Mälardalen University
School of Innovation, Design and Engineering
Västerås, Sweden

Thesis for the Degree of Bachelor in Computer Science 15.0 credits

ACHIEVING A REUSABLE REFERENCE ARCHITECTURE FOR MICROSERVICES IN CLOUD ENVIRONMENTS

Zacharias Leo
zlo16001@student.mdh.se

Examiner: Jan Gustafsson
Mälardalen University, Västerås, Sweden

Supervisors: Václav Struhár, Radu Dobrin
Mälardalen University, Västerås, Sweden

Company supervisors: Niclas Tenggren, Andreas Stenlund
Enfo Sweden, Västerås, Sweden

June 25, 2019

Abstract

Microservices are a new trend in application development. They allow for breaking down big monolithic applications into smaller parts that can be updated and scaled independently. However, there are still many uncertainties when it comes to the standards of the microservices, which can lead to costly and time consuming creations or migrations of system architectures. One of the more common ways of deploying microservices is through the use of containers and container orchestration platform, most commonly the open-source platform Kubernetes. In order to speed up the creation or migration it is possible to use a reference architecture that acts as a blueprint to follow when designing and implementing the architecture. Using a reference architecture will lead to more standardized architectures, which in turn are most time and cost effective.

This thesis proposes such a reference architecture to be used when designing microservice architectures. The goal of the reference architecture is to provide a product that meets the needs and expectations of companies that already use microservices or might adopt microservices in the future.

In order to achieve the goal of the thesis, the work was divided into three main phases. First, a questionnaire was conducted and sent out to be answered by experts in the area of microservices or system architectures. Second, literature studies were made on the state of the art and practice of reference architectures and microservice architectures. Third, studies were made on the Kubernetes components found in the Kubernetes documentation, which were evaluated and chosen depending on how well they reflected the needs of the companies.

This thesis finally proposes a reference architecture with components chosen according to the needs and expectations of the companies found from the questionnaire.

Table of Contents

1 Introduction **1**
1.1 Problem Formulation . . . 1
1.1.1 Research questions . . . 1
1.1.2 Limitations . . . 2

2 Background **3**
2.1 Reference Architectures . . . 3
2.2 Cloud Services . . . 3
2.3 Microservices and containerization . . . 3
2.4 Software Architectures . . . 4
2.4.1 Monolithic Architecture . . . 5
2.4.2 Microservice Architecture . . . 5
2.5 Kubernetes . . . 7
2.6 Non-functional requirements . . . 8

3 Related Work **9**
3.1 Microservice bad practices . . . 9
3.2 Security reference architecture . . . 9
3.3 Microservice reference architecture . . . 9

4 Method **11**
4.1 Questionnaire . . . 11
4.2 Literature study and initial reference architecture design . . . 11
4.3 Component evaluation . . . 11

5 Ethical and Societal Considerations **13**

6 Company expectations and views on microservices **14**

7 Microservice and reference architecture pattern studies **14**
7.1 Reference architecture design practices . . . 14
7.2 Microservice architecture design . . . 15
7.2.1 Microservice guidelines . . . 15
7.2.2 Containerized microservice design patterns . . . 16
7.2.3 The initial architecture . . . 18

8 Exploration of the Kubernetes components **18**
8.1 Container runtimes . . . 18
8.2 Monitoring . . . 19
8.2.1 Resource metrics pipeline . . . 20
8.2.2 Full metrics pipeline . . . 20
8.3 Logging . . . 21
8.3.1 Basic logging . . . 21
8.3.2 Node level logging . . . 21
8.3.3 Cluster level logging . . . 21
8.4 Deployment statistics . . . 22
8.5 Micro-gateways . . . 22

9 Results **24**
9.1 Customer views and expectations on microservices . . . 24
9.2 Component analysis . . . 25
9.2.1 Container runtimes . . . 25
9.2.2 Monitoring . . . 26
9.2.3 Logging . . . 26
9.2.4 Deployment statistics . . . 26

9.2.5 Micro-gateways 26
9.3 The proposed microservice reference architecture 28
9.4 Description of operation 28

10 Discussion **30**

11 Conclusions **31**
11.1 Future work 32

References **36**

Appendix A Questionnaire **37**
A.1 Section 1 37
A.2 Section 2 37
A.3 Section 3 37
A.4 Section 4 37
A.5 Section 5 38

List of Figures

1 Containerized applications using Docker. Source: [1] 4
2 Applications running in virtual machines. Source: [1] 4
3 A visual comparison of microservice and monolithic architectures. 6
4 An overview of the Kubernetes architecture. 8
5 Example of a sidecar pattern for a web server. Source: [2] 16
6 An example of an ambassador container implementation. Source: [2] 17
7 An example of an adapter container implementation. Source: [2] 18
8 The initial Kubernetes-based microservice architecture. The red colored components are yet to be chosen. 18
9 Memory usage for each runtime. 25
10 CPU usage for each runtime. 25
11 The Kubernetes cluster architecture with the components added. 28

1 Introduction

Microservices are a current trend in a server-side enterprise application development. It allows to decompose software into small, well defined modules and thus improve reusability, maintainability and scalability. However, implementation of a software solution using microservices is not straightforward and it needs many considerations, e.g., selecting the suitable platform, components and architecture of the application. In order to fully utilize the scalability of microservices it is beneficial to deploy them in a cloud.

Microservice architectures are becoming important for software companies that wants to stay competitive within their line of business. In order to to this, frequent updates needs to be done to their products, but the traditional way of building software severely limits the flexibility of updates and maintenance. Microservices however allow for frequent updates and maintenance by allowing each microservice to be updated independently of the rest of the application, allowing more teams to work in parallel without having to schedule updating times. However, the creation of a microservice architecture is not to be taken lightly. It requires a different way of thinking compared to the traditional way of software architectures, often leading to the migrations and creations of these architectures being long and costly processes.

The thesis presents a case study on identifying a standardized way of building container-based microservice architectures using the Kubernetes platform based on business expectations of companies, leading to faster production times. This would result in a "blueprint" describing how the architectures (or parts of them) should be built to fulfill a number of non-functional requirements. With more things in common, maintenance of these architectures can partly be automated to speed up the maintenance time significantly. The thesis suggests a number of Kubernetes components that fit the reference architecture proposed in this thesis. The components handle *container runtimes*, *monitoring*, *logging*, *deployment statistics* and *micro-gateways*.

The following three goals are the core of this thesis:

- Identify non-functional requirement expectations (scaleability, traceability, etc.) on microservice architectures through interviews with companies planning to utilize such architectures.
- Explore the possibility of and identify an administrable architecture solution that fulfills non-functional requirements, such as maintainability, traceability, modifiability, etc..
- Identify Kubernetes components offering functionality aligned with the needs and expectations discovered. Explore the capabilities, limitations and compatibility issues of the components.

The thesis is done in cooperation with Enfo. Enfo is a consulting company that offers services that involve microservices in cloud environments. The company is trying to introduce a reusable microservice based architecture that could act as a standard when building future microservice architectures. The company provided a contact with their customers willing to perform interviews for identifying sought after qualities in such architectures, or to learn from their experience if they have already made the migration between architectures.

1.1 Problem Formulation

There is no standardized or commonly used way of designing a microservice architecture (using any microservice platform including Kubernetes), which mostly leads to each architecture being custom made without much resemblance of others. The problems with custom built architectures are that experience can not be transferred between projects, leading to longer production time and more error prone end products. Maintaining multiple architectures that are distinct from each other also take more time than maintaining architectures that are similar, which can often be done through automation.

1.1.1 Research questions

The research questions of this thesis, in the context of reusable microservice architectures, are:

- *What are the key expectations on non-functional requirements of companies on a microservice based architecture?*

- *What are usually the core features implemented into reference architectures, and how can they fit in the proposed microservice reference architecture?*
- *Which Kubernetes components should be used in the proposed reference architecture, based on the company expectations?*

1.1.2 Limitations

With a limited time frame of only 10 weeks it is not possible to evaluate every Kubernetes component available. To stay within the time frame, this thesis only explores the components suggested, or mentioned, in the Kubernetes documentation.

There are many practices when it comes to deploying microservices. Commonly used techniques are virtual machines, containers or by the serverless approach where the application is broken down into functions that are stored at a cloud service provider available to be used as needed. This thesis is limited to the studies on the container-based approach.

The timeframe also limits the amount of detail that can be given to the reference architecture. To stay within the time frame this thesis focuses on the Kubernetes-based microservice part of the architecture. A complete reference architecture is outside the scope of this thesis. A microservice design is proposed together with its corresponding Kuberentes components.

The reference architecture is designed with a cloud based enterprise domain in mind, meaning a large company system. Reference architectures are often evolved over time with the involvement of many experts in the area. However, this work is done over a short time without access to experts to validate or give input, meaning the microservice reference architecture lay the foundations for future improvements. The thesis also provides some suggestions on how to continue the development of the reference architecture.

2 Background

In the last section an introduction was given to the area relevant in this thesis work. This section provides a more detailed description of the technologies that was mentioned in the introduction and are discussed further into the thesis. It describes the fundamentals of reference architectures, cloud services, software architectures, Kubernetes, microservices and containerization, as well as non-funtional requirements.

2.1 Reference Architectures

Today it is critical for many companies to quickly develop and maintain many systems. Without any guidelines this process can be hard and time consuming, resulting in many complex custom built systems. Standardization is important to shorten the time required to efficiently develop and maintain systems. Standardization results in a common way of building something, meaning it is easier to understand for developers and experience can be transferred between projects. This is solved by using reference architectures that function as templates on how to develop the fundamentals of systems by providing best practices, guidelines, documentation and technology proposals. Making use of reference architectures can lower the complexity of systems by reducing diversity, leading to lower costs of development and maintenance. However, proper documentation is needed to fully utilize reference architectures [3]. Reference architectures are designed to be abstract, therefore they typically do not provide much details on implementation.

2.2 Cloud Services

Cloud services are quickly becoming an important part for companies of all sizes [4]. Cloud services enable the companies to store their data and perform calculations at a third party cloud service, such as Microsoft Azure or Amazon's AWS (Amazon Web Services). By using a third party cloud service the companies eliminate the need for an on-site data storage and local computation power and therefore they do not have to worry about managing their data storage and hardware, which is instead handled by the cloud service provider. Many services offer a "pay-as-you-go" cost plan which means that you only pay for what you use, a very flexible payment plan that is useful for growing companies or companies with varying data usage. Cloud services allows for scalability since the companies themselves does not have to invest in their own storage servers and instead specifies to the service provider how much storage they will need [4]. This approach is known as "serverless", which basically means that the storage servers are abstracted from the companies themselves [5]. This is a very good alternative for new small-scale companies that does not have the money to invest into server hardware, or the knowledge on how to handle it, letting them focus more on their area of expertise.

However, there may be disadvantages when using cloud services. Since the servers are hidden from the companies utilizing the services they can not see what is happening on the server side, which can make troubleshooting hard. It can be hard to tell whether problems occur locally at the company location, or if the problems occur on the server side. Security might also be a concern when leaving data into the hands of a third party, trusting them to handle it correctly. There are many security aspects that should be taken into account when migrating to a cloud service, for example data loss or theft [6].

2.3 Microservices and containerization

Microservices are a new addition to the software development scene. The idea was first presented by James Lewis in 2012 at a San Francisco conference [7]. One of the first companies to adopt the idea on a large scale was Netflix, who then called it "fine-grained service oriented architecture (SOA)" [8]. With "fine-grained SOA" they meant a service that is broken down into smaller parts where each part specializes on their given task, for example handling user input and no more. Microservices has since become more widely used by software developers in both small and big companies, like Amazon and Ebay [9] who rebuilt their entire architectures into microservice based architectures.

Microservices allows for breaking down big applications into smaller parts working independent of each other. The reasons for splitting an application into multiple microservices is mainly to improve maintainability, scalability, reusability and reliability [9].

Earlier attempts to avoid big and complex applications include object-oriented programming (OOP) and service-oriented architecture (SOA) [10]. OOP allows for encapsulating distinct objects present in the application to simplify it and make it more maintainable by limiting the dependencies between these objects. The encapsulation of the objects allow for maintaining them in isolation, as well as enabling them to be reused [11]. However, practicing OOP on a monolithic architecture still limits the scalability of the application because the objects can not be scaled independently. SOA came up with the idea of splitting up the monolithic applications into distributed loosely coupled services that worked together to perform tasks [10]. The idea of microservices are built from these earlier approaches, some people claim that microservices and SOA are the same thing. However, the main difference between SOA and microservices is that microservices are smaller, they share no data storage with each other and they are uncoupled [12].

Microservices can be stored in different ways depending on the practice used. Either multiple of them can be stored inside a host, a virtual environment; they can be stored by themselves inside a virtual machine or a container; or they can be handled the serverless way, where the microservices are broken down into functions and deployed to a third party cloud provider available to be used as needed [13, p. 56-62]. This thesis focuses on the container approach where Kubernetes is used to coordinate these containers. In the case of container approach the microservices are stored, isolated from each other, in containers together with all the parts they need to run, for instance the code, libraries, dependencies and libraries. The containers operate in their own virtual environment with it's own file system without connection to the computer's underlying hardware or file system. This disconnection makes the containers portable across different operating systems and clouds [14]. The most well known and widely used container technology is Docker [15]. Docker containers functions like virtual machines, but without a complete virtual operating system, instead Docker uses the operating system already existing on the computer. This makes the containers much more lightweight than virtual machines, taking up less space and memory, as well as being faster to boot [16]. See Figure 1 and Figure 2 for a comparison.

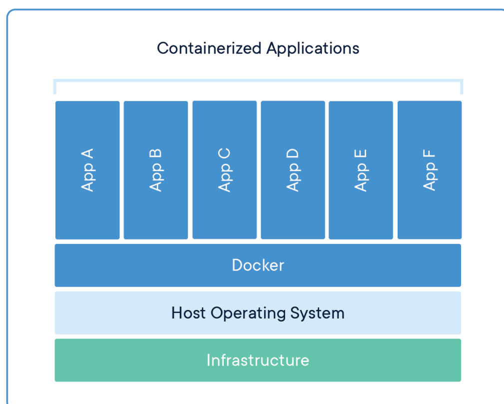

Figure 1: Containerized applications using Docker. Source: [1]

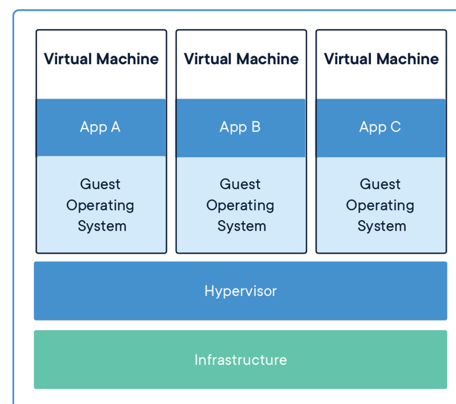

Figure 2: Applications running in virtual machines. Source: [1]

2.4 Software Architectures

There are many ways of designing the architectures for software. When computation and storage is handled using a cloud, it is often called "cloud computing architectures". This section will focus on the two most distinct architectures.

2.4.1 Monolithic Architecture

Monolithic architectures is the traditional way of designing and building applications. These architectures are usually built in three parts: a user client such as a website interface, a database and the application itself. In monolithic architectures the application is one entity that do all the work including handling data, executing functions, handling user requests, etc. [17].

Applications have for a long time been built using monolithic architectures, therefore standards and guidelines exist describing how it should be done. These guidelines helps eliminate a lot of the planning stage and therefore speeds up the development process. The monolithic architectures can work very well, however they also have a couple of notable downsides, which is the reason why many big companies move away from this architecture style. The problems of the monolithic architectures appear when it grows into a big application [17]:

- All the code is in a single code base/repository, which makes it very big and complex. It is hard to make changes and update the application when the code is big, and there might be uncertainties whether some changes affect other unintended parts [17].
- Scaling the application is inefficient, since you can not scale parts of the application without scaling the rest. Therefore scaling the application will lead to inefficient use of resources [17].
- Updating and making changes to the application is troublesome because changes can only be made by updating and restarting the application, which will disrupt all the users. For this reason it is hard to assign teams of developers to different parts of the application, since they can not update their individual parts without waiting for the other teams to be ready to deploy as well [17].
- If one part of the application fails or crashes all of it will be affected and crash. [17].
- The application is stuck with the original technology, unless all of it is rebuilt. It takes time and costs money to rebuild, which is often not worth it from a business perspective [17].

As mentioned in Section 2.3, there have been different attempts to try to avoid the downsides of the monolithic architectures. SOA and OOP have been used to increase maintainability, reusability and decrease coupling. Other widely used approaches to increase reusability are the standardized libraries, such as *stdio.h* [1], the C library that handles input and output. However, these libraries were designed specifically to be reused. When developing a monolithic application the coupling between functions often become too complex and hinders future reuse of them, unless they are specifically designed to be reused. The next section describes how microservices avoids some of the disadvantages that comes with monolithic architectures by using the ideas of low coupling, SOA and OOP.

2.4.2 Microservice Architecture

Monolithic applications work well on smaller applications, but as they grow in size and become more complex they become harder to maintain. In this case it is more suitable to make use of a microservice architecture. The microservice architecture, unlike the monolithic, breaks down the big application entity into smaller services that are put into containers. This approach eliminates some of the drawbacks of the monolithic architecture [17]:

- Each microservice is small and specializes on one task, making it easy to understand and maintain and not as daunting as a big and complex code [17].
- Each microservice specializes on one task and has low coupling with other microservices, therefore they can be reused multiple times, possibly over multiple applications [18].
- Each microservice can be scaled independently of the others, allowing for more efficient use of resources [17].
- No commitment is needed for technology, each service can use it's own technology and the developers can choose what programming language and tools they want to use when creating a service. However, this should be regulated somewhat in order to not use too much technology variation in one application [17].
- Updating and redeploying the microservices is fast. Different teams can work on different parts of the application and update services independently of the others. This allows for assigning different teams to different tasks on the application [17].

[1] https://www.gnu.org/software/m68hc11/examples/stdio_8h-source.html

- If a memory leak or fault occurs in a service it will be isolated to that service only and the rest of the application is not affected [17].

However, like in the monolithic architecture there are some drawbacks. Even though it is easy to update and deploy the services on their own, it is much more complex to maintain a large number of them. When designing a microservice architecture there are many things that needs to be considered and thought of in a different way than the traditional way. For instance there are different approaches on how the microservices should be split up. All of this needs careful coordination and planning and will therefore take longer time to develop than the monolithic architecture. Documentation needs to be done in order to keep track of, updating and maintaining the microservices [17].

These drawbacks make the development of these systems take longer and the complexity of it might be discouraging to developers. The advantages of a microservice architecture only show when an application is of a large size and all the setup and documentation is finished, otherwise a monolithic architecture might work just as well with less development time invested into it.

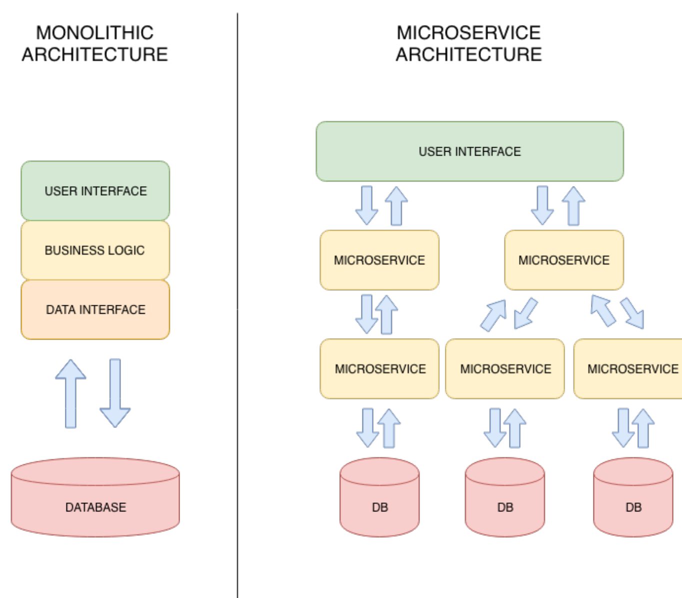

Figure 3: A visual comparison of microservice and monolithic architectures.

2.5 Kubernetes

The microservices are isolated from each other in a distributed system without shared data. In order to control the microservices a container orchestration platform is needed. Kubernetes is one of the leading platforms available for this purpose. It began as a small project at Google called Borg [19]. In 2014 it was released together with the Linux Foundation as an open source project under the name Kubernetes. With Kubernetes it is possible to manage and organize the containers in predictable ways. Kubernetes provides a set of objects used to build the architecture:

- **Pods:** Pods are the basic objects of the Kubernetes architecture. They consist of one or more containers that share storage and network. Each pod is assigned its own unique IP address which allows the applications to use ports without conflicts. Containers that are placed in the same pod can communicate with each other over localhost, but in order for them to communicate with containers inside different pods they need to use that pods' IP address [20].
- **Volumes:** If a container inside a pod dies the data that belongs to it would normally be lost. However, volumes make sure that the containers can restart in the same state that it died in by keeping track of the data belonging to that container and its contents. A volume is basically a directory containing data that the containers use. Volumes have the same lifetime as the pod it belongs to, therefore if the pod is terminated, so is the volume [21].
- **Namespaces:** Namespaces are mainly used when a cluster, a group of nodes and masters, has multiple users working on them across different teams. Namespaces allows for dividing the resources among the users [22].

The above described objects are used for building the higher level architecture of Kubernetes. The architecture is composed of multiple components that in turn contain the objects described above. The Kubernetes architecture can be broken down into the following parts:

- **Nodes:** Nodes are the workers of the architecture. They contain one or more pods and the necessary services needed to run those pods. The services in the nodes are described below [23]:
  - **Container Runtimes:** Inside the pods are the containers, and in order for them to function a container runtime such as Docker is needed [24].
  - **Kubelet:** Kubelet is the "overseer" of the node that makes sure that each container is running and working according to their specifications. If a container is not working as intended it is redeployed to the same node [24].
  - **Kube-proxy:** The kube-proxy is responsible for forwarding requests to the containers based on the IP and port number of the incoming requests [24].
- **Node controller:** The node controller is responsible for controlling the worker nodes by scheduling pods to them, balancing the workload and makes sure that every node is healthy. The node controller contains various components in order to fulfill its role:
  - **etcd:** The etcd stores the desired state configured for the cluster. The other components consistently check the etcd to make sure everything is running in its desired state [25].
  - **API Server:** The Kubernetes API server configures and validates incoming data to fit the target pod or service. It is both the external interface (the user) and internal interface (the components) of Kubernetes. The user can change the desired state inside the etcd through API requests through the API server, and thereby the workloads and the node containers [25].
  - **Controller Manager:** The controller manager is responsible for watching the cluster through the API server and attempt to keep it running in the desired state [25].
  - **Scheduler:** The scheduler is responsible for assigning unscheduled pods to nodes with available resources. In order for the scheduler to assign pods in an effective way it needs to keep track of the available resources of the nodes. It then schedules the pods to the best fitting node, which in turn balances the work load [24].

These objects and components together make up the self-healing and load balancing Kubernetes architecture. Self-healing is possible through the components always trying to run in the desired state. If any container dies the node leaves the desired state, therefore the container that died will be redeployed in order for the cluster to reach the desired state again. Load balancing is an

important aspect to fully utilize all the available resources, which is automatically done by the scheduler.

Furthermore there are a couple of addons available for Kubernetes. Such as DNS servers for the clusters for ease of communication, a web dashboard for graphic visualization of the clusters, container monitoring, and container logging [24].

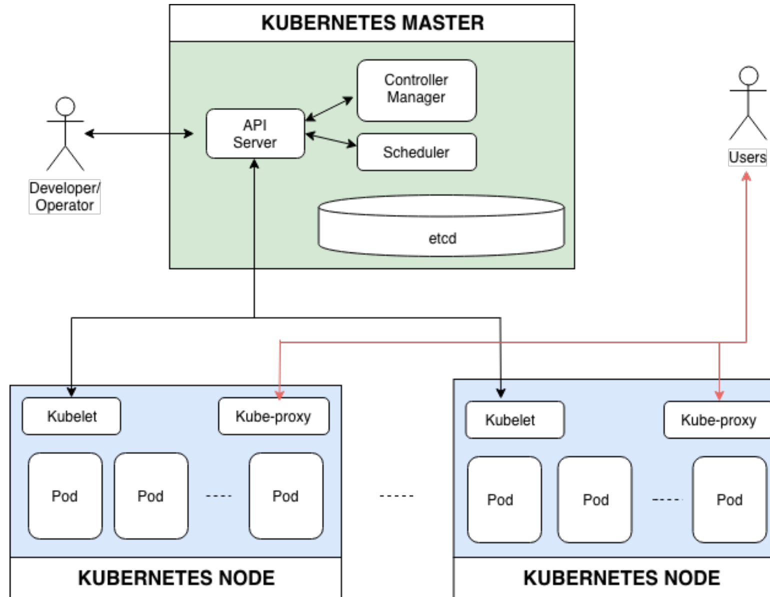

Figure 4: An overview of the Kubernetes architecture.

2.6 Non-functional requirements

When building a product it is important to take into account the non-functional requirements, also known as NFR. Non-functional requirements refer to scalability, maintainability, security, safety, reliability, availability, usability, compatibility among others [26]. Non-functional requirements describe how well a product works, while functional requirements describe what a product does. As an example a functional requirement would state that if the car crashes the airbag will be triggered, while a non-functional requirement states that when the airbag triggers it has to be within 0.1 seconds. It basically describes *how* fast it should be done, not *what* should be done. There are multiple NFRs, with the importance varying depending on the purpose of the product being worked on. Despite the importance of the NFRs they often get neglected and not enough time and effort is dedicated to guarantee that they are fulfilled, which can lead to serious accidents [26]. Therefore it is important to always keep them in mind when designing a product.

It can be hard to design a product to both fulfill user expectations as well as fulfilling NFRs. R. Phalnikar [27] suggests a framework to ease the process of identifying both functional and non-functional requirements. However he also mentions that new problems might arise when building software for cloud environments and that special care is needed when designing such software. Ideally the NFRs should be considered right from the beginning when designing the architecture.

3 Related Work

In the last section the relevant information was described, such as Kubernetes, reference architectures and microservice architectures. This section describes some earlier work done on related areas to the one being explored in this thesis. They provide a starting point for this work. Below are the summaries of such studies that explore microservice practices and reference architectures.

3.1 Microservice bad practices

Taibi and Lenarduzzi [28] explored bad practices and pitfalls that occur when building and migrating to microservice architectures. In their study they performed face-to-face interviews with various people they met during conferences. They limited the interviewees to people that had at least 2 years of experience working with microservices to ensure the validity and value of the answers given. The goal of their interviews was to learn about bad practices when building microservice architectures from the experience of the interviewees. The interviewees were tasked to mention what troubles they had encountered and also to rate the severity of those troubles. Lastly they also mentioned how those troubles could be avoided, or at least be reduced in terms of severity. Taibi and Lenarduzzi then compiled the results of the 72 interviews and managed to narrow it all down to 11 bad practices with their corresponding solutions by grouping together similar answers.

Carrasco et al. [29] felt that the work described above was too focused on the experience of practitioners and could be further improved by having state-of-the-art and practice added. Therefore they conducted literature studies on both academic literature (from Google Scholar) and grey literature (from Google) to further add to the earlier work done by Taibi and Lenarduzzi. Their research resulted in adding 6 more bad practices and how to avoid or solve them when migrating to or building a microservice architecture.

These earlier studies are very useful for this thesis and are important to keep in mind when designing the reference architecture and the guidelines that are proposed. One thing to keep in mind though is that the second study by Carrasco et al. did not only use academic sources. Some of these guidelines are mentioned in Section 7.2.1.

3.2 Security reference architecture

Section 2.1 briefly touched upon possible security issues with cloud services, and section 2.5 mentioned that NFRs (such as security) can be extra challenging to achieve for applications using the cloud. To counter security issues that comes from the use of cloud services many companies have developed *security reference architectures* (SRA) that take into account the possible security threats that their application can face with their corresponding way of tackling those threats (such as authentication to ensure that the users are who they say they are). However, Fernandez and Monge [30] felt that the available SRAs were either too focused on a specific application, or lacking important details. They contributed with a more detailed and general SRA presented with unified model language (UML) diagrams. Their study presented several security threats connected to cloud architectures and how they can be avoided or dealt with.

Security is one of the main concerns of cloud architectures and therefore much care should be given to it. The study made by Fernandez and Monge [30] can be of great value to the reference architecture that will be proposed in this thesis, through having already proposed some solutions to the security threats. The reference architecture in this thesis might take some ideas from the SRA proposed by Fernandez and Monge and combine them with information found from different studies. There are no mentions on how this SRA can be used for microservices, but some of the ideas mentioned might be adaptable.

3.3 Microservice reference architecture

In a study by Yu et al. [31] they explored microservices in the context of enterprise architectures. They explored how enterprise architectures are commonly built and adapted it to fit to a microservice based architecture and then proposed a microservice reference architecture. They highlighted each "building block" of the architecture and provided some solid points on why it was there, such as a repository where the developers could easily document the microservices to

enhance maintainability and reusability. One thing lacking in this reference architecture is some recommendations on technologies that can be used, such as Kubernetes components.

4 Method

The last section described earlier work that are related to the work being done in this thesis, this section describes the methods that will be used to fulfill the goals of the thesis work.

In Section 1, the core thesis goals were mentioned: identify company expectations on microservices, identify the possibility of a microservice reference architecture and identify which Kubernetes components that fit in the reference architectures and meet the company expectations. In order to meet these goals the work was divided into three sections consisting of different methods. The first section used a questionnaire to identify company views and expectations on microservices; the second section conducted a literature study on papers and articles to study the arts and practices of microservice and reference architectures; the third and final section studied the Kubernetes documentation for the available components that were later evaluated based on the needs of the companies. Some of the sections ran in parallel with each other, such as the interview and research phase (while waiting for answers to the interview questions). The sections are described more in detail below.

4.1 Questionnaire

The goal of the questionnaire was to identify the views and expectations that companies have on microservices. The respondents were asked about their general view on microservices; what improvements they hope to achieve from the migration (scalability, maintainability, etc.) where they rated each aspect between 1 and 5; what they think are the biggest disadvantages of the microservice architecture on scale from 1 to 5; if they are thinking about migrating to a microservice architecture, or if they already have migrated. The responses from this questionnaire will be used to identify what Kubernetes components better fit the needs of companies. See Appendix A to view the questionnaire.

This method was further divided into three phases. First a questionnaire was created where the questions had to be carefully chosen and formulated in order to achieve the expected outcomes of the questionnaire. Before the questionnaire was designed the practices of questionnaire design were studied in a book by Iarossi [32] on the subject to avoid common pitfalls.

Finally the questionnaire was sent to the customers that were provided by Enfo, as well as experts on LinkedIn who work as IT directors at different companies. The participants were chosen based on their experience in the areas of system architecture and microservices in order to ensure that the survey results came from reliable and knowledgeable sources. Because the participants were located around the country and had busy schedules it was not viable to interview them face-to-face and therefore it was better to make use of a questionnaire. When the questions were answered the results were be compiled and analyzed.

4.2 Literature study and initial reference architecture design

Studies were made on published articles and papers that describe microservice architectures, Kubernetes and reference architectures. There are also several books on the subject of microservices and architectures, as well as the Kubernetes website that holds the documentation on the platform. This phase, along with the questionnaire results, built the fundamentals of the thesis work by identifying the conclusions of related studies and the company views and expectations. The articles and papers were searched for in the IEEE Xplore, ACM Digital Library and Google Scholar databases.

Analyzing earlier work helped with avoiding repeating work that had already been done, as well as avoiding mistakes and pitfalls.

The results of this phase were the basics of the reference architecture, without the Kubernetes components added to it. The reference architecture was built through analyzing how earlier reference architecture had been designed and by trying to adapt them to fit the microservices.

4.3 Component evaluation

Studies were made on the available Kubernetes components presented in the Kubernetes documentation. The relevant components that were studied handle *container runtimes*, *monitoring*,

logging, *deployment statistics* and *micro-gateways*. The features of the components were gathered and described briefly.

The components were tested and evaluated to see which set of components fitted best with the needs identified by the questionnaire, in terms of complexity of usage and setup, etc. This phase explored whether some components have compatibility issues with each other, if there are some significant limitations to some components or if some components can be used to provide multiple features, for example logging and monitoring simultaneously. The evaluation was done by identifying the component features described in their documentation and, if possible, performance tests. The components were evaluated by how well their features fit the needs and expectations of the companies found in the questionnaire results.

When the components had been evaluated and chosen they were added to the reference architecture. This phase resulted in a reusable reference architecture for microservices.

5 Ethical and Societal Considerations

In the last section the methods that will be used to answer the research questions were described. This section identifies the ethical and societal aspects of this thesis work.

This thesis work includes interviews with companies. Before they participated they were informed that their participation was anonymous, which was guaranteed through the use of Google Forms where no email addresses were collected, therefore there was no way to know who answered what. Who answered what was irrelevant while analyzing the results because the results were just there to give an overview of the general view of microservice architectures. Since this thesis work also includes literature studies it is very important to correctly and fairly cite the authors of the studies used, in order to avoid plagiarizing.

In terms of societal aspects a reusable microservice reference architecture could make microservices more available for companies to adopt. This would result in the designing, building and maintenance process being faster and therefore cheaper. Nowadays cloud data centers use a lot of energy to execute their functionality where 30-45% of that energy consumption is assessed to be wasted on doing nothing because applications scale inefficiently [33, p. 4]. If microservices were more widely used the energy consumption could be lowered because it allows for deactivating unused parts of applications and more efficient scaling, which in turn would lead to a more eco-friendly approach.

6 Company expectations and views on microservices

In Section 4.1 the method of using a questionnaire was described, this section describes how that was done and the results are presented in Section 9.1.

The questionnaire was sent out to experts provided by Enfo and to experts found on LinkedIn that work as IT directors at different companies. However, it proved to be difficult to find people that had knowledge about microservices and wanted to take spend time to answer the questionnaire. The questionnaire ended up getting 6 responses, which did not allow a complete exploration of the expectations, but still provided a hint of the company views and expectations. The questionnaire was made in Google Forms that presents the results in tables and graphs making it easy to get an overview of the results.

7 Microservice and reference architecture pattern studies

In Section 6 the process of gathering information from experts through a questionnaire was described. This section goes through the process of gathering information regarding microservice architectures and reference architectures through literature studies.

Section 4.2 mentioned literature studies on the subject of microservice architectures and reference architectures. This section describes how the literature studies were conducted and briefly describe the findings.

The literature studies were made on papers and articles found on the IEEE Xplore, ACM Digital Library and Google Scholar. The databases were searched using the following keywords: (*reference architecture** OR *RA*), *microservice** and *design pattern**. The search results were filtered by identifying their relevance by their headlines and reading their abstracts, if they were identified as relevant the complete documents were analyzed. The purpose of this section is to explore the state of the art and practice of reference architectures and how they can be applied to microservices to create a generic reference architecture. It also gives an overview on different microservice architecture patterns that are commonly used. The findings of this section, together with the questionnaire results, were used to design the microservice reference architecture.

7.1 Reference architecture design practices

Reference architectures can be challenging to develop, and are usually developed through evolution over a long time and often in cooperation of *domain experts*, *application developers*, *application users* and *architecture developers* [34]. In order to ensure that the architectures are reusable they consist of *common points* and *variable points*. The common points of these architectures are the components and technologies that are general enough to be able to be reused over multiple systems, and the variable points are the components that can be adapted to the system that is developed where multiple technologies can be proposed. The domain expert, together with a software architecture expert, is responsible for identifying the variable points of the architecture and providing documentation and tools for the application developer.

The literature studies showed that just like reference architectures provide a template for system architectures, there also exists references on how to build a reference architectures, such as RAModel [35], which provides documentation templates and guidelines on reference architecture creation, and ProSA-RA [36] that is inspired by RAModel. RAModel and ProSA-RA can both be used to create new reference architectures, as well as to analyze existing ones to identify if they lack some information. The ProSA-RA model divides the process of creating reference architectures into four steps and uses the RAModel to validate that all necessary information is captured in each step:

- **Step 1: Information Sources Investigation:** This step involves gathering the relevant information on the system and its target domain, an example of a domain is web development or enterprise. This phase is to get an overview of the available system technologies that can or will be used, the standard architecture patterns and the terms and concepts of the domain. The information can be gathered from people involved in the domain; publications and technical reports on the domain and its technologies; and already existing reference architectures.

- **Step 2: Architectural Analysis:** The second step is where the information gathered in the first step is translated into functional and non-functional requirements that the system should address. These functional and non-functional requirements are then further translated into reference architecture requirements that are more comprehensive and in-depth than they are for the system. Finally, the language is translated to use the concepts and terms that are used in the target system domain. Domain experts and system analysts should be involved in this step to validate the requirements.
- **Step 3: Architectural Synthesis:** In this step the description of the reference architecture is built by following the recommendations of the RAModel. It describes the architectural patterns used, as well as functionality, the terms used, possible variation points etc. UML (Unified Modelling Language) can be used to visualize the description of the architecture.
- **Step 4: Architectural Evaluation:** Finally, the reference architecture needs to be evaluated to ensure that the reference architecture is designed correctly. It is done by letting different stakeholders, such as system architects, developers and domain experts fill in a checklist derived from the RAModel. The purpose of the evaluation is to review if everything in the reference architecture is correctly described or if something can be changed or removed.

The reference architecture of this thesis work is aimed at the enterprise domain. Enterprise domains vary a lot between companies, which means that there are no common patterns and requirements except for the obvious ones (such as security, maintainability, availability). Microservices and Kubernetes already cover most of the generic requirements by nature through its self-healing environment enabling reliability for example. All of the components are also general enough to provide the basic functions of an enterprise domain, such as monitoring and logging. Therefore, the first and second step have to rely fully on the questionnaire results and information gathered from papers and articles describing the views and expectations on microservices.

Step 4 of the ProSA-RA model can not be done on this thesis work, since it will not be complete enough to be evaluated. Instead, it will propose the foundations for future work that can later on be evaluated by the ProSA-RA.

7.2 Microservice architecture design

An enterprise architecture consist of many components that each handles part of the logic needed for the company system. The microservices are only part of the architecture, but are essential for the system to function. Because there are many microservices that handle different tasks of the system they need to be easily discoverable for the other services that need them. Pods uses dynamic IP addresses that changes each deployment and therefore there needs to be some form of automatic service discovery in order to keep track of the IP address of each service [31]. Kubernetes provides two different ways of service discovery [37], either through a DNS server or through environmental variables added by the Kubelet when a new pod is created. The DNS server is the recommended approach. It schedules a pod on the cluster that runs a DNS server which keeps record of the running services by keeping track of the IP address of each pod. The record can then be referred to when routing traffic to specific pods.

7.2.1 Microservice guidelines

To avoid common pitfalls when implementing microservices it can be helpful to follow some guidelines. This section provides an overview of a couple of guidelines that can be useful to keep in mind.

In general there are some guidelines on what to avoid when designing microservices presented by Taibi et al. [28]. Some of the given guidelines for avoiding common issues are:

- Do not make the microservices too complex, let them handle one or just a few functionalities. If a microservice is responsible for too many functionalities it is better to decompose it into multiple services.
- Avoid having many dependencies between microservices. Too many dependencies will make the microservice architecture seem like a monolithic architecture.

- Implement time-out messages for calling microservices. This will avoid unnecessary calls to unresponsive services.
- Do not share data among microservices, they should only have direct access to their own data. If they need to access data belonging to another service it should be done through an API call.
- Use an API gateway for security and traffic control. The API gateway can handle authorization and provide an interface for external consumers.
- Avoid using too many different technologies. One of the advantages of microservices is that it allows for flexibility in the choice of technologies, since each microservice can use its own technology independently of the others. However, without any standards on what technologies that are allowed it can quickly become a mess of different technologies.
- Only keep containers that are tightly coupled inside the same pod. Containers inside the same pod have access to the same file system, which can be problematic, unless one is a sidecar container that handles logging for the main container for example (more on this further down).

7.2.2 Containerized microservice design patterns

There are different approaches available when deploying containerized microservices. These different approaches makes it possible to enhance or simplify the applications by providing extra functionality. These design patterns are too specific to be able to put into a reference architecture, however, they are necessary to mention briefly for the future improvements on this work. Burns and Oppenheimer [2] discuss some of the main design patterns. Three of the common patterns are designed below:

- **Sidecar pattern:** This is the most common single-node pattern where the sidecar container enhances the main container. As an example, the main container is the application running on the pod, while the sidecar container might be responsible for passing on the logs from the main container to an external storage. The benefits of separating the logging functionality from the main container is that the sidecar container can be reused as a log saver for multiple containers; it can be configured to only use computing power when it is available; the two containers can be tested, updated and deployed independently; and if the sidecar container stops functioning properly, the main container can continue running without disruption. However, if the sidecar container is used for multiple main containers it needs to be tested for each of the container versions, which can be time consuming. All of the benefits described here apply to the other single node patterns as well. A visual example of a sidecar pattern for a web server is shown in Figure 5.

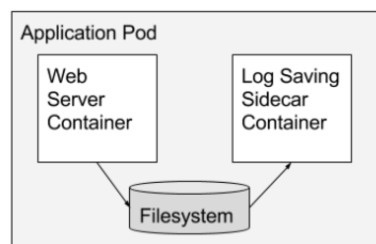

Figure 5: Example of a sidecar pattern for a web server. Source: [2]

- **Ambassador pattern:** The ambassador pattern works in a similar fashion as the sidecar pattern. But instead of having a container that enhances the main container, the ambassador pattern provides an ambassador container that works like a proxy for the main container. This means that the main container can connect to *localhost*, then the ambassador container is configured to discover external services and correctly route the traffic to the receiving external service. This can significantly ease the work of the programmers, since they only

need to think about connecting to a single server. This is possible because containers on the same pod share the IP address. A visual example of an ambassador container is shown in Figure 6.

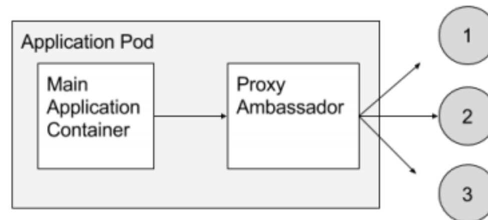

Figure 6: An example of an ambassador container implementation. Source: [2]

- **Adapter pattern:** The adapter container reformats the output from the main container into a standard output format so that the output is uniform across the whole application. This eases the monitoring of the application, by having everything in the same format. See Figure 7 for a visual example of an adapter container.

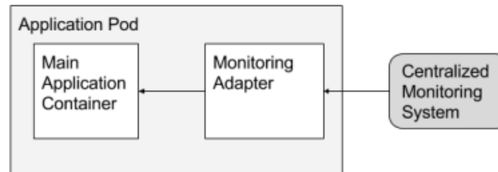

Figure 7: An example of an adapter container implementation. Source: [2]

7.2.3 The initial architecture

Based on the knowledge gathered from the sources a generic Kubernetes-based microservice architecture could be designed, as shown in Figure 8. It has taken into account the guidelines such as adding an API gateway and a DNS service discovery container running on a pod. The red colored components of the architecture are those that are yet to be added to it, which is the next step of this thesis work. The architecture was inspired by a diagram by *The New Stack* [38]. Refer to Figure 4 for a more detailed view of the Kubernetes master and worker nodes.

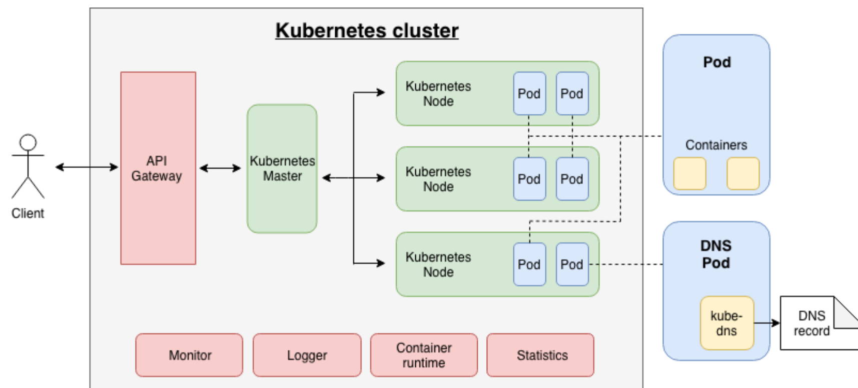

Figure 8: The initial Kubernetes-based microservice architecture. The red colored components are yet to be chosen.

8 Exploration of the Kubernetes components

Section 4.3 mentioned the component exploration and evaluation that was made to find the Kubernetes components that would be used in the reference architecture. This part of the thesis describes the exploration of the available Kubernetes components and why the components are important. It gives an overview of the identified components. The component search was limited to what is listed on the Kubernetes website in order to ensure compatibility between them and the Kubernetes platform. The results and analysis are presented in Section 9.2.

8.1 Container runtimes

Container runtimes are responsible for executing and managing containers. Container runtimes can be divided into higher and lower level runtimes where the lower level is responsible for the

basics of running the container, and the higher level provides extra features over the basics of the lower level. As mentioned in 2.3 Docker is the most commonly used container runtime, but there are more available to choose from. However Kubernetes limits the choices slightly by requiring the container runtimes to support *Container Runtime Interface* (CRI), which acts as an *Application Programming Interface* (API) between the kubelet and the containers inside the nodes [39]. Therefore, the search for container runtimes were limited to the ones that supports CRI. The following three container runtimes are listed in the Kubernetes documentation [2] (the fourth, Frakti, was ignored because of the lack of documentation and therefore did not seem to be widely used):

- **containerd:** Containerd is an open-source project. It is an industry standard container runtime that manages the whole container lifecycle. Cri-containerd is the version that is built to fit the CRI interface in order to work with Kubernetes. Containerd is built on top of the standardized lower level runtime *RunC* where containerd provides extra features over the basic ones provided by RunC [40].
- **Docker:** Docker is the most commonly used container runtime. It is built on top of the containerd runtime as well as the lower level *RunC* runtime. It is a security minded standardized runtime with a big and active community [41]. Docker is the default container runtime used in Kubernetes.
- **rkt:** rkt is a security-minded container runtime developed for being used in cloud-native applications. Rkt is pod-based in the same way as Kubernetes, where each pod contain one or more applications. It allows users to enter parameters, such as the desired isolation level of the pods or applications [42] However, looking at the rkt Git repository [3] it does not seem active with its last commit more than a year ago. Because of the inactive development, rkt will be left out of the candidates.
- **CRI-O:** CRI-O is a standardized container runtime that is optimized for Kubernetes. It is built around satisfying the Kubelet CRI requirements through monitoring and logging, as well as resource isolation. Because CRI-O is following the same deployment cycles as Kubernetes the latest versions will always be compatible with each other, which might not be the case with other runtimes. CRI-O manages the full container lifecycles and supports multiple container formats, including Docker. Similarly to containerd CRIO-O is also built on top of RunC [43].

The three canditates remaining are **Docker**, **containerd** and **CRI-O**. A test was conducted to compare the CPU and memory usage of each runtime. The test was done by setting up a single-node local Kubernetes cluster with Minikube by following a setup guide by bakins on Github [4]. The setup used Prometheus to collect the data and Grafana to visualize it on a customizable web dashboard. The queries used for visualizing the data on Grafana were

sum(rate(container_memory_usage_bytesimage!=""[100m])) by (pod_name)

for memory usage, and

sum(rate(container_cpu_usage_seconds_totalimage!=""[100m])) by (pod_name)

for CPU usage. These queries displayed the memory consumption and CPU usage of each pod running on the cluster. Minikube runs Docker as a runtime by default, in order to change runtime it had to be restarted with the other runtimes by running the commands **minikube start –container-runtime=cri-o** for CRI-O and **minikube start –container-runtime=containerd** for containerd. The setup had to be redone from start for each runtime. Each cluster was left running for 30 minutes to make sure they had the same conditions. After 30 minutes the data for the most consuming pod was collected for each runtime.

8.2 Monitoring

Monitoring is important to get an overview of how the application behaves. Through monitoring you can see the performance of the application and its usage of resources, making it possible to

[2] https://kubernetes.io/docs/setup/cri/
[3] https://github.com/kubernetes-incubator/rktlet
[4] https://github.com/bakins/minikube-prometheus-demo/blob/master/README.md

identify possible bottlenecks. Through monitoring it is possible to find possible malicious software and other security issues as well [44]. Kubernetes provides detailed information regarding the performance of your application on containers, services and pods making it possible to remove bottlenecks and improving the performance of the application. The monitoring can be done on two different pipelines: the resource metrics pipline and the full metrics pipeline. The Kubernetes website [5] was searched to identify the available tools for examining the monitoring tools. The following tools were found:

8.2.1 Resource metrics pipeline

The resource metrics pipline that provides limited information on resource usage in cluster components, such as how much memory or CPU power each pod consumes.

- **Kubelet:** The kubelet exist on every node and acts as a bridge between the master and worker nodes. It collects the individual container resource statistics from the interface of the container runtime. The kubelet fetches its data from the cAdvisor [45].
- **cAdvisor:** cAdvisor is an open-source performance and resource monitor. It discovers all containers automatically and proceeds to extract statistics regarding memory usage, CPU usage, network usage and filesystem. By analyzing the "root" container, the container highest in the hierarchy, it can get a complete view of the statistics of all containers. cAdvisor provides a UI for displaying the statistics [45].

8.2.2 Full metrics pipeline

The full metrics pipeline provide richer metrics about the whole system that the Kubernetes cluster can automatically react to and scale depending on the current state of the cluster.

- **Prometheus:** Prometheus is a widely used open-source systems monitoring toolkit. It collects metrics from the jobs running that are then available for visualization on platforms such as Grafana. It only natively supports the Prometheus metrics, with special configurations being needed to support custom metrics. Prometheus provides a tool called *Prometheus Operator* that simplifies the setup on Kubernetes. Prometheus aims to be reliable and presents data even when parts of the application fails which can help with diagnosing problems. However, the collected data might be inadequate when 100% accurate data is needed [45] [46].
- **Sysdig:** Sysdig is a full stack monitoring platform. It auto-discovers all of the environment and displays information about query response times, Prometheus collected metrics, container data, Kubernetes events and more. With the help of alerts the users are notified of events that require attention. It then displays it all in one pane to give an extensive overview of the environment, it also supports use of Grafana to visualize the metrics. It supports custom metrics, such as JMX and StatsD without any special setup. Sysdig also provides a tool called *Sysdig & Sysdig Inspect* that allows for troubleshooting and analyzing the environment. With Sysdig there is a security container runtime tool called Falco. With Falco it is possible to define the usual and expected behaviour of the containers, when something unusual happens the user is notified of suspicious behaviour [45] [47].
- **Google Cloud Monitoring:** Google Cloud Monitoring or *Stackdriver Monitoring* is a service for monitoring applications hosted on Google Cloud Platform or AWS. It can alert and visualize metrics gathered from Kubernetes on a *Cloud Monitoring Console*. The user can create and customize dashboards that visualize the date to their liking [45].
- **New Relic Kubernetes:** New Relic Kubernetes is a tool that monitors the CPU and memory consumption of the cluster on each level. It shows the desired amount of pods and ensures that they are running and healthy. New Relic also measures the data throughput and error rate in the cluster applications. In order to help satisfy the users of the application it is possible to measure the response times and error rates of the user interfaces. New Relic also provides information about relationships between the objects in the cluster and their metrics; alerts to notify the user about events that needs attention; and pre-built dashboards to visualize the data with custom queries. A new feature called Kubernetes Cluster Explorer was recently released which gives a visual color-coded overview of the cluster

[5] https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

topology, which presents resource consumption of each container, pod and node. The Cluster Explorer simplifies identifying dependencies in the cluster and solving issues that occur. New Relic is limited to being hosted on the AWS, Google, Microsoft and Rackspace clouds, or it can be self-hosted which means that it is hosted and stored locally [48].

8.3 Logging

Logging is important in order to be able to analyze if everything works as expected or to pinpoint where and when something went wrong. It is also important for security reasons where logs can display possible malware activity or other security breaches. In Kubernetes logging can be done on multiple levels: basic logging at container level, logging at node level and logging at cluster level [49].

8.3.1 Basic logging

The basic logging on the container level is simple done by outputting data to the output stream. The logs are displayed by running the *kubectl logs* command. These logs are temporary and will be deleted when the pod containing the container is deleted. In order to avoid the deletion of logs they need to be gathered by a logging agent that passes them on to an external logging solution. Two of these logging solutions are described in the *Cluster level logging* section below.

8.3.2 Node level logging

Everything that is written to the output stream inside a container is redirected somewhere by a container engine. The Docker container engine redirects the output to a *logging driver* that format the output into a JSON file by default. It is important to remember rotating the logs in order to avoid them taking up all the node memory by replacing the old log with a new.

8.3.3 Cluster level logging

On the Kubernetes website they suggest three different approaches for cluster level logging:

- **Using a node level agent:** This is the most common approach for cluster level logging and is recommended in the Kubernetes documentation. This is basically the same approach as described in *Node level logging*, except that it is on a higher level with a node agent running on each node. The recommended way of doing this is by running a *DaemonSet* that implements a node agent. A *DaemonSet* automatically starts a pod, running the specified program in a container, each time a node is created. This ensures that each node in the cluster runs the desired program, in this case a logging agent. There are two optional logging tools suggested on the website, that both use a customized *FluentD*[6], an open source data collector, as the node agent. The node agent has access to a directory containing all of the logs from the containers on that node. FluentD is a lightweight agent that provides many plugins allowing the user to customize it.
  - **Stackdriver Logging:** Stackdriver Logging is a part of the Stackdriver toolset. It allows for logging applications that run on Google Cloud Platform (GCP) or Amazon Web Services (AWS). Stackdriver Logging and Stackdriver Monitoring are both part of the same toolset, leading to them both working seamlessly together. Stackdriver Logging is capable of high scaling, as well as providing real time data analysis of either GCP or AWS, or a combination of both [50].
  - **Elasticsearch:** Elasticsearch is an open-source search engine with good scaling possibilities. In order to use Elasticsearch as a logging tool it needs to be combined with *FluentD* that collects and parses the logs. Elasticsearch can then be used to search and mine the data collected by FluentD. *Kibana* is a tool that is used to visualize the Elasticsearch data. The combination of these three tools is known as the *ELK stack* [51].

[6]https://www.fluentd.org/

- **Using a sidecar container with the logging agent:** A sidecar container is a container running on the pod that cooperates with the other containers running on the pod. There are two suggested approaches when using a sidecar container for logging. The two approaches are the following:
  - The sidecar container outputs logs to its own output stream. This approach takes advantage of the logging agent and Kubelet that already run on the node, letting the sidecar container read the logs and sending them to its output stream. By doing this the output stream from different parts of the application can be separated. This is useful if different logs are in different formats, so you can have one sidecar container for each of those formats to keep those streams separated.
  - Run a customized logging agent inside a sidecar container. This approach is intended to be used when when the node level logging agent does not fulfill your needs, therefore this side car logging agent should be customized to do what the other logging agent can not.
- **Passing logs directly from the application:** The last suggested approach is to directly push the logs from the application to where you want to store them. However, the Kubernetes documentation does not go into detail about this.

Out of the three above suggested approaches the Kubernetes documentation recommends using a node level agent. This approach does not need as much resources as using a sidecar container, since it only has one agent running on each node.

8.4 Deployment statistics

Deployment statistics is important to see get an overview of what is successfully running on the cluster and what is not. It can also important to see what jobs have successfully been completed and what has failed to complete. All of this is covered automatically by most of the monitoring tools available. Sysdig, which was chosen to be used as a monitoring tool give statistics on this, therefore it is not needed to get an additional component to handle this task.

8.5 Micro-gateways

Micro-gateways are API gateways that follow the principles of microservices to be small. API gateways are in turn gateways that controls the incoming and outgoing traffic to and from microservices. Without an API gateway the microservices can directly communicate with each other which in turn can lead to security threats and complicated maintenance. With an API gateway microservices are only handed the data that they need [28]. API gateways present an interface between API clients and the computational parts of the applications. API gateways eases the communication between different clients, for example mobile phones and computers, and the services. The gateway transforms the incoming traffic to a standard format that every service understands and thereby removing the need for services to be configured to understand many different queries and reducing the code complexity of the services. API gateways also provide load balancing by sending the traffic to the service best suited to perform the task, as well as security in the form of filtering traffic and providing authentication [13]. These gateways can be beneficial when migrating a monolithic architecture to a microservice architecture by providing mapping between the monolithic application and the partly built microservice architecture. The Kubernetes documentation does not recommend any API gateways, instead they mention some ingress controllers that are responsible for routing the incoming traffic to the correct end point [52]. Therefore, the API gateways presented here were chosen from *learnk8s* [53], a site dedicated to teaching Kubernetes. The three API gateways suggested by learnk8s are suggested as ingress controllers on the Kubernetes website because they have ingress controllers built into them. The API gateways suggested are the following:

- **Kong:** Kong is an open source API gateway that fully supports RESTful API, which is the most commonly used API standard for web and cloud software. Some of the popular features offered by Kong plugins are traffic control, authentication, traffic logging, traffic monitoring and traffic transformation. It can handle any number of microservices, making it viable for

both big and small applications. The Kong configurations are stored in either a PostgreSQL database or Apache Cassandra datastore [54].

- **Ambassador:** Ambassador is an open source Kubernetes-native API gateway, meaning it was built specifically for Kubernetes and therefore integrates well with it, which is the biggest difference between Kong and Ambassador. Scalability, reliability and availability is not achieved by Ambassador itself, it instead on Kubernetes to handle it. It also stores the configurations on the Kubernetes database, meaning no extra database is needed when using Ambassador. It offers features such as traffic rate limiting, authentication and a diagnostics tool used for debugging [55].
- **Gloo:** Gloo is another Kubernetes-native API gateway. Gloo has similar features as the two gateways mentioned above, including rate limiting, authentication, transformation and security. However, it also has some features that make it different from the others; it automatically discovers and keeps track of each end point of the environment; it integrates with many of the top open source projects; it can route requests directly to functions running in the back end, such as a function in a microservice. This last feature makes hybrid apps in Gloo possible and is the feature that makes it unique from other API gateways. Hybrid apps are applications that use different patterns and architectures, for example by combining both containerized microservices and serverless in the same application [56].

9 Results

Sections 6, 7 and 8 describes the work that was done to answer the research questions, this section presents and analyses the results from those sections, including the chosen Kubernetes components to be used in the reference architecture.

9.1 Customer views and expectations on microservices

The questionnaire was answered by 6 respondents. Of all the respondents 50% of them already have, or is in the process of migrating to a microservice architecture. The remaining respondents seem reluctant on migrating towards a microservice architecture because of the reasons that microservices are complex, new and not yet standardized enough and that the migration is too costly and time consuming. However, all respondents who have not adopted microservices report that they are having issues with big and complex codebases and maintenance and believe that microservices can solve those issues and in turn improve the quality of their systems. The questionnaire participants were asked to rate the importance of some aspects they wanted to achieve from adopting microservices. The rating was done on a scale from 1-5 where 1 was not important and 5 was important. As shown in Table 1, maintainability was the most important aspect to improve, closely followed by scalability and function reusability. Least important of the aspects were achieving smaller codebases on the applications and the ability to use different technologies, however, they were still rated as above moderately important. In similar studies conducted by Knoche and Hasselbring [57] and Taibi et al. [58] it shows that the most important aspects of microservices are maintainability and scalability.

Kubernetes and microservices solves these problems by their design, by providing the ability to scale, maintain and update parts of the application independently and split it up into smaller parts to reuse. However, the maintainability and reusability also become more complex, requiring good documentation and monitoring in order to utilize these functionalities. The respondents who already had adopted microservices reportedly achieved high increase in both maintainability and scalability.

Aspect	Importance mean value (1-5)
Maintainability	5
Scalability	4.5
Function reusability	4
Technology flexibility	3.5
Smaller codebases	3.5

Table 1: The importance rating mean value given by the questionnaire respondents on different aspects they expect from microservices. The aspects were ranked between 1 and 5 where 1 is not important and 5 is important.

The respondents were also tasked to rate the severity of the disadvantages that come with microservices. They rated each aspect at a scale from 1 to 5 where 1 was not problematic, and 5 was problematic. The results were split into two data sets, one set for all of the participants combined (Table 2) and the other for the participants who had adopted microservices who might have experienced the disadvantages themselves (Table 3). By comparing the two severity rating of the two data sets it is shown that the respondents who uses microservices see the disadvantages as more problematic than those who do not use microservices. The biggest difference is that the data in Table 3 has rated complex maintenance as the least severe, while Table 2 has it rated as top two. These disadvantages can not be fully avoided, but they can be mitigated through the use of various tools and components, which is what the selected Kubernetes components are used for.

The complex design and planning phase was rated as problematic, and was one of the reasons mentioned for not wanting to migrate to a microservice-based architecture. Therefore this reference architecture is proposed to simplify the migration and creation of a microservice architecture. With a blueprint to help speed up and lower the cost of the architecture design it would enable more companies to adopt microservices.

Disadvantage	Severity mean value(1-5)
Complex maintenance	3.33
Complex design and planning	3.33
Complex testing and deployment	3.17
High memory consumption	3

Table 2: The mean value of the severity rating of the disadvantages given by all respondents on a scale from 1 to 5, where 1 is not problematic and 5 is problematic.

Disadvantage	Severity mean value(1-5)
Complex design and planning	4
Complex testing and deployment	4
High memory consumption	4
Complex maintenance	3.5

Table 3: The mean value of the severity rating of the disadvantages given by the respondents who have adopted microservices on a scale from 1 to 5, where 1 is not problematic and 5 is problematic.

9.2 Component analysis

This section analyzes the components picked in section 8 to see which components would fit best in the reference architecture. The analysis takes into account how the features of the components fit with the questionnaire respondents' views and expectations. A requirement of the components is that they should not limit the reusability of the reference architecture by only allowing use of specific technologies.

9.2.1 Container runtimes

As shown in Figure 9 the memory consumption of CRI-O was much lower than the memory consumption of Docker and containerd. Containerd proved to consume a lot more memory than both CRI-O and Docker. However, Docker used a lot more of the CPU power than CRI-O and containerd, as shown in Figure 10. Containerd and CRI-O were a lot more even on the CPU usage.

CRI-O seems to be the best fit as a runtime, not only because of the low performance costs, but also because it is especially tailored for Kubernetes. Seeing as the majority of the questionnaire respondents find the microservice memory consumption as problematic it would be wise to choose a runtime that is less memory consuming. For these reasons CRI-O will be used as a container runtime in the reference architecture.

Chosen container runtime: CRI-O

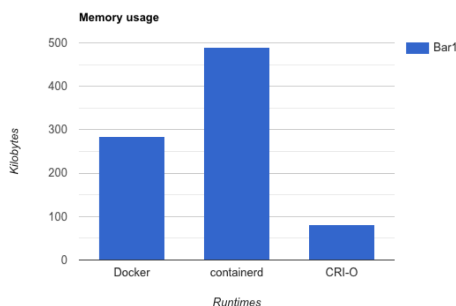

Figure 9: Memory usage for each runtime.

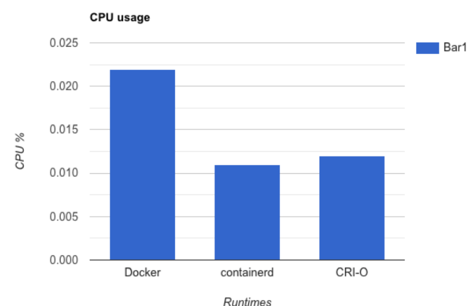

Figure 10: CPU usage for each runtime.

9.2.2 Monitoring

The resource metrics pipeline does not provide enough data to be useful for enterprise monitoring, but is more fitting for smaller projects. Google Cloud Monitoring likewise does not fit in the reusable reference architecture because it is limited to using GCP or AWS, which would severely limit the cloud services available for the reference architecture. New Relic Kubernetes is also limited to a few cloud services or to being self-hosted, which could remove the reusability of the reference architecture.

The two monitoring tools left standing are therefore **Sysdig** and **Prometheus**. Sysdig is a paid service that is easy to set up with Kubernetes. With the enterprise subscription comes around the clock commercial support. Prometheus is open source and free to use, however, the setup can be complex and time consuming and support is given by the community with a couple of third party commercial support options available.

The questionnaire respondents reported that they find complex maintenance, testing and deployment problematic. With a good monitoring solution this would be simplified. Setting up Prometheus for a big enterprise could be time consuming and complex, wasting precious time. In order to troubleshoot with Prometheus it has to be set up as well. Sysdig provides a plug-and-play monitoring service with its' own troubleshooting toolkit working out of the box for a relatively low cost.

Because of the ease of set up, testing and maintenance **Sysdig** is chosen for the reference architecture as it fits the needs of the questionnaire respondents better than Prometheus does. The respondents who had not yet migrated to a microservice architecture rated the complex design and planning phase as problematic, with Prometheus it would take longer and be more problematic than by using Sysdig. With a dedicated support team it is easier to get help quickly when it is needed, which can prove to be critical if any problem occur.

Chosen monitoring toolkit: Sysdig

9.2.3 Logging

Since Stackdriver has the same problem as Google Cloud Monitoring with being limited to AWS and GCP as the available cloud services it does not fit the goal of being reusable. This leaves the EFK stack. However, with an already available monitoring tool with its own way of visualizing data there is no need for Kibana. All that is needed is simply FluentD to collect the logs and Elasticsearch to organize the logs, which can then be visualized in the monitoring tools' dashboard. Sysdig was the toolkit chosen to be used for monitoring. Sysdig can be used together with Falco, that was mentioned earlier, FluentD and Elasticsearch to provide powerful logging and security [59]. By reusing components to perform a wider set of functionality setting up the system is easier and quicker. It will also be easier to maintain one, or a few, technologies than multiple and licensing costs will be lower as well. Therefore, Sysdig, together with FluentD and Elasticsearch, fits the needs of the questionnaire respondents best to provide logging.

Chosen logging toolkit: FluentD and Elasticsearch together with Sysdig

9.2.4 Deployment statistics

As mentioned in section 8.4 the deployment statistics can be monitored with Sysdig, eliminating the need for use of more technologies.

Chosen tool for deployment statistics: Sysdig

9.2.5 Micro-gateways

Choosing an API gateway is hard without knowing the specific gateway requirements of a company, which can vary a lot. Gloo, however, is a new gateway that fills a niche functionality that is mostly useful for serverless environments or for migrating monolithic architectures. It also does not provide

as much features as Kong and Ambassador or as good documentation. Therefore, Gloo does not fit well in this reference architecture.

Ambassador provides both a free version, as well as a more feature-rich enterprise version. It is easy and quick to set up with the Kubernetes environment, which reduces the complexity of the microservice planning and design phase that was a concern for the questionnaire respondents. The enterprise version also provides realistic testing environments for the services, which eases the service testing, which was also a concern for the questionnaire respondents. It also provides Prometheus and StatsD metrics, which makes it fit well with the Sysdig monitoring toolkit.

Kong also provides both a free and an enterprise solution, however, the free version has a lot less features than the Ambassador free version. Kong is not as easy to set up with the Kubernetes environment as Ambassador is, since it is not Kubernetes-native. It provides a lot of plug-ins that allows the users to tailor the gateway to their needs, but they can be complex to set up as well. It does not provide a realistic testing environment like Ambassador does. Both Kong and Ambassador are widely used, meaning there is a lot of both commercial and community support available. Ambassador fits the needs of the questionnaire respondents better than Kong because of the ease of setup, as well as the possibility of realistic testing environments.

Chosen gateway: Ambassador/Ambassador Pro

9.3 The proposed microservice reference architecture

In Section 7.2.3 the initial Kubernetes microservice reference architecture was presented without the fitting Kubernetes components. This section presents the proposed reference architecture with the chosen Kubernetes components, along with a brief description of how it works.

The proposed microservice reference architecture is shown in Figure 11. As shown in Section 9.1 the most important aspects of the microservices reported by the questionnaire respondents and similar studies ([57], [58]) were maintainability and scalability, tightly followed by function reusability. The improved maintainability and scalability are fulfilled by Kubernetes and microservices by nature, while function reusability needs additional in-depth documentation on each microservice to enable the developers to reuse them. The most problematic disadvantages rated by the questionnaire respondents were complex maintenance and complex setup of the microservice architecture, tightly followed by high memory consumption. Some non-functional requirements such as security and reliability were taken for granted as important for each system. To mitigate these disadvantages the following Kubernetes components were added to the reference architecture:

- **CRI-O container runtime:** The CRI-O container runtime is Kubernetes-native and strives to always work seamlessly with Kubernetes, which will decrease maintenance with possible future compatibility issues. It is also the least CPU and memory consuming container runtime which was rated as a problematic disadvantage by the questionnaire respondents.
- **Ambassador API gateway:** This gateway is Kubernetes-native and is therefore easy to set up with the Kubernetes environment, therefore speeding up the initial setup of the environment. It provides security in the form of authentication and authorization as well as a testing environment for realistic testing of the services.
- **Sysdig:** Sysdig was chosen for monitoring because of the easier setup, the compatibility of different metrics formats and a dedicated troubleshooting tool.
- **FluentD + Elasticsearch:** These tools were chosen to handle logging because they work together with Sysdig to display the logs, meaning there are fewer technologies to maintain.

These components together best reflects the needs and expectations of the companies found by the questionnaire, while also keeping reusability in mind. This architecture can be used as a reference architecture when creating future microservice architectures. By using this proposed reference architecture when creating an architecture, the needs and expectations of the companies will be met.

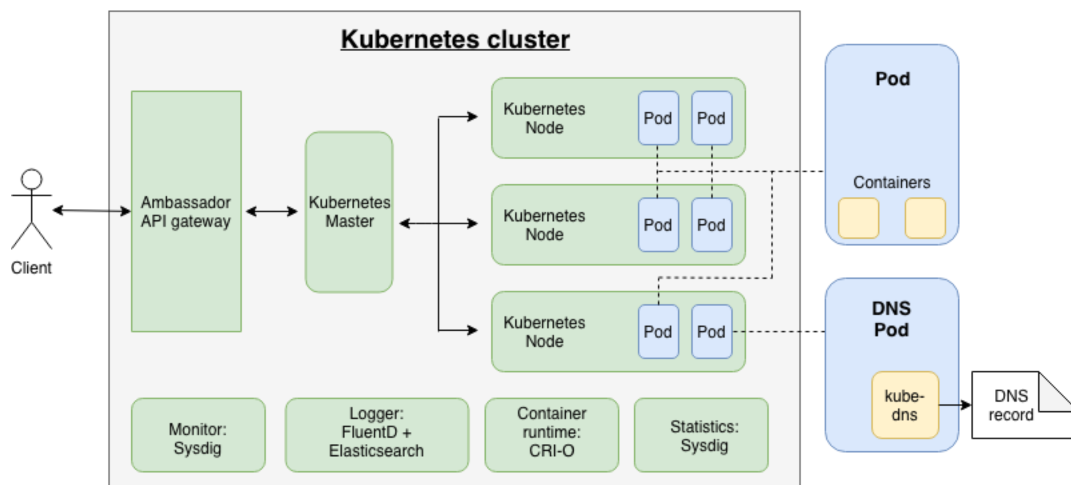

Figure 11: The Kubernetes cluster architecture with the components added.

9.4 Description of operation

The Ambassador gateway collects information regarding the cluster layout from the DNS record collected by the kube-dns container. This information is used for load balancing and routing traffic

to the correct end points. The Ambassador gateway also performs authentication and authorization on incoming traffic.

The monitoring is performed by Sysdig that plants an agent container on each node to collect the data from the pods running on those nodes. This way Sysdig can present data and statistics specific to the cluster or down to container level, providing an overview of the whole system. The collected data can then be displayed on the customizable Sysdig dashboard. FluentD has agents running on each node that collects the logs given by the pods running on the hosts, which together with Elasticseach and Sysdig then can be visualized.

The CRI-O container runtime sits on each node between the Kubelet and the containers. It is a lightweight runtime that performs the commands given by the Kubelet on the containers, such as starting and stopping them.

10 Discussion

The thesis has conducted a study on the customer views and expectations on microservice architectures in order to shape a reference architecture according to their needs. The results from the questionnaire, together with literature studies, helped form the foundations of a microservice reference architecture to be used together with Kubernetes. However, the limited time frame and the niche area of microservices proved it hard to find experts in the area that could answer the questionnaire, leading to a low participation. Despite the low participation, the answers given, along with similar expectations identified from the literature studies were enough to give a hint on what components that should be used to fit the respondents. Kubernetes as a platform and microservices handles most of the expectations that the respondents have, such as scalability, increased maintainability and function reusability. However, there are disadvantages that come with microservices that can not be evaded, but they can be mitigated by making use of different technologies added to the Kubernetes environment. The respondents ranked the disadvantages according to the severity of them to identify which ones should be prioritized to be mitigated by the use of components. The components were also chosen to not be limiting the rest of the technology choices, such as cloud service providers.

The result is the foundations of a reusable microservice reference architecture with the purpose of meeting company needs and mitigating the most problematic disadvantages through the use of Kubernetes and carefully chosen components to go with it. The components used for the architecture have been chosen with easy setup, light weight and maintainability in mind. All of these components have the possibility to be swapped for other components if needed for future use, making it flexible. The reference architecture took into account some of the microservice "bad smells" and practices mentioned by Taibi et al. [28] and Carrasco et al. [29] mentioned in Section 3, that can be used for future implementation. When implementing it is also recommended to use Swagger [7] or some similar tool for documenting the APIs. That section also mentioned a security reference architecture presented by Fernandez and Monge [30] that described how to implement security into the architecture. They mentioned solutions such as authentication and authorization, together with logging to increase the security. The reference architecture in this thesis work presents multiple layers of security for the system, in the forms of authentication and authorization that can be set up in the API gateway, CRI-O as a secure runtime and FluentD to work together with Sysdig Falco to spot out of the ordinary behaviors on the system. Section 3 also mentioned a microservice reference architecture for enterprise architectures presented by Yu et al. [31], which was on a higher level than the one presented in this work. The reference architecture proposed in this thesis is more focused on how the microservices are implemented, meaning this reference architecture can be used together with the one presented by Yu et al. to cover the whole architecture on multiple levels.

The Kubernetes components were chosen from the ones presented in the Kubernetes documentation, however there are many more components available, but they are out of the time frame of the thesis. Therefore, there is a possibility other components exists that better fit the questionnaire results, but they are outside the scope of this thesis.

For future improvements on the reference architecture,it is recommended to follow the guidelines provided by the ProSA-RA [36] and RAModel [35] to achieve a good reference architecture that can later be evaluated by the same tools. By following this path a finished microservice reference architecture will be achieved, which in turn will speed up the creation of microservice architectures, lowering the cost and making microservices available to a broader audience. Microservices, by nature, enables faster updates more frequently, which in turn allows technology to evolve faster. If microservices was available to everyone, then perhaps the technology could evolve faster than it is today.

[7] https://swagger.io/

11 Conclusions

Today companies need to update their systems more frequently, sometimes multiple times a day. However, the traditional monolithic way of building software often prevents updating big software because it is too time consuming and therefore updates only happen when time allows. The microservice approach is a new way of decomposing the monolithic application into smaller parts that can be updated and scaled independent of each other, resulting in more efficient maintenance and more scalable applications. However, microservices are new and lack standards and guidelines on how to create applications using this approach, which in turn leads to long and costly application developments or migrations from monolithic applications. Furthermore, microservice architectures are also more complex than monolithic architectures because the applications are distributed into multiple smaller containerized services. In order to properly maintain and manage all the containerized services a container orchestration platforms is needed. The most popular container orchestration platform is Kubernetes and it is the one in focus in this thesis work. Kubernetes provides a self-healing and auto-scaling environment for the microservice application, with many different components to add onto the basic platform.

An approach to speeding up the creation of system architectures is the use of a reference architecture. A reference architecture acts as a blueprint for the architecture, providing the diagrams, documentation and technology suggestions for the architecture. Following the guidelines of a reference architecture leads to quicker, cheaper and more stanardized architectures, which is turn allows for cheaper and more time efficient maintenance of them.

This thesis proposes a microservice reference architecture for Kubernetes-based applications. The reference architecture proposes a set of Kubernetes components based on the views and expectations that companies have on microservices. The components were picked to handle *logging*, *monitoring*, *deployment statistics*, *micro-gateway* and *container runtimes* according to the needs of the companies. In order to propose a microservice reference architecture that reflects the company views and expectations the following research questions had to be answered:

RQ1. What are the key expectations on non-functional requirements of companies on a microservice based architecture?

This research question was answered by letting experts fill in a questionnaire about microservices. The results from earlier similar studies were added to complement the questionnaire results. The biggest advantage that the respondents expected from microservices was increased scalability and maintainability, while the biggest disadvantages were complex maintenance, as well as complex design and planning of microservice architectures. Refer to Section 9.1 and Tables 1, 2 and 3 for a more in-depth description of the results.

RQ2. What are usually the core features implemented into reference architectures, and how can they fit in the proposed microservice reference architecture?

Reference architectures evolve over time with many individuals involved, such as *domain experts*, *application developers*, *application users* and *architecture developers*. Each individual contributes to the reference architecture with their area of expertise. A reference architecture consists of common and variable points. The common points are the parts of the architecture that can not be changed, while the variable points are parts that have multiple choices of technologies or implementations. It is the job of the domain experts and architecture developers to identify these points and then provide the application developers with sufficient documentation of the architecture. The documentation describes the architecture and provides guidelines regarding what technologies can or should be used. UML diagrams are often used in combination with the documentation. A microservice reference architecture can be created in the same way, but requires more careful documentation in order to reduce the complexity of the architecture. More information regarding reference architectures can be found in Section 7.

RQ3. Which Kubernetes components should be used in the proposed reference architecture, based on the company expectations?

The Kubernetes components were identified in the Kubernetes documentation, and studied further in their own documentation. The components were evaluated based on how well their features reflected the views and expectations of the companies found through the questionnaire. The Kubernetes components chosen to be in the reference architecture are:

- **Container runtime: CRI-O**
- **Gateway: Ambassador/Ambassador Pro**
- **Monitoring: Sysdig**
- **Logging: FluentD + Elasticsearch in combination with Sysdig**
- **Deployment statistics: Sysdig**

For a more in-depth explanation and analysis of the components and why they were chosen, refer to Section 9.2. Section 8 describes the component exploration process.

The results of this thesis work is a microservice reference architecture designed with the companies' need and expectations in mind. This reference architecture can be utilized in order to speed up creation and maintenance of future microservice architectures. For an explanation of the resulting architecture and how the Kubernetes work together, refer to Section 10.

11.1 Future work

The microservice reference architecture proposed in this thesis need some future work in order to complete it. It is suggested to continue using the ProSA-RA [36] reference when creating the documentation and diagrams. It provides guidelines and references for the creation of reference architectures, ensuring that no crucial parts of the architecture is forgotten or ignored. ProSA-RA also provides ways of evaluating the architecture.

References

[1] Docker, "What is a Container? — Docker," 2019. [Online]. Available: https://www.docker.com/resources/what-container

[2] B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[3] S. Martínez-Fernández, C. P. Ayala, X. Franch, and H. M. Marques, "Benefits and drawbacks of software reference architectures: A case study," *Information and Software Technology*, 2017.

[4] A. Islam, M. Irfan, K. Mohiuddin, and H. Al-Kabashi, "Cloud: The Global Transformation," in *2013 International Conference on Cloud & Ubiquitous Computing & Emerging Technologies*, 2013.

[5] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

[6] A. Ziani and A. Medouri, "Risks and Security Requirements for Cloud Environments," in *Proceedings of the 3rd International Conference on Smart City Applications - SCA '18*, 2018.

[7] James Lewis, "Micro Services: Java, the Unix Way," 2012. [Online]. Available: https://www.infoq.com/presentations/Micro-Services

[8] Rick Osowski, "Introduction to microservices IBM Developer," 2017. [Online]. Available: https://developer.ibm.com/tutorials/cl-ibm-cloud-microservices-in-action-part-1-trs/#netflix-streaming-the-birth-of-popularized-microservices

[9] S. Li, "Understanding Quality Attributes in Microservice Architecture," in *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, 2017.

[10] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," *Present and Ulterior Software Engineering*, 2017.

[11] P. Wegner, "Concepts and paradigms of object-oriented programming," *ACM SIGPLAN OOPS Messenger*, 1990.

[12] Z. Xiao, I. Wijegunaratne, and X. Qiang, "Reflections on SOA and Microservices," in *2016 4th International Conference on Enterprise Systems (ES)*, 2016.

[13] C. Richardson and F. Smith, *Microservices - From Design to Deployment*. NGINX, 2016.

[14] Kubernetes, "What is Kubernetes? - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[15] Docker, "Enterprise Application Container Platform — Docker," 2019. [Online]. Available: https://www.docker.com/

[16] E. N. Preeth, F. J. P. Mulerickal, B. Paul, and Y. Sastri, "Evaluation of docker containers based on hardware utilization," *2015 International Conference on Control Communication & Computing India (ICCC)*, 2015.

[17] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015.

[18] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture," *ACM SIGAPP Applied Computing Review*, 2018.

[19] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15, 2015.

[20] Kubernetes, "Pods - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod/

[21] Kubernetes, "Volumes - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/

[22] Kubernetes, "Namespaces - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[23] Kubernetes, "Nodes - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/

[24] Kubernetes, "Kubernetes Components - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/#node-components

[25] OpenShift, "Kubernetes Infrastructure - Infrastructure Components — Architecture — OKD Latest," 2019. [Online]. Available: https://docs.okd.io/latest/architecture/infrastructure_components/kubernetes_infrastructure.html

[26] V. Bajpai and R. P. Gorthi, "On non-functional requirements: A survey," in *2012 IEEE Students' Conference on Electrical, Electronics and Computer Science*, 2012.

[27] R. Phalnikar, "Validation of Non-functional Requirements in Cloud Based Systems (Short Paper)," in *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, 2016.

[28] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, 2018.

[29] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: migration and architecture smells," in *Proceedings of the 2nd International Workshop on Refactoring - IWoR 2018*, 2018.

[30] E. B. Fernandez and R. Monge, "A security reference architecture for cloud systems," in *Proceedings of the First International Conference on Dependable and Secure Cloud Computing Architecture - DASCCA '14*, 2014.

[31] Yale Yu, H. Silveira, and M. Sundaram, "A microservice based reference architecture model in the context of enterprise architecture," in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, 2016.

[32] G. Iarossi, *The Power of Survey Design*, 1st ed. Washington DC: The World Bank, 2006.

[33] M. Xu, "Brownout-oriented and energy efficient management of cloud data centers," Ph.D. dissertation, University of Melbourne, 2018.

[34] I. Philippow and M. Riebisch, "Systematic definition of reusable architectures," in *ECBS*, 2001.

[35] E. Y. Nakagawa, F. Oquendo, and M. Becker, "RAModel: A Reference Model for Reference Architectures," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2012.

[36] E. Y. Nakagawa, M. Guessi, J. C. Maldonado, D. Feitosa, and F. Oquendo, "Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures," in *2014 IEEE/IFIP Conference on Software Architecture*, 2014.

[37] Kubernetes, "Services - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/

[38] MSV Janakiram, “Kubernetes: An Overview - The New Stack,” 2016. [Online]. Available: https://thenewstack.io/kubernetes-an-overview/

[39] Kubernetes, “Introducing Container Runtime Interface (CRI) in Kubernetes - Kubernetes,” 2016. [Online]. Available: https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/

[40] containerd, “containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability,” 2019. [Online]. Available: https://containerd.io/

[41] Docker, “Container Runtime with Docker Engine — Docker,” 2019. [Online]. Available: https://www.docker.com/products/container-runtime

[42] CoreOS, “rkt, a security-minded, standards-based container engine,” 2019. [Online]. Available: https://coreos.com/rkt/

[43] Cri-o, “Cri-o - Lightweight Container Runtime for Kubernetes,” 2019. [Online]. Available: https://cri-o.io/

[44] J. Regenold, K. Wang, G. Smith, Q. Liu, and L. Chen, “Enhancing Enterprise Security through Cost-effective and Highly Customizable Network Monitoring,” in *Proceedings of the 10th EAI International Conference on Mobile Multimedia Communications*, 2017.

[45] Kubernetes, “Tools for Monitoring Resources - Kubernetes,” 2019. [Online]. Available: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

[46] Prometheus, “Overview — Prometheus,” 2019. [Online]. Available: https://prometheus.io/docs/introduction/overview/

[47] Sysdig, “Sysdig Monitor: Docker, Kubernetes and Prometheus Monitoring — Sysdig,” 2019. [Online]. Available: https://sysdig.com/products/monitor/

[48] New Relic, “Kubernetes Monitoring — New Relic,” 2019. [Online]. Available: https://newrelic.com/platform/kubernetes

[49] Kubernetes, “Logging Architecture - Kubernetes,” 2019. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/logging/

[50] Google Cloud, “Stackdriver Logging — Stackdriver Logging — Google Cloud,” 2019. [Online]. Available: https://cloud.google.com/logging/

[51] Elastic, “Getting Started — Elasticsearch Reference [6.7] — Elastic,” 2019. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/6.7/getting-started.html

[52] Kubernetes, “Ingress Controllers - Kubernetes,” 2019. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

[53] Daniele Polencic, “Can you expose your services with an API gateway in Kubernetes?” 2019. [Online]. Available: https://learnk8s.io/kubernetes-ingress-api-gateway/

[54] Kong Inc, “Kong FAQ — Kong API Gateway Questions & Answers,” 2019. [Online]. Available: https://konghq.com/faqs/

[55] Ambassador, “Features and Benefits — Ambassador,” 2019. [Online]. Available: https://www.getambassador.io/about/features-and-benefits

[56] Gloo, “Introduction :: Gloo Docs,” 2019. [Online]. Available: https://gloo.solo.io/introduction/

[57] H. Knoche and W. Hasselbring, “Drivers and Barriers for Microservice Adoption-A Survey among Professionals in Germany,” *International Journal of Conceptual Modeling*, 2018.

[58] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation,” *IEEE Cloud Computing*, 2017.

[59] Michael Ducy, “Kubernetes Security Logging with Falco & Fluentd. — Sysdig,” 2018. [Online]. Available: https://sysdig.com/blog/kubernetes-security-logging-fluentd-falco/

A Questionnaire

A.1 Section 1

Does you company make use of a microservice architecture? (Pick one)

- Yes (Go to Section 2)
- No (Go to Section 3)

A.2 Section 2

What deployment strategy to you use? (Pick one)

- Containers
- Virtual machines
- Serverless
- Other (Enter text)

How would you describe the experience of the migration? (Enter text)

Please rate the change in performance of each area compared to before the migration:

- Scalability (1-5, Worse-better)
- Function reusability (1-5, Worse-better)
- Maintainability (1-5, Worse-better)
- Complexity (1-5, Worse-better)

Please rate the disadvantages according to your view of them:

- Complex design and planning phase (1-5, No problem-problematic)
- Complex maintenance (1-5, No problem-problematic)
- Complex testing and deployment (1-5, No problem-problematic)
- Memory consuming (1-5, No problem-problematic)

A.3 Section 3

What are your views on microservices? (Are they essential for big systems? Are they just an overhyped trend?) Please enter some thoughts. (Enter text)

Do you experience problems with complex codebases and/or maintenance? (Pick one)

- Yes
- No

Has your company thought about migrating to a microservice architecture? (Pick one)

- Yes (Go to Section 4)
- No (Go to Section 5)

A.4 Section 4

Please rate the importance of each of the following aspects that you hope to achieve from microservices:

- Scalability (1-5, Not important-very important)
- Smaller codebases (1-5, Not important-very important)
- Function reusability (1-5, Not important-very important)
- Technology flexibility (1-5, Not important-very important)
- Maintainability (1-5, Not important-very important)

Do you think microservices can solve some of the problems that you experience today? (Pick one)

- Yes
- No

Do you think microservices can improve the quality of your system? (Pick one)

- Yes
- No

Please rate the disadvantages according to your view of them:

- Complex design and planning phase (1-5, No problem-problematic)
- Complex maintenance (1-5, No problem-problematic)
- Complex testing and deployment (1-5, No problem-problematic)
- Memory consuming (1-5, No problem-problematic)

A.5 Section 5

What reasons do you have for not wanting to migrate? (Pick one or more)

- Costly and time consuming migration
- The current system works fine as it is
- Microservices are new and therefore lack standards and guidelines
- Too complex
- Other (Enter text)

Please rate the disadvantages according to your view of them:

- Complex design and planning phase (1-5, No problem-problematic)
- Complex maintenance (1-5, No problem-problematic)
- Complex testing and deployment (1-5, No problem-problematic)
- Memory consuming (1-5, No problem-problematic)