [19]

Figure 3.1: Haar Features

up the pixel intensities in each region and calculates the difference between them.

However, out of potentially millions of Haar features found in an image, only a few are actually useful to us. In our case, to perform face detection, the features corresponding to the eyes, the nose, the mouth, and the outline of a person's face are important. In order to separate these, the algorithm uses the *Cascade classifier*, a more accurate and complex method. Instead of applying all of the features at the same time, this method splits them into multiple stages of classifiers and applies them one at a time.

In the third phase, the *Database Service* gathers the information about the number of faces that were spotted, along with the timestamp of the frame, which are combined into a string, and inserts them into a MongoDB [9] database. The data pipeline formed by our application is demonstrated with a diagram in Figure 3.2.

3.2 Application communication

To provide communication between the services in our infrastructure, we used Apache Kafka, since it is a modern solution, focused on maximizing the throughput. Hence, the decision to create an app that produces large amounts of data.

Figure 3.2: Simplified diagram of our data pipeline. The Camera Service captures the camera frames, which are processed by the Processing Service. The output is stored in a database by the Database Service.

3.2.1 Using Kafka to create our data pipeline

First of all, we created a Kafka producer at the Camera Service. Every time a video frame is captured, the frame itself as well as the date and time of the capture are packaged and serialized using the Python *pickle* module. Then, the Kafka producer proceeds to publish the serialized message to a topic that we created, the *Camera topic*, as illustrated in Figure 3.3.

At the other end, the Processing Service receives the messages published to the Camera topic, by using a Kafka consumer, which subscribes to that topic. When the Processing Service consumes a new message, it deserializes it using pickle and then forwards to the face detection algorithm.

In order to improve the scalability of our application, we wanted to have the ability to run multiple instances of the Processing Service at the same time. To do so efficiently, we needed a way to split the message consumption between the different instances of the service. Conveniently, Kafka offers a solution to our problem through the concept of *consumer groups*.

When consumers join a consumer group and subscribe to a certain topic, only one consumer from the group consumes each message from the topic. The messages will effectively be load-balanced over the multiple consumer instances that subscribe to the topic. [4] Hence, by utilizing this mechanism, we gain the advantages of both message queuing and publish-subscribe models.

When the processing is completed, the algorithm outputs the number of faces found in the camera frame. If one or more people are spotted, the application sends this information

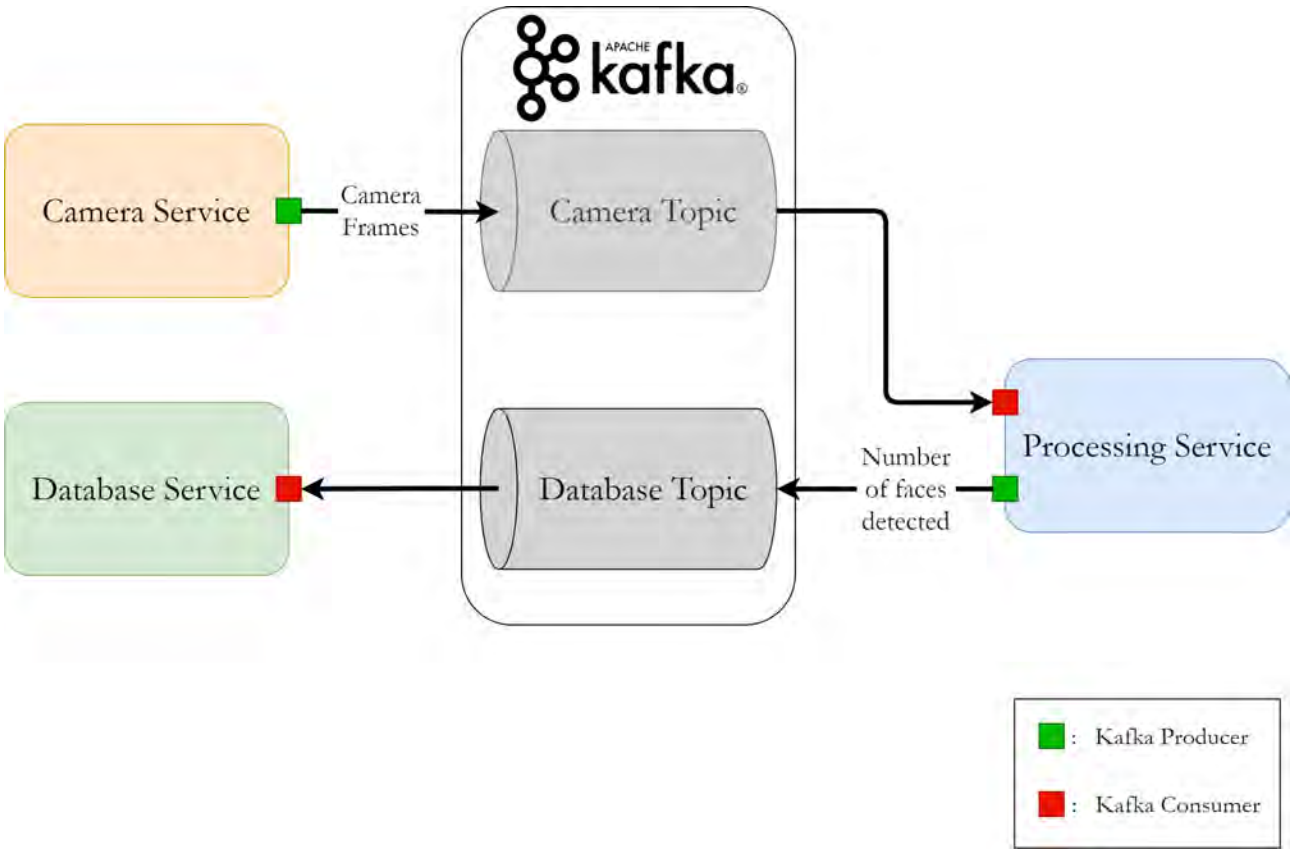


Figure 3.3: Diagram of our data pipeline, including the two Kafka topics that we use.

as well as the timestamp of the frame to the Database Service. This task is carried by a Kafka producer that publishes it to another topic we created, the *Database topic*.

The Database Service, subsequently, has a Kafka Consumer that subscribes to the Database topic. Whenever a new message arrives, the application deserializes the data and creates a new database entry for the event.

3.2.2 Creating a feedback loop to adjust the frame rate dynamically

The data pipeline in its current state is fully functional. We do, however, face a troubling issue regarding the rate with which we produce data (frames in our case). As noted in section 3.1, the amount of frames captured by the Camera Service every second in our application is set arbitrarily by the user, since we do not have a clear indication of what the ideal value is.

This could lead to a number of issues.

First, suppose that the amount of frames we chose is more than our data pipeline can handle because of a bottleneck caused by the Processing Service. In this case, an increasing number of frames would be concentrating on our Camera Topic. However, this is not an issue for Kafka, since it writes the messages to disk and retains them for a long time period, set to 7 days by default. In addition, considering that the data is persistent, the only size limitation is the free space on our hard disk. Despite that, we would have an issue regarding the processing latency. As long as the Camera Service continues producing frames at the same rate, the number of messages stacked in the Camera topic would continuously increase, to a point where it would take days in order to process a frame from the time it was captured.

Additionally, in case our application is bottlenecked because of the network bandwidth, we would face a problem at the Camera Service. More specifically, Kafka producers temporarily maintain the messages produced by the application in a buffer, so that they can send them to the broker in a more efficient manner. As this buffer is of limited size, when the size limit is reached, the new messages will be evicted. This could result in messages getting discarded for a certain amount of time, making our application less consistent.

Finally, if the number of frames we selected arbitrarily is less than the actual capacity of the pipeline, we are wasting bandwidth that we could use to process more frames and potentially identify faces that we would otherwise miss.

To work the aforementioned issues around, we decided to create a feedback loop, so the Processing Service can share information that will help the Camera Service determine an efficient frame rate. This reconfiguration of the frame rate occurs every 5 seconds by default. Every 5 seconds, we count the number of frames processed by the Processing Service. Then, we divide that number by 5, in order to obtain the rate of processing for the frames.

In order to send this information to the Camera Service, we had to create another Kafka producer at the Processing Service. When the frame rate reconfiguration algorithm is triggered, the application produces the frame processing rate to a new topic, the *Feedback topic*. On the other side, the Camera Service has a consumer, which is a subscriber to that topic. When it receives a new message, it proceeds to modify the camera frame rate, so that it matches the rate with which the frames are being processed, as indicated in Figure 3.4.

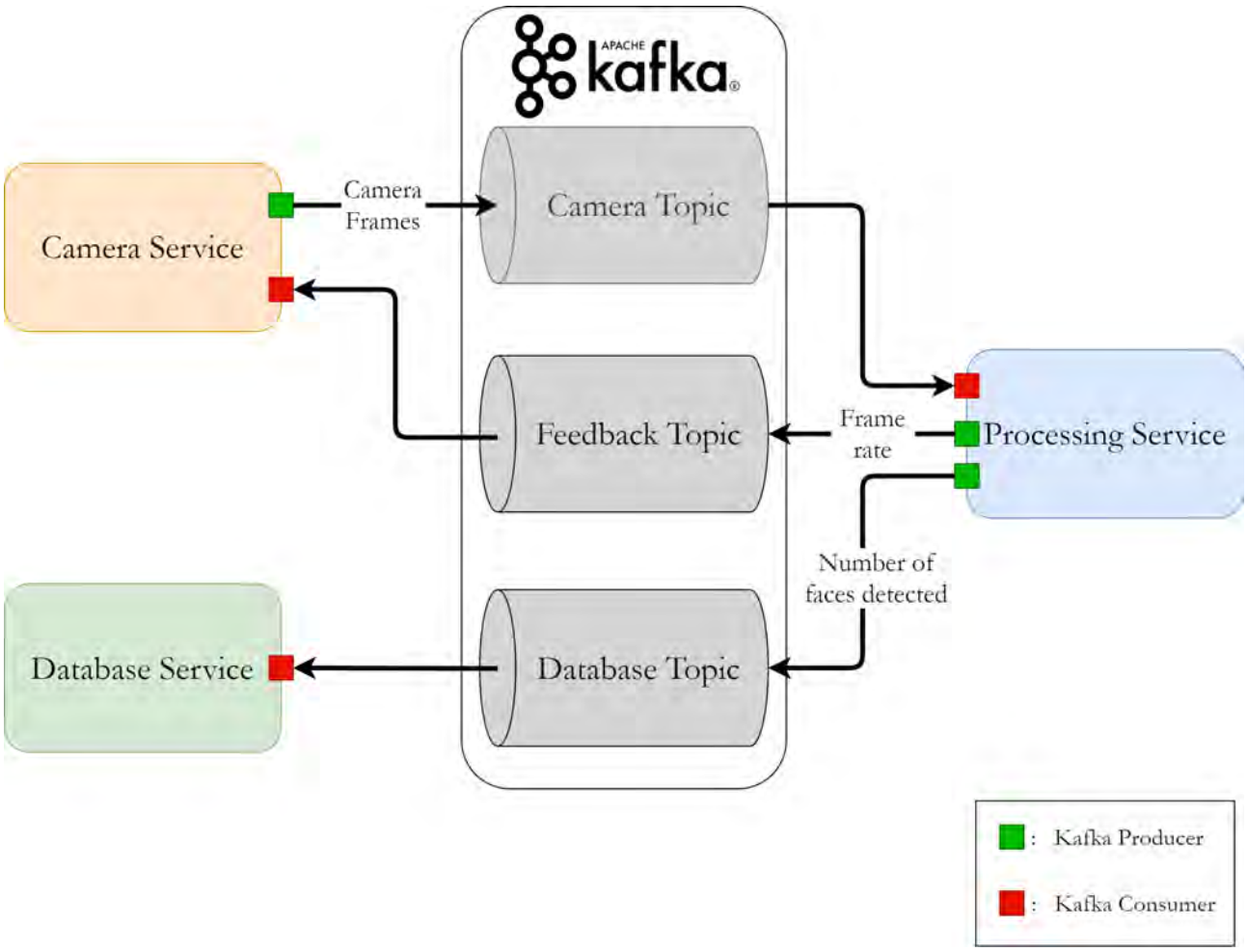


Figure 3.4: Final diagram of our data pipeline, including the Feedback topic used for updating the frame rate.

As this approach can only reconfigure the frame rate to a lower value, we decided to double the camera frame rate every 30 seconds, in order to motivate the system to improve its throughput at the next reconfiguration, in case the network bandwidth has increased.

3.3 Application deployment

Having designed the application, including the communication logic, we then had to deploy our application on an edge-cloud environment. As Figure 3.5 demonstrates, the Camera Service resides on the edge devices, the Processing Service on the fog nodes, and the Database

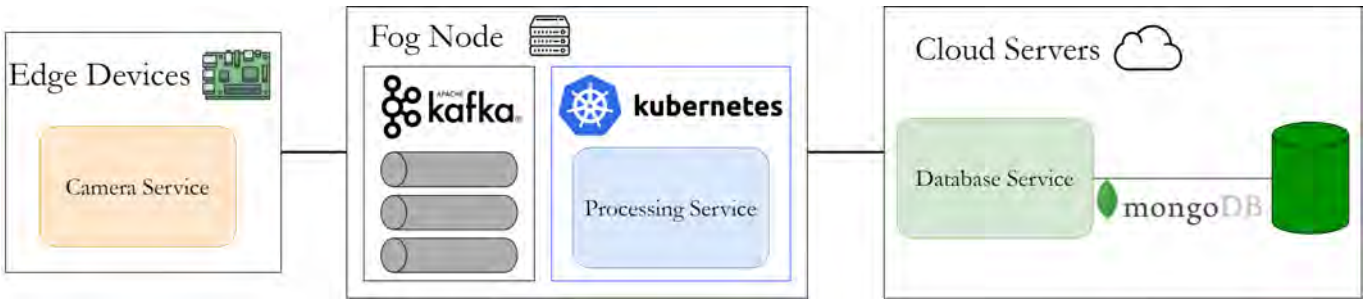


Figure 3.5: The Camera Service runs on the edge devices, the Kafka broker and the Processing Service run on the fog nodes and the Database Service is hosted on the cloud.

Service on the cloud servers. We also thought that the best option would be to place the Kafka and Zookeeper brokers on the fog instead of the cloud, in order to minimize the latency of the messages sent from the edge devices to the fog.

3.3.1 Deploying our application with Kubernetes

Since most of the processing load of our application would be on the Processing Service, we decided to use Kubernetes in order to deploy it on the fog servers to benefit from its scalability and resiliency features. To deploy on Kubernetes, we first had to package the service in a Docker container. To do so, we created a Dockerfile for our container, which includes all the dependencies and the commands required for our program to run. After making sure that the application functions correctly, we pushed the container image to Docker Hub, so that it is ready to be used by Kubernetes.

Before deploying our app, we have to set up our Kubernetes cluster, by running the `kubeadm init` and `kubeadm join` commands. After creating our cluster, we configured it using a YAML file. We configured a Kubernetes deployment in our YAML file instead of configuring a pod since it provides more capabilities we can benefit from. Figures 3.6 and 3.7 show the state of our Kubernetes cluster shortly after we created it.

By using a Kubernetes deployment, we can rollout a *ReplicaSet*, which ensures that a specific amount of pods are up and running at all times. If for example, we choose to have 3 replicas of our application running, the ReplicaSet will maintain them, by observing their

```
(base) akis@y700:~/Documents/k8s/pipeline$ kubectl get all
NAME                                          READY   STATUS    RESTARTS   AGE
pod/face-detection-deployment-8f476b5f8-8j5jl   1/1     Running   0          3m55s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   38m

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/face-detection-deployment   1/1     1            1           3m55s

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/face-detection-deployment-8f476b5f8   1         1         1       3m55s

NAME                                                                  REFERENCE                                     TARGETS   MINPODS   MAXPODS   REPLICAS
horizontalpodautoscaler.autoscaling/face-detection-deployment   Deployment/face-detection-deployment   0%/30%    1         10        1
```

Figure 3.6: Output of the `kubectl get all` command, which lists all Kubernetes resources, such as pods, deployments and services.

state. Whenever a pod crashes, the ReplicaSet will spin up a new one, making sure that 3 pods are active at all times.

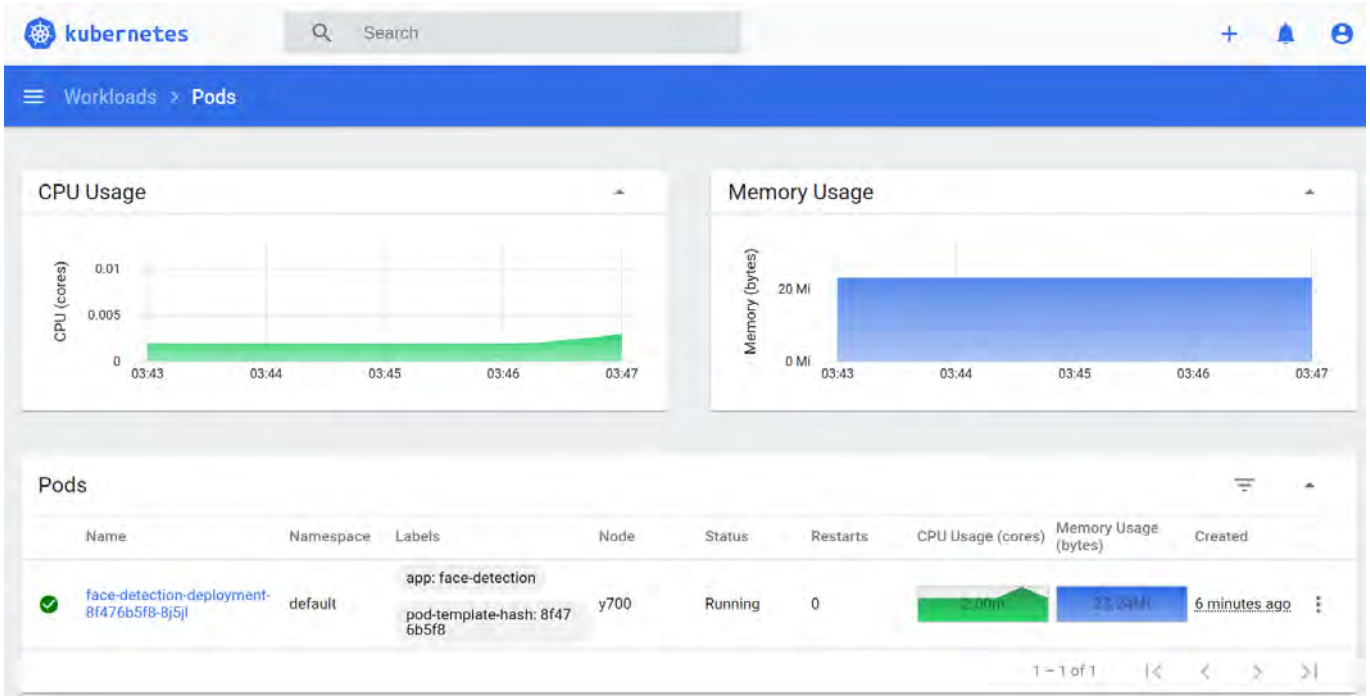


Figure 3.7: Screenshot of the *Kubernetes Dashboard*, which provides a graphical overview of the system. We can see the state of our system under light load.

3.3.2 Kubernetes Autoscaling

Another tool that we benefited from by using a Kubernetes deployment is the Horizontal Pod Autoscaler (HPA). Its job is to automatically scale the number of pods in the deployment based on the CPU or memory usage. When defining an HPA, the user sets the desired CPU or memory target. Kubernetes uses the miliCPU unit when setting CPU targets. For example, a 100m CPU limit corresponds to 10% of a processor core. The user also specifies the minimum and maximum amount of replicas allowed to run at the same time. The desired number of replicas is calculated using the following formula:

`desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]` [13]

The HPA calculates the `currentMetricValue` by taking the average of the values of the respective resource metric (CPU or memory) across all pods that are active. In order to avoid noise that can be affecting the metrics when adding or removing replicas, scaling up the deployment will not happen if any other rescaling process has happened in the last three minutes, whereas scaling down the deployment will not happen if any other rescaling process has happened in the last five minutes. Moreover, in order to take account of the potentially unstable load, the rebalancing process will only take place if the difference between the current CPU or memory usage and the desired one exceeds a specific tolerance. Figures 3.8 and 3.9 display the state of our Kubernetes cluster under heavy load.

```
(base) akis@y700:~/Documents/k8s/pipeline$ kubectl get all
NAME                                                READY   STATUS    RESTARTS   AGE
pod/face-detection-deployment-8f476b5f8-2rm9r       1/1     Running   1          13m
pod/face-detection-deployment-8f476b5f8-7dvw5       1/1     Running   1          9m35s
pod/face-detection-deployment-8f476b5f8-k4w4j       1/1     Running   1          9m35s
pod/face-detection-deployment-8f476b5f8-rvdxx       1/1     Running   1          9m35s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   81m

NAME                                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/face-detection-deployment      4/4     4            4           13m

NAME                                                      DESIRED   CURRENT   READY   AGE
replicaset.apps/face-detection-deployment-8f476b5f8       4         4         4       13m

NAME                                                                    REFERENCE                                     TARGETS    MINPODS   MAXPODS   REPLICAS
horizontalpodautoscaler.autoscaling/face-detection-deployment   Deployment/face-detection-deployment   100%/30%   1         4         4
```

Figure 3.8: Output of the `kubectl get all` command when our system is under heavy load. The Horizontal Pod Autoscaler has increased the number of the pods to 4, as a reaction to the increased load.

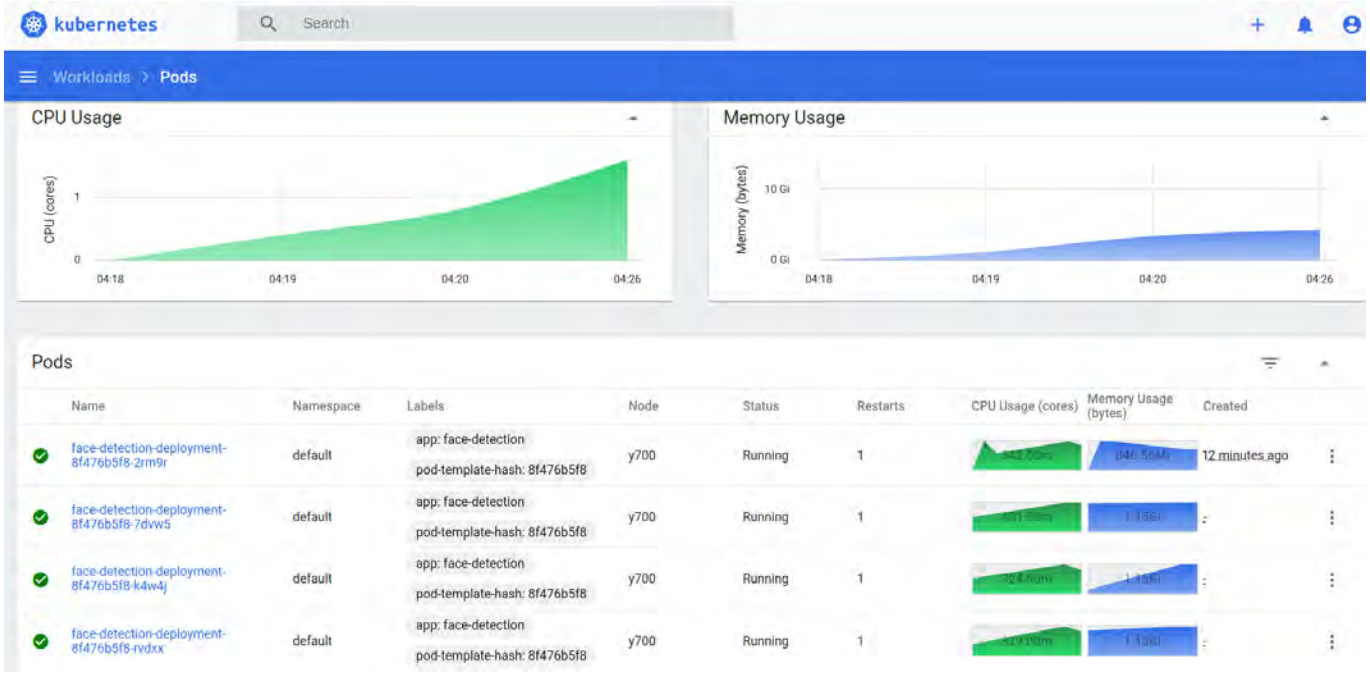


Figure 3.9: Screenshot of the *Kubernetes Dashboard* when our system is under heavy load. We observe that our CPU and Memory usage have both significantly increased.

Chapter 4

Experimental Evaluation

To evaluate the performance of our implementation, we conducted several experiments. We compared different settings for our data pipeline and then, simulated a less optimal edge-cloud environment to observe the performance of our application in a scenario closer to the real world.

4.1 Experimental Setup

4.1.1 Hardware

For testing purposes, we interconnected the computers that we used in our evaluation with a Local Area Network. As the edge device, we used a Raspberry, Pi 3 connected to a web camera that is capable of recording video at a resolution of 640x480 at 30 frames per second. We also used a laptop as the fog node and a desktop as the cloud server. The specifications of these computers are listed in Table 4.1.

4.1.2 Network connection

As shown in Figure 4.1, both the edge and cloud systems were connected to the network via Ethernet cable. Regarding the edge device, we conducted experiments using either an Ethernet cable or Wi-Fi to connect it to the local network.

Table 4.1: Device Hardware Specifications

	Edge Device	**Fog Node**	**Cloud Server**
CPU	ARM Cortex A53 (ARMv8)	Intel i7 6700HQ	Intel i5 4570
Cores (Threads)	4 (4)	4 (8)	4 (4)
RAM	1 GB	8 GB	8 GB
OS	Raspberry Pi OS 10	Linux Mint 19.1	Ubuntu 18.04.4
Linux Kernel	5.4.51-v7+	4.15.0-20	4.15.0-118
NIC spec	100Mb/s	100Mb/s	100Mb/s

To evaluate the performance of our network, we measured the throughput and latency between the edge device (Raspberry Pi) and the fog node (laptop). The results can be seen in Table 4.2. We used the *iperf* tool to measure the bandwidth of our network and the *ping* tool to evaluate the latency of our connection.

Table 4.2: Network performance

Pi connection	**Throughput**	**Latency**
Ethernet	11.3 MB/s	0.511 ms
Wi-Fi	5.17 MB/s	6.429 ms

4.2 System Evaluation

While setting up the communication aspects of our data pipeline, we identified three main Kafka clients for Python, *kafka-python* [7], *pykafka* [17] and *confluent-kafka-python* [3]. We

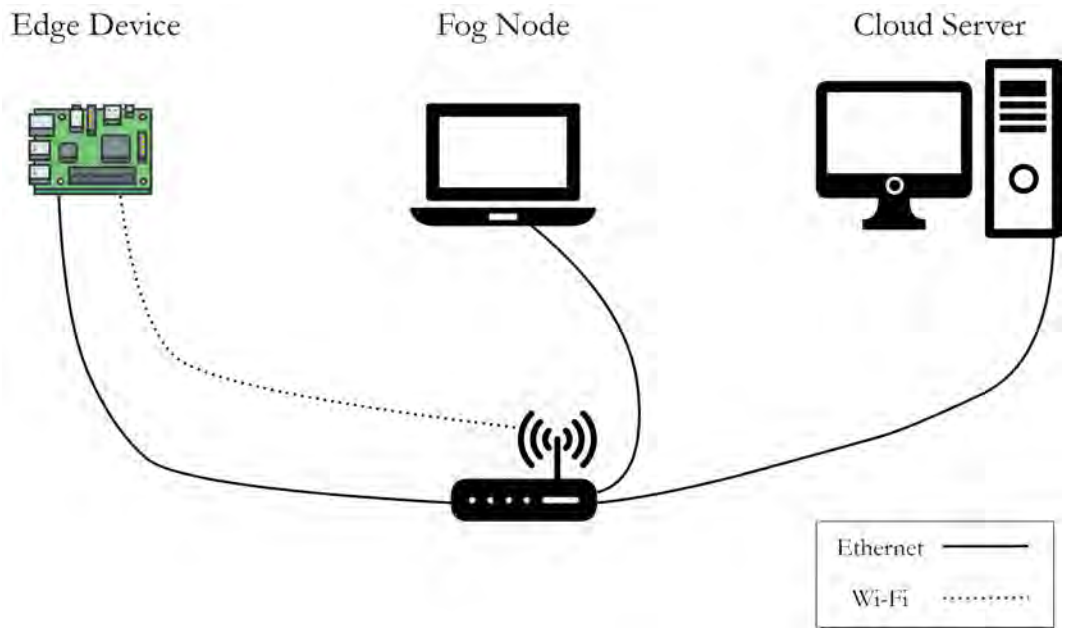


Figure 4.1: Network diagram

decided to try all three of them and compare their performance. The performance was measured by observing the number of messages received by the consumer at the Processing Service within a specific time window (10 seconds). To count the elapsed time, the `time()` function from the `time` Python module was used. Then, we calculated the throughput using the following formula:

$$Throughput = \frac{frames\_received * frame\_size}{time\_elapsed}$$

As shown in Tables 4.3, 4.4 and in Figure 4.2, it was obvious that confluent-kafka-python was, by far, the better performing library. It was capable of reaching the limits of the throughput of our network connection, both when using Wi-Fi and Ethernet. This can be accounted to the fact that confluent-kafka-python is a python wrapper of the *librdkafka* C library. Pykafka and kafka-python on the other hand, while being slower, they emphasize on offering a more user-friendly API. Since this is not an issue for us, we decided to go ahead with the client developed by Confluent.

Table 4.3: Throughput measurements when connecting the edge device to the network using Wi-Fi

Kafka Client	**Mean Throughput (MB/s)**	**Median Throughput (MB/s)**	**Standard Deviation (MB/s)**
confluent-kafka-python	5.118	5.115	0.011
pykafka	1.021	1.023	0.0205
kafka-python	0.43	0.446	0.0476

Table 4.4: Throughput measurements when connecting the edge device to the network using Ethernet

Kafka Client	**Mean Throughput (MB/s)**	**Median Throughput (MB/s)**	**Standard Deviation (MB/s)**
confluent-kafka-python	11.136	11.138	0.009
pykafka	1.056	1.066	0.0215
kafka-python	0.425	0.445	0.0438

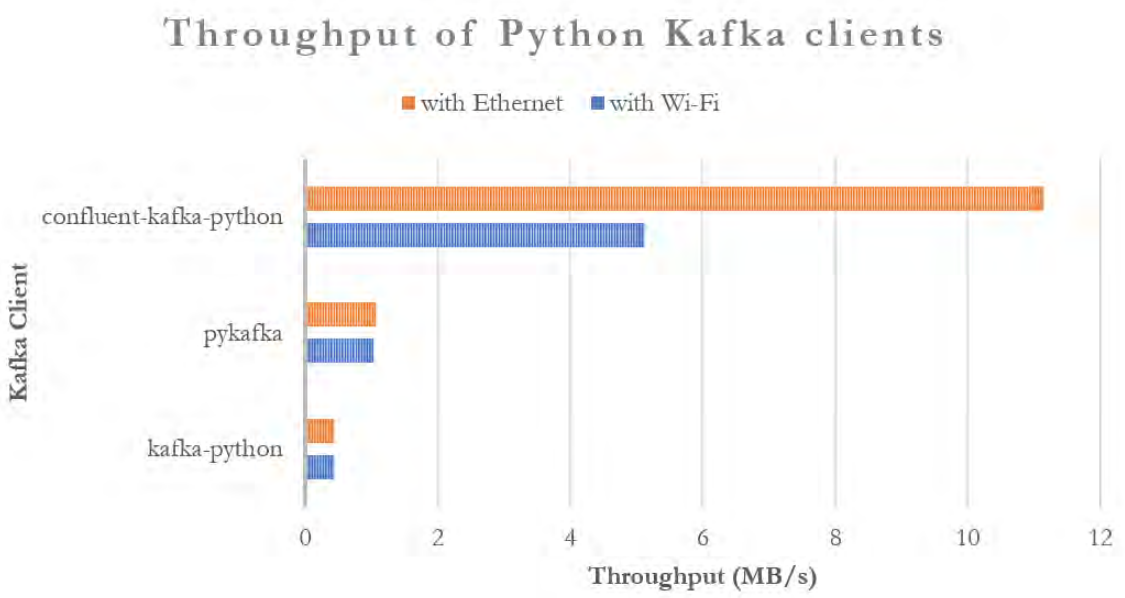


Figure 4.2: Python Kafka client throughput comparison

4.3 Taking advantage of edge computing

To evaluate the potential benefits of edge computing in our data pipeline, we assessed the performance of our application when processing takes place on the fog node and when it takes place on the edge device itself. Since the fog node is more powerful, it is evident that the face detection algorithm will run faster there than on the edge device. The question that needs to be answered, then, is whether the communication overhead of sending all of the frames over the network, from the edge device to the fog, will overcome the processing deficit of running the algorithm on the edge device.

As demonstrated in Figure 4.3 and in Tables 4.5 and 4.6, using the fog nodes improves the overall time required to process a frame. More specifically, when the edge device is connected via Wi-Fi we experience a speedup of 2.6x, while when it is connected via Ethernet we experience a speedup of 4.34x. Thus, as long as the network connection between the edge and the fog is fast enough, we can benefit significantly from using the power of the fog nodes. In most cases, applications will run much faster on fog hardware, especially when there is the possibility of using GPUs to accelerate the performance.

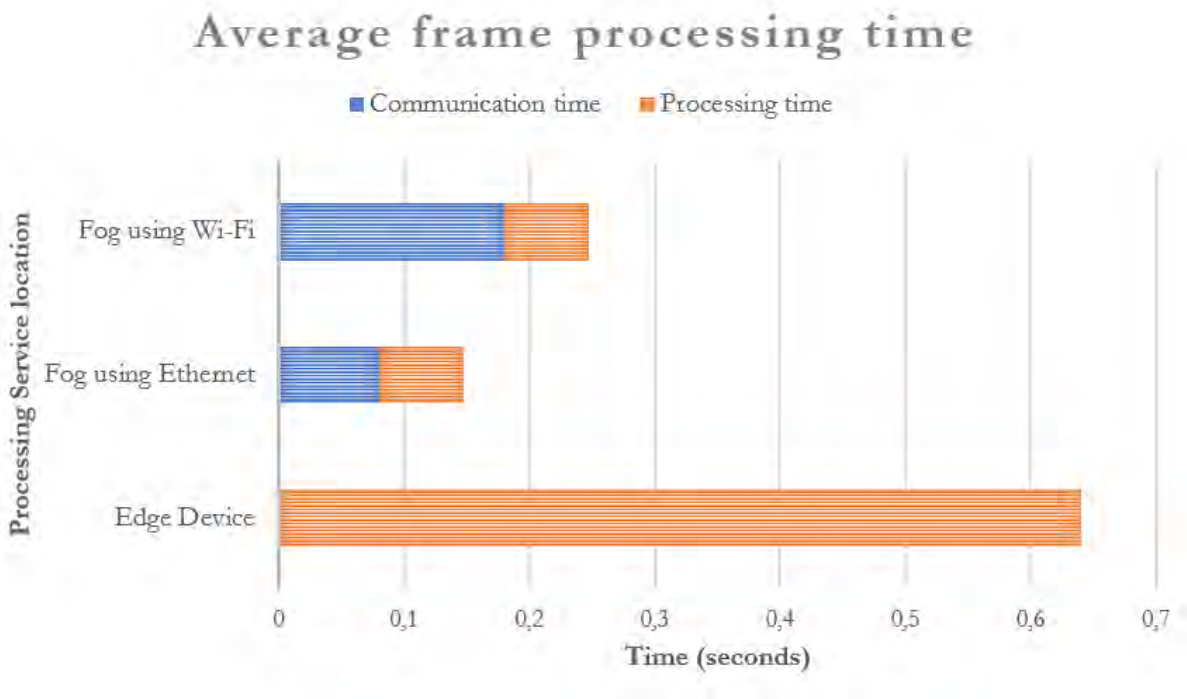


Figure 4.3: Total frame processing time measurements, including both communication and processing time

Table 4.5: Communication time

Processing Service location	**Mean Time (sec)**	**Median Time (sec)**	**Standard Deviation (sec)**
Edge Device	0	0	0
Fog using Ethernet	0.0796	0.0784	0.0083
Fog using Wi-Fi	0.1795	0.1788	0.0071

Table 4.6: Processing time

Processing Service location	Mean Time (sec)	Median Time (sec)	Standard Deviation (sec)
Edge Device	0.6394	0.6237	0.0614
Fog Node	0.0676	0.0509	0.0314

4.4 Evaluating our application in more realistic conditions

To further test our application, we used the ns-3 network simulator [14]. We tested the data pipeline with various latency and packet loss values to evaluate its behavior in non-perfect real-world conditions. To configure the ns-3 simulation environment as shown in Figure 4.4, we first set up some virtual interfaces. Then, the physical Ethernet interface of our machine was connected to the first virtual interface using an Ethernet bridge. Thus, the packets received by our host were forwarded through the interfaces to the simulator. Using the simulator, we created an emulated link, which we then used to tamper with the quality of the connection.

Firstly, we started monitoring the throughput of our system while progressively increasing the amount of latency in our emulated network connection. As can be seen in Figure 4.5 and in Table 4.7, while latency is increased until it reaches 250ms, the throughput is decreasing at a relatively steady rate. For values over 250ms, however, that rate is substantially increased.

Then, we used the ns-3 simulator to simulate packet loss in our network connection. As shown in Figure 4.6 and in Table 4.8, the throughput decreases as we increase the packet loss rate. For values greater than 10%, the throughput starts decreasing at a significantly higher rate.

We should note that in both tests, despite the throughput reduction, our data pipeline was functional even at high latency. The consumer still received messages in the same order that they were published, as a result of the at-least-once delivery guarantees provided by Kafka.

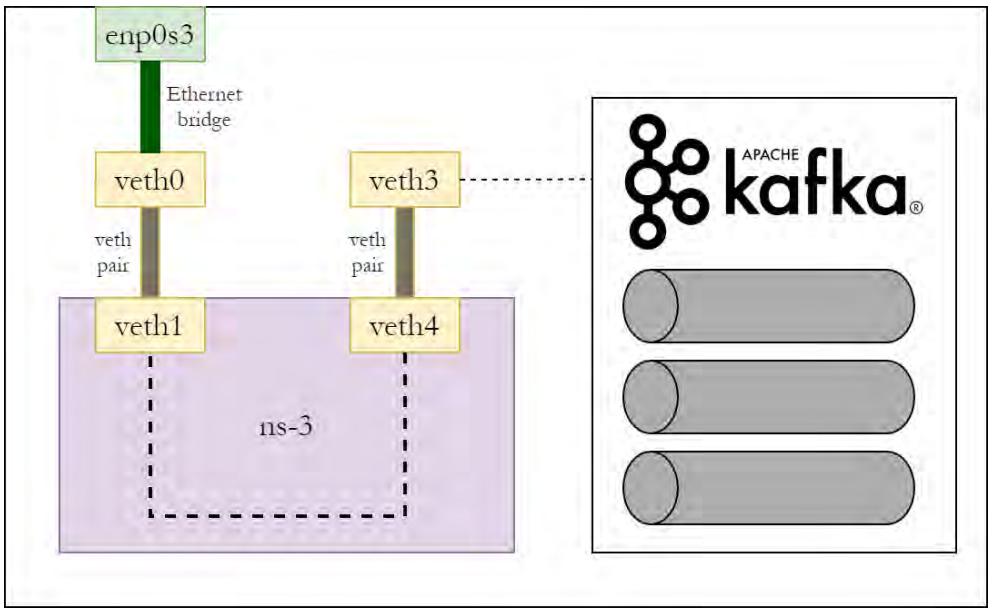


Figure 4.4: ns-3 simulation setup

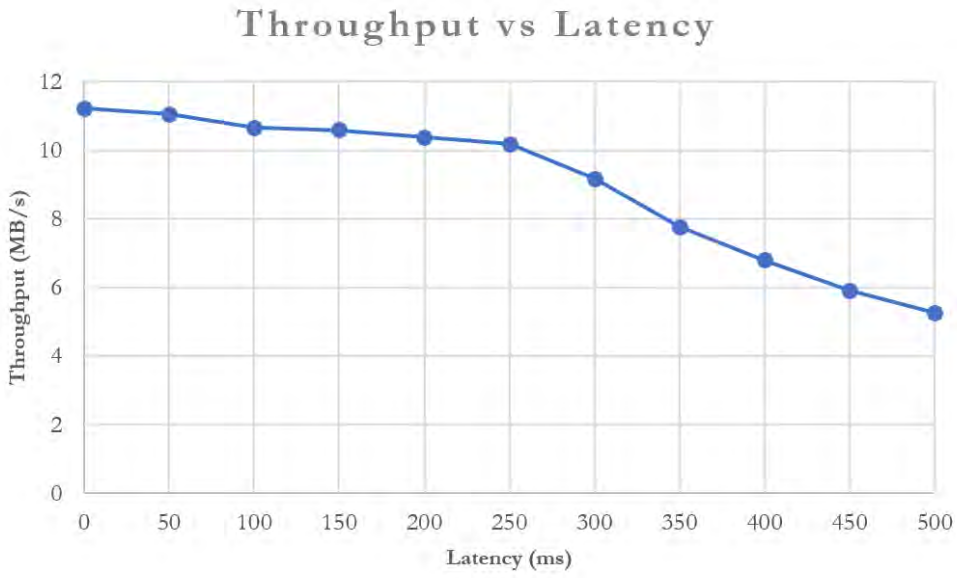


Figure 4.5: Throughput measurements with various amounts of latency

Table 4.7: Throughput measurements with various latency values

Latency (ms)	**Throughput (MB/s)**
0	11.219
50	11.051
100	10.673
150	10.589
200	10.379
250	10.184
300	9.163
350	7.773
400	6.798
450	5.919
500	5.266

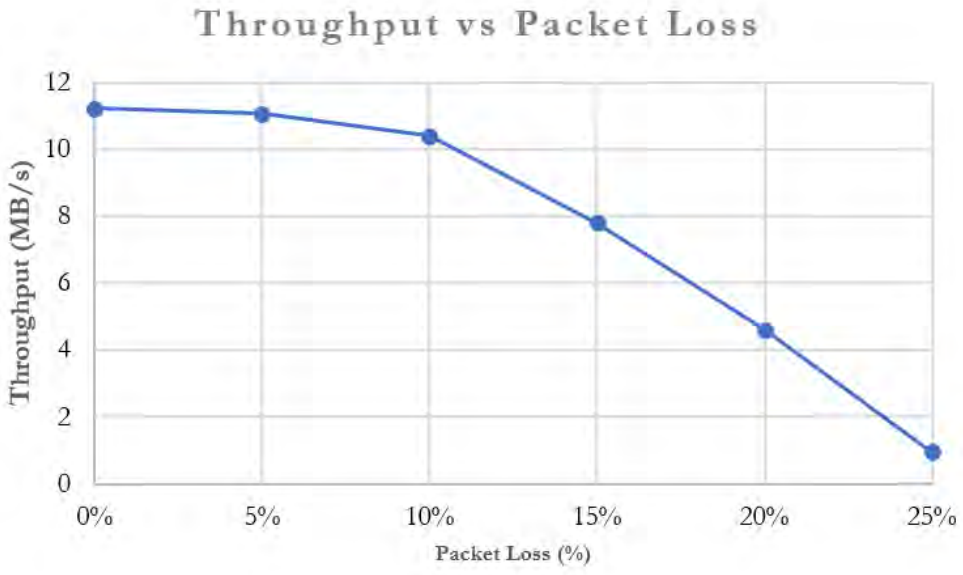


Figure 4.6: Throughput measurements with various amounts of packet loss

Table 4.8: Throughput measurements with various amounts of packet loss

Packet Loss (%)	**Throughput (MB/s)**
0%	11.228
5%	11.087
10%	10.357
15%	7.794
20%	4.614
25%	0.946

Chapter 5

Conclusion

We experimented with deploying applications on edge-cloud environments. To achieve communication between the different services of applications we have used Kafka, because of its scalability, high throughput, and fault-tolerance. We have also used Kubernetes to create and manage the deployments, since it provides useful features such as self-healing and auto-scaling. Moreover, we have created an application based on the micro-services architecture, in order to test and evaluate our infrastructure. The results of our evaluation show that our system can utilize the edge-cloud architecture efficiently, increasing the performance of applications.

In the future, we can investigate the performance of our such hierarchical edge/fog/cloud deployments with more applications in order to achieve a more thorough understanding of its capabilities and disadvantages. It would be interesting to try applications with various characteristics, such as high or low CPU and memory usage and applications that have to communicate using messages of various sizes. Furthermore, we can use the Kafka Connect API to stream data between Kafka and other applications efficiently. We can also utilize the Kafka Streams Java API for real-time processing in applications that interact heavily with Kafka topics. Last but not least, we can also deploy our infrastructure at a larger scale in order to perform more tests regarding Kubernetes' scalability features. By using a computer cluster as the fog node as well as a public cloud provider or private cloud infrastructure, we can examine its performance more thoroughly under real-world conditions.

Bibliography

[1] Apache Foundation. Apache Kafka, 2020. URL https://https://kafka.apache.org/.

[2] Apache Foundation. Apache Kafka Documentation, 2020. URL https://kafka.apache.org/081/documentation.html.

[3] Confluent. Confluent-Kafka-Python client, 2020. URL https://github.com/dpkp/kafka-python.

[4] Confluent. Introduction to Apache Kafka, 2020. URL https://docs.confluent.io/current/kafka/introduction.html.

[5] Docker. What is a Container?, 2020. URL https://www.docker.com/resources/what-container.

[6] Docker. Docker Overview, 2020. URL https://docs.docker.com/get-started/overview/.

[7] dpkp. Kafka-Python client, 2020. URL https://github.com/dpkp/kafka-python.

[8] IBM Cloud Education. What is Docker?, 2020. URL https://www.ibm.com/cloud/learn/docker.

[9] M. Inc. MongoDB, 2020. URL https://www.mongodb.com/.

[10] Kim Wuestkamp. Kubernetes Istio simply visually explained, 2020. URL https://itnext.io/kubernetes-istio-simply-visually-explained-58a7d158b83f.

[11] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[12] Kubernetes. Kubernetes, 2020. URL `https://kubernetes.io/`.

[13] Kubernetes. Kubernetes - Horizontal Pod Autoscaler, 2020. URL `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`.

[14] nsnam. Network Simulator version 3, 2020. URL `https://www.nsnam.org/`.

[15] OmniSci. Fog Computing, 2020. URL `https://www.omnisci.com/technical-glossary/fog-computing`.

[16] OpenCV. OpenCV, 2020. URL `https://opencv.org/`.

[17] Parsely, Inc. PyKafka client, 2020. URL `https://github.com/Parsely/pykafka`.

[18] Red Hat. What is Kubernetes?, 2020. URL `https://www.redhat.com/en/topics/containers/what-is-kubernetes`.

[19] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, volume 1, pages I–I. IEEE, 2001.

[20] Vladimir Kaplarevic. Understanding Kubernetes Architecture With Diagrams, 2020. URL `https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams`.

[21] Wikipedia contributors. Apache kafka — Wikipedia, the free encyclopedia, 2020. URL `https://en.wikipedia.org/wiki/Apache_Kafka`.