Kim Lehtinen

Scaling a Kubernetes Cluster

School of Technology and Innovations
Master's thesis in Automation and
Computer Science

Vaasa 2022

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Kim Lehtinen
Title of the Thesis: Scaling a Kubernetes Cluster
Degree: Master of Science in Technology
Programme: Automation and Computer Science
Supervisor: Prof. Timo Mantere
Instructor: Prof. Timo Mantere
M.Sc. Mika Filander
Year: 2022 **Pages:** 73

ABSTRACT:
Kubernetes is a container orchestration tool that has become widely adopted for deploying and scaling containers. Devatus Oy as well as their subsidiary company Fliq Oy are interested in knowing how containerized applications can be scaled on Kubernetes. The objective of this thesis is to research how a Kubernetes cluster can be scaled as well as containerized applications running on Kubernetes.

This thesis begins with an introduction to necessary background knowledge needed to understand what Kubernetes is. Cloud computing and distributed systems are introduced, since Kubernetes is a distributed system used in cloud environments for the most part. Furthermore, distributed applications and workloads are introduced through the concept of microservices. The concept of containerizing applications is thoroughly introduced to understand the runtime environment of the applications deployed to Kubernetes. Finally, Kubernetes architecture as well as its main components are introduced to understand how container orchestration works.

The research on Kubernetes scalability is divided into three different parts. First part consists of researching how containerized applications can be scaled on Kubernetes. Second part is focused on how the Kubernetes cluster itself can be scaled. The final part consists of load testing one of Fliq's example REST API applications deployed to a local Kubernetes cluster. The purpose of load testing is to gain further insight into scaling applications running on Kubernetes. Load test results are compared between the initial deployment configurations and after scaling the application.

The load test results show that containerized applications can be scaled both vertically and horizontally. Vertical scaling can be achieved by increasing the requested and limited CPU and RAM resources for a Pod. Horizontal scaling can be achieved by increasing Pod replicas as well as having a Service in front of the Pods that load balances the incoming traffic. Load test results show that both vertical and horizontal scaling can increase the number of users supported by an application deployed to Kubernetes. Scaling horizontally is preferred for Fliq's example REST API since it decreased average response time and increased throughput.

KEYWORDS: Kubernetes, Cloud Native, Cloud Computing, Scalability, Load testing

Contents

Figures

Tables

Abbreviations

API	Application Programming Interface
CNCF	Cloud Native Computing Foundation
CLI	Command Line Interface
CPU	Central Processing Unit
CRI	Container Runtime Interface
HPA	Horizontal Pod Autoscaler
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
OCI	Open Container Initiative
OS	Operating System
RAM	Random Access Memory
REST	Representational State Transfer
SQL	Structured Query Language
VM	Virtual Machine
VPA	Vertical Pod Autoscaler
YAML	YAML Ain’t Markup Language

Acknowledgement

To begin with, I would like to thank Devatus and Fliq for an interesting thesis project. I have learned a lot about scaling applications and container technologies. These are valuable skills that will be useful for my career.

This thesis project would not have been possible without the people I have around me. I want to thank my thesis instructor and boss Mika Filander for making this thesis project happen and for always supporting me. Finally, I want to thank my fiancée, family, and friends for always being there for me.

Kim Lehtinen
Vaasa, 28.2.2022

1 Introduction

The use of Cloud Native technologies has increased over the last few years. Instead of building monolithic applications and using virtual machines, companies are moving towards container technologies and distributed systems. One of these technologies is Kubernetes, a container orchestration tool designed to run containerized applications at scale. The purpose of this thesis is to study how an application running on Kubernetes can be scaled.

1.1 Project founders

This thesis project is done for both Devatus Oy and their subsidiary company Fliq Oy. Devatus Oy is a software development company, that specializes in developing digital services for industrial companies (Devatus 2020). In addition to digital service development, they offer cloud solutions, data analytics and IoT solutions.

Fliq Oy, on the other hand, is a software development company that specializes in smart factory solutions for industrial companies (Fliq 2020). Fliq Oy offers a smart factory product called Fliq, which is a cloud-based product for companies to follow up on industrial processes and visualizing data that is gathered from IoT sensors.

1.2 Objective

Fliq Oy has recently moved from traditional monolithic applications to splitting their applications into smaller services, called microservices. In addition, they have chosen to package and run these microservices inside containers. Furthermore, they decided to manage and orchestrate their containers using Kubernetes.

Devatus Oy and Fliq Oy are interested in how to scale Kubernetes clusters. The purpose of this research is to understand how applications running on Kubernetes can be scaled and find possible defects and challenges when running applications on Kubernetes. This is done by studying different methods for scaling applications running on Kubernetes and how the cluster itself can be scaled. In addition, load testing is performed against an example application running in a test cluster to get a better understanding of scaling applications on Kubernetes.

This thesis begins with an introduction to cloud computing, since Kubernetes is a technology used in cloud. Kubernetes clusters can consist of distributed servers and applications, which is why distributed systems are also introduced. REST APIs are introduced shortly since load tests are performed against an example REST API application running on Kubernetes. The concept of load testing is finally introduced to understand how it can be used to test the scalability of applications running on Kubernetes.

Before investigating Kubernetes scalability, one must first understand core concepts of a technology like Kubernetes. This done with an introduction to several cloud native computing concepts like microservices, container technologies and container orchestration. In addition, key concepts in Kubernetes are explored to further understand the technology.

In order to understand how a Kubernetes cluster can be scaled, several scaling methods are researched. To begin with, Kubernetes monitoring is researched to understand how resource metrics can be retrieved from applications running on Kubernetes. Secondly, scalability methods on the application level are researched to understand how applications can be scaled when running on Kubernetes. Finally, the scaling of the Kubernetes cluster itself is studied to understand how cluster resources can be scaled.

To gain even further insight into how a Kubernetes cluster can be scaled, load tests are performed against an example application running in a local Kubernetes cluster. This is

achieved by first creating a local Kubernetes cluster which is used as test environment. Load tests are performed against an example application that is deployed to the local cluster. After the initial load test, scalability methods are applied, and new tests are executed. The load test results are analyzed to see if the scalability methods work. In addition, future ideas for scaling the application are discussed based on the scalability methods researched in this thesis and load testing results.

1.3 Structure

The second chapter of this thesis is an introduction to distributed cloud computing, which consists of theoretical background to understand cloud computing, distributed systems, REST APIs and performance testing. Chapter 3 introduces Cloud Native Computing, which consists of introduction to microservices, container technologies and Kubernetes. In chapter 4, the research of scaling a Kubernetes takes place. In chapter 5 the test environment is shown. Chapter 6 is scalability testing of an example application running on a local Kubernetes cluster. The final chapter is the thesis conclusion.

2 Cloud Computing

Since the focus of this thesis is scalability testing of a Kubernetes cluster, one must first understand the building blocks of cloud computing and distributed systems. This chapter is an introduction to cloud computing, distributed systems, REST APIs and performance testing. Information presented in this chapter is also important to understand the next chapter, where cloud native computing is introduced.

2.1 Introduction to Cloud Computing

Cloud computing is a term heard often these days in the IT sector. While it may sound like new thing, (Wang, Ranjan, Chen, & Benatallah 2011: 4–5) states that cloud computing is based on older ideas of computing, and historical changes in society. There was a time in history when people came up with the idea that they can create a business where they provide services and resources that people need and do not want to maintain themselves such as electricity power Wang says. Cloud computing is an effect of the same kind of revolution, computing power and resources can now be sold and distributed in the same way.

According to Wang (2011: 4–5), sharing computer power and resources is also not a completely new invention. If one looks at the evolution of computing, the earlier versions of computers were shared among users before personal computers (PC) were invented. After the Internet revolution, it made sense again to share computing resources to PCs and mobile phones via the Internet.

Marinescu (2013: 1) says that cloud computing offers computing power and resources via the Internet to users in a flexible way. A user of cloud computing only has to pay for what is actually used. Marinescu also states that cloud computing is a successor of utility computing, which introduced the business idea and model of sharing computing

resources to users. The area of cloud computing began when big companies started offering these kinds of services to users.

Cloud computing infrastructure can be deployed in several ways. Wang (2011: 11–13) explains three different techniques used to deploy cloud infrastructure. The most popular technique is public cloud, which means that instead of doing cloud computing oneself, cloud computing is provided by other companies through the public internet, and thus anybody can use these resources as needed in exchange for money. The opposite to this is private cloud, where the cloud computing infrastructure is internal and not available for anyone to use or buy. Last technique described by Wang is hybrid cloud, which is a combination of public and private cloud, where one can choose what part of the cloud computing infrastructure is exposed to the public internet, and what part should remain private. These deployment techniques described by Wang are demonstrated in figure 1.

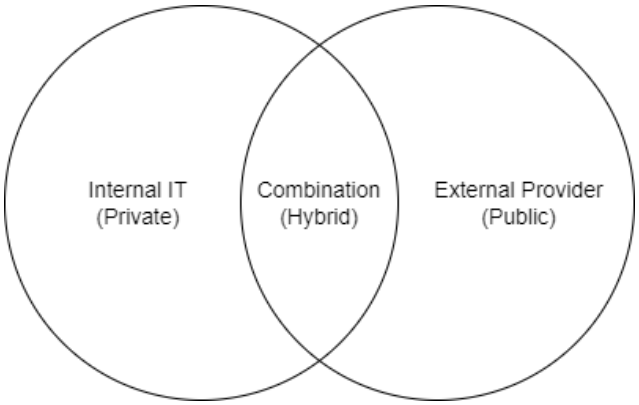


Figure 1. Techniques used to deploy cloud infrastructure (Based on Wang 2011: 12)

In addition to deployment techniques, Wang (2011: 13–14) says that there are three popular types of cloud services: Infrastructure as a Service (IaaS), Software as a Service (SaaS) and Platform as a Service (PaaS). What is common between these services is that users of these services only pay for what they use, and these services offer different types of cloud computing resources. According to Wang, IaaS offer infrastructure services like storage or virtual machines. PaaS on the other hand, is built on top of IaaS and works as a platform that can be used to create new products and applications, by

providing cloud services like software testing, application deployment, databases etc. SaaS is software that is accessible via the Internet, usually in a web browser Wang says.

2.2 Distributed Systems

Distributed systems come in different shapes, layers and have various definitions. These terms are often heard in IT today, especially cloud computing. In this chapter, distributed systems are introduced both at hardware and software levels of computing.

Marinescu (2013: 27) describes distributed systems as a set of interconnected computers. A software layer called “middleware” is used to connect computers to each other by exposing a network channel interface. Middleware is what glues these computers together and allows computers to share computing resources. Furthermore, Marinescu mentions that common characteristics of middleware are system scalability, information sharing, concurrency, and information accessibility.

According to Puder, Römer, Pilhofer, & Romer (2005: 8), a distributed system can also be seen as interconnected processes, in addition to computers. A distributed system running several computer processes can run either on the same machine or multiple machines. No matter if a distributed system consists of computers or processes, they share a similar model where a set of nodes are connected and can communicate with each other.

2.3 REST API

In this thesis, an example application is used to test Kubernetes scaling methods. This application is deployed to a local Kubernetes cluster where it is load tested. This chapter explains what a REST API is since the example application is of this type.

There are numerous ways to design and architect applications. Applications often provide an interface through which they can be used by other systems, called Application Programming Interface (API) (IBM 2020). One common way to architect web-based APIs is to use Representational State Transfer (REST).

REST APIs are web-based applications that can be consumed through HTTP/HTTPS protocol. These applications provide resources that can be accessed through an exposed API by following a set of rules. REST APIs follow the client-server model, where the client can access the API by sending HTTP requests including a URI for a specific resource that the API exposes. (Kanjilal 2013: 24–25).

REST APIs support a set of HTTP methods. For retrieving resources, the GET method is usually used. When new resources are created, the POST method is used. For modifying resources, the PUT method is preferred. DELETE method can be used to remove resources. Finally, the HEAD method can be used for accessing HTTP headers. (Kanjilal 2013: 26).

2.4 Performance testing

Load testing is used in this thesis for testing Kubernetes scalability methods. This chapter introduces what performance and load testing is. In addition, JMeter which is the load testing tool used in this thesis is introduced.

2.4.1 Performance and load testing

Performance testing can be used to find out system performance. The system being tested can for example be an application, server, or network. The tests can be performed on a complete system or parts of it. The benefit from doing performance testing is that

it can be used to check if the system under test is able to operate in different conditions (Erinle 2013: 23).

Load testing is one type of performance testing. It can be used to test how much load can be applied on the system under test (Erinle 2013: 29). If the system is an application, load testing can for example be used to find out the maximum number of users it is able to support (BlazeMeter 2019).

2.4.2 JMeter

JMeter is an open-source performance testing tool. It was first created in 1998 by The Apache Software Foundation. JMeter can be used to test several application types, for example web applications, databases, or email. Being multithreaded, JMeter is able to create test scenarios for high user load. (Erinle 2013: 30).

New features can be added to JMeter by installing plugins. One plugin used in this thesis is Concurrency Thread Group. It can be used to setup concurrent threads for testing user load (BlazeMeter, 2016). This plugin is used in this thesis to do load tests that simulate high user load.

3 Cloud Native Computing

Cloud Native Computing Foundation (CNCF) is a foundation that originated from the Linux Foundation, with the intention of supporting open-source cloud native projects (CNCF 2021a). These projects go through different stages of maturity, to help companies find suitable solutions (CNCF 2021b). CNCF organize conferences around the world, to help build a cloud native community that connects companies, software developers and users of the projects that CNCF supports (CNCF 2021a). In this research, several cloud native technologies are used that are supported by CNCF.

Cloud native computing has over the recent years become popular within the cloud computing landscape. The term cloud native itself has numerous definitions and can therefore confuse people. CNCF (2018) have their own definition for what cloud native is, and part of it is shown below.

> *Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.*
> *These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.*

To conclude, cloud native technologies are technologies that encourage its users to deploy distributed software to the cloud in a fast-changing environment (CNCF 2018). CNCF's definition shows that cloud native computing has derived from cloud computing, by encouraging features of cloud computing, for example, deployment techniques and latest innovations within cloud computing. Cloud native computing focuses more on getting the best out of cloud computing.

The rest of this chapter goes over the most basic concepts and technologies of cloud native technologies. First, the idea of microservices is introduced to get a better understanding of distributed software. Secondly, containers are introduced to understand how

to package and run software in cloud native environments. Finally, container orchestration and Kubernetes are introduced since these are core concepts of understanding the rest of this research. The knowledge presented in this chapter is used in later chapters to understand how Kubernetes clusters can be scaled.

3.1 Microservices

In this chapter, microservices architecture model is introduced. This architecture model aligns with the cloud native philosophy: applications should be easy to distribute and decoupled (CNCF 2018). This chapter compares monolithic applications with microservices and shows why microservices are suitable for scalable and distributed systems.

For a long time, companies have faced the difficulty of scaling software and cloud infrastructure due to the advancement of digitalization. In addition, the maintenance of source code has also become challenging due to applications growing larger. As a result of many attempts and different solutions to scale software, microservices has derived as an alternative software architectural design. (Newman 2015: 1–2).

Traditionally, a software system is usually a monolithic application. In this simple architecture, a software system is one application only, containing all code and functionalities for that system. Large monolithic web services often consist of the following components all in the same package: frontend (web client), backend (web server) and database. (MuleSoft 2020).

The drawback of monolithic applications is that whenever a programmer makes a change to any layer of a monolithic application, the whole application has to be rebuilt and released whenever a new deployment is made (MuleSoft 2020). This aligns with Newman saying that source code is getting harder to maintain. When all of the source code is found in one place only, and the application consists of many components and layers in the same binary, the project is harder to maintain in the long run.

The goal of microservices is to scale software by dividing it into smaller services, where each service is designed for a specific purpose or task, hence the name microservices. The services can for example be divided by task or business functionality in order to give each service a meaningful purpose. This type of division makes sure that the size of a microservice stays small compared to a monolithic application. (Newman 2015: 1–2).

Microservices are typically APIs by design. As a result, they can work together by sending requests to each other. This minimalistic and practical design makes microservices ideal for distributed systems, since they can be deployed separately and still manage to work together. (Newman 2015: 3).

A great advantage that comes with microservices is that the same technology doesn't have to be used everywhere or solve all problems. Each service can be implemented with the technology that is best suited to solve a specific problem. The services can still communicate with each other as long as they continue to communicate through their exposed APIs. (Newman 2015: 4).

In figure 2 an example based on one of Newman's (2015: 4) examples is demonstrated. Here three different microservices are shown: Posts, Users and Pictures. This could for example be a social media application where users can write posts and upload pictures. Each service is implemented with different programming language, and they use different databases for storage. In this example, Golang was best suited for handling user relations together with a SQL database. For storing posts, the combination of a Node.js API server and NoSQL database was the most optimal solution, and for images a Java application paired with a blob storage was a good solution. The point of this example is to show that microservices open the possibilities for selecting the best technology to solve a specific problem (Newman 2015: 4). In addition, this example shows how a monolithic application can be split into microservices where each service is designed for a specific task, and still manage to work together.

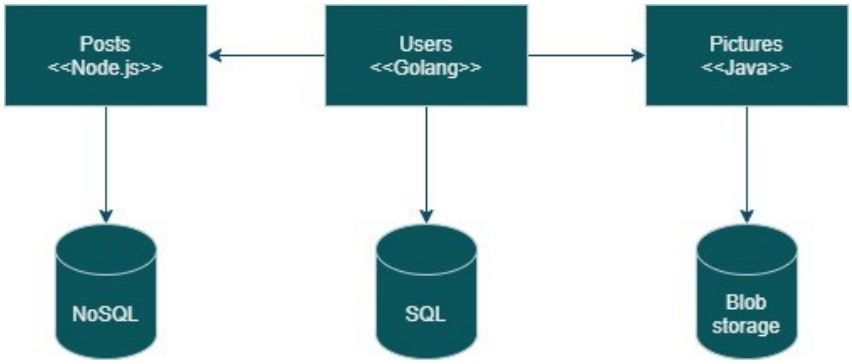


Figure 2. Microservices (Based on Newman 2015: 4)

When a monolithic application experiences a problem, all parts of that application suffer as a result (Newman 2015: 5). If the social media application shown in figure 5 would have been a monolithic application, and the “Pictures” module would experience severe problems, the application would not be able to handle user requests, posts, or pictures since the whole application is a single unit that is experiencing a problem. If the same application uses microservices, and the “Pictures” service experiences downtime, all other services are still usable.

Microservices architecture is often used in scalable and distributed systems. In a monolithic application one is not able to choose which part of the application should be scaled (Newman 2015: 5). If the social media application shown earlier was a monolith, the whole application would have to be scaled even if the "Posts" feature would be the only one experiencing performance issues. When using microservices, one is able to specify which service should be scaled (Newman 2015: 5).

In figure 3 the scalability of microservices is demonstrated. This is the same application as shown in figure 2 where a social media application has three microservices. Here the number of instances of each service has been scaled accordingly. Microservices architecture offers the possibility to replicate a specific service by deploying multiple instances. As a result, one can choose which part of a system has to be scaled and by how much. In figure 3, if the “Posts” service is the most demanding, it can be replicated three times for example. The “Pictures” service is the least demanding and only needs one instance.

Figure 3. Microservices in distributed systems (Based on Newman 2015: 6)

3.2 Container Technology

The focus of this thesis is to understand how a Kubernetes cluster can be scaled. Kubernetes is a platform for running and distributing applications inside containers, which is why one must first understand the basics of container technology before learning what Kubernetes is. This chapter introduces container technologies by comparing containers with virtual machines and the basics of Docker containers. Docker is one of the most popular container runtimes and often used together with Kubernetes.

As of Kubernetes version 1.20, Docker as a container runtime on Kubernetes has been deprecated. The reason for this is that Docker provides a lot of features in addition to the runtime that are not necessary for Kubernetes. Docker was never intended to be integrated into Kubernetes, while other container runtimes are by implementing a Container Runtime Interface (CRI). One of these is “containerd” which is the runtime that Docker uses under the hood. Therefore, Kubernetes decided to deprecate Docker since containerd can be used without Docker. (Kubernetes 2020a).

According to Kubernetes (2020a), even if Docker is not used as the container runtime, applications built using Docker will still run on Kubernetes since CRI-compliant container runtimes uses OCI (Open Container Initiative) images to run containerized applications. Applications built and packaged using Docker produces OCI images and will therefore run on Kubernetes (Kubernetes 2020a).

Docker provides a good technology for packaging software into OCI-compliant container images and running containers locally. Therefore, Docker is used in thesis to explain container technology concepts. Chapter 3.2.1 is an introduction to what containers are. In chapter 3.2.2 containers are compared with virtual machines (VM) in order to understand the difference between these two popular virtualization technologies.

3.2.1 Introduction to containers

Containers became popular and more accessible in the last decade when container technologies like Docker were introduced (D2iQ 2018). However, the technology and idea of packaging and running software inside isolated containers is older (D2iQ 2018). Below is how Google Cloud (2022) describes what containers are.

> *Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of whether the target environment is a private data center, the public cloud, or even a developer's personal laptop.*

According to Google Cloud's (2022) definition of what containers are, container technologies offer a more universal way of packaging and running applications across environments. All of the source code and libraries that is needed to run an application can be put inside the container (Docker 2021a). To conclude, containers can be used to isolate and distribute software.

3.2.2 Containers vs Virtual Machines

In this thesis both containers and virtual machines are used. Containers are used to run containerized applications on a Kubernetes cluster, and virtual machines are used to create a local Kubernetes cluster for scalability testing in chapter 6. In this chapter these two virtualization technologies are compared to understand the difference between them.

Figure 4 shows the difference between virtual machines and containers. Starting from the lowest level, both virtualization technologies are dependent on an underlying infrastructure in the form of a computer together with an operating system running on it (Poulton 2020: 71). The first difference in VMs is that they are dependent on a hypervisor to virtualize physical resources (Poulton 2020: 73). Containers don't need a hypervisor since the virtualization happens on the operating system (OS) level (Poulton 2020: 73). The final difference between containers and VMs are that an OS has to be installed in each VM, while multiple containers can use the same host OS (Poulton 2020: 73).

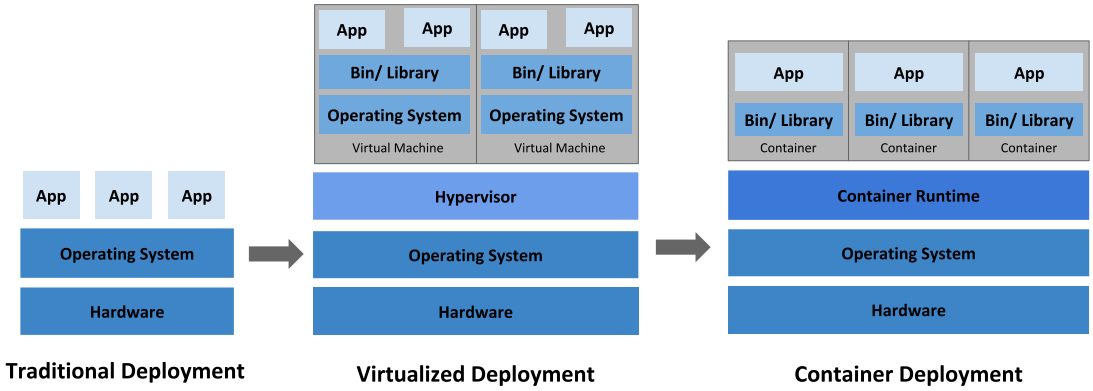


Figure 4. Container vs VM (Kubernetes 2021a). License: CC BY 4.0.

Figure 4 also shows the benefit of separating applications from each other either using VMs or containers. Before these technologies, all applications were running on the same server. The only way to truly separate applications was to add more servers, which would result in unused resources. VMs solved this problem by creating several VMs on the

same host. Containers are able to solve the same problem with less overhead, since containers can use host OS. (Kubernetes 2021a).

Containers take less time to create due to VMs having to install an entire OS each time during creation (Poulton 2020: 74). The computer on which the containers will be running on has a running OS ready for use (Poulton 2020: 74). Poulton (2020) concludes containers as a more cost-effective solution followingly "You can pack more applications onto less resources, start them faster, and pay less in licensing and admin costs, as well as present less of an attack surface to the dark side" (Poulton 2020: 74). These benefits can make containers a compelling option when deciding how to run and scale software.

3.2.3 Container image

Container images are needed in order to create and run containers. They contain everything that is needed to run an application in a container. It is common for container images to build upon other images called “base images”. (Microsoft 2021).

When using Docker as the technology to create container images, a file named "Dockerfile" is used. This file can be used to tell Docker step by step how a docker image should be created. Each step in the Dockerfile can be thought of as a command to tell Docker what to do. (Docker 2021b).

Below is an example of a Dockerfile code created by Docker (2021b). The first command is "FROM", which tells Docker to base the new container image upon ubuntu container image. The application source code is copied into the container image using the "COPY" command. The "RUN" command compiles the application code. Finally, the "CMD" command starts the application when the container has been created. (Docker 2021b).

```
# syntax=docker/dockerfile:1
FROM ubuntu:18.04
COPY . /app
```

```
RUN make /app
CMD python /app/app.py
```

The example above complies with Microsoft (2021) definition of a container image. This example container image uses ubuntu v18.04 as base container image. An application together with its dependencies is copied into the container image, and the application is compiled inside the container image. The base image has python installed, which is used to run the main application file. Therefore, it can be concluded that this container image includes everything needed to run the application.

3.2.4 Container registry

When a container image has been built, it can be run locally. However, in order to let other people and servers access the same container images, a container registry is needed. Container registries can be used to accumulate and distribute container images (Poulton 2020: 51).

In figure 6, the flow of creating and running container images using a container registry is shown. Everything starts with having an application that should be built and deployed. A “Dockerfile” is used to package the application source code and its libraries into a container image (Poulton 2020: 89). In order to be able to distribute the container image, it is sent to a container registry where it will be stored and is accessible by others.

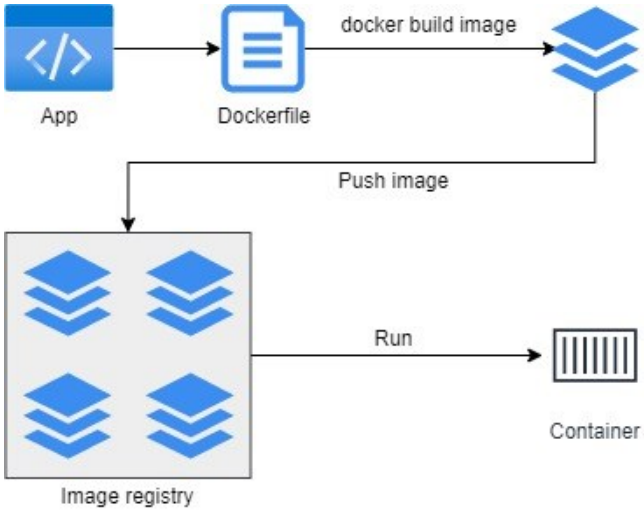


Figure 5. Docker image and container flow (Based on Poulton 2020: 89)

Using container registries allows another person or machine to access the same container image someone else has built. If the intent is to deploy the application to a production server, the server can pull the container image from the container registry and run the application inside a container, without having to understand how the application should be built. The container image can be executed as it is, including everything that is needed to run the application.

3.3 Kubernetes

In the previous chapter containers as a technology was introduced to explain what they are and what they do. In this chapter, the container orchestration technology used throughout this thesis is introduced, called Kubernetes. The goal of this thesis is to understand how applications can be scaled on Kubernetes. Before learning how to scale a cluster, one must first understand what container orchestration is and what Kubernetes does. That is the focus of this chapter.

Managing container workloads without a container orchestration tool is possible. However, this becomes harder at a larger scale when containers have to be scalable and distributed on multiple servers. This is where a container orchestration tool like Kubernetes

comes in to solve this problem. Kubernetes (2021a) describes Kubernetes as "…a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation".

Kubernetes is an open-source project that has derived from a container orchestration tool at Google called "Borg" (Kubernetes 2015). Google has used Borg to manage clusters and containerized applications for many years (A. Verma, L. Pedrosa, M. Korupoly, D. Oppenheimer, E. Tune & J. Wilkes 2015). The lessons learned from building Borg and other orchestrations tools at Google have been used to create Kubernetes (B. Burns, B. Gant, D. Oppenheimer, E. Brewer & John Wilkes, 2016).

With the help of Kubernetes, it is possible to build a cluster consisting of multiple servers. Kubernetes takes care of managing and distributing container workloads in the cluster. The system administrator can tell what Kubernetes should do, by defining something called the "desired state" (Kubernetes 2021a). Kubernetes takes care of keeping the cluster in the desired state by comparing its actual state (Kubernetes 2021a). This is how Kubernetes manages to achieve things like automatic deployments, rollbacks, and self-healing (Kubernetes 2021a).

In figure 6, a high-level architecture of Kubernetes is shown. This architecture overview shows the main components in a Kubernetes cluster. This figure shows that Kubernetes forms a cluster by joining multiple server nodes together. These nodes can be of any machine type that has a container runtime installed, which allows for Kubernetes to be installed both in the cloud and on bare metal servers (R. Muddinagiri, S. Ambavane & S. Bayas 2019: 240). This is makes Kubernetes a portable container orchestration technology.

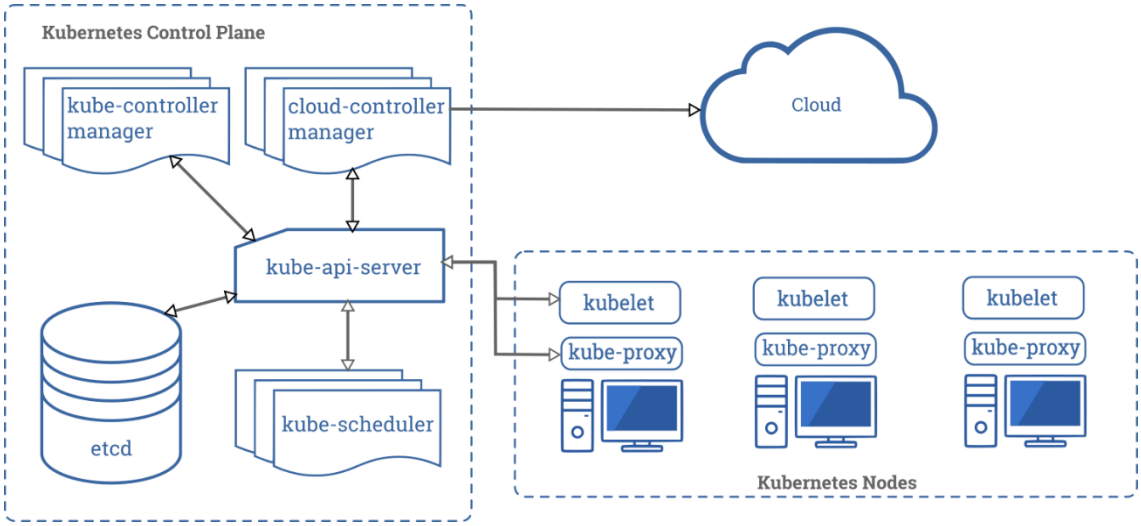


Figure 6. Kubernetes components (Github 2020a). License: CC BY 4.0.

One of the nodes in a Kubernetes cluster is assigned to be the control plane. Its job is to manage the whole cluster, making sure that the actual state matches the desired state. The first key component in the control plane is the “kube-apiserver”, which serves as a gateway to Kubernetes API. The “kube-scheduler” component takes care of creating and distributing containers to available nodes. The “kube-controller-manager” is the component that makes sure the actual state matches the desired state. The “etcd” component is the database where the desired state of the cluster is stored. Finally, the “cloud-controller-manager” is an optional component that can be used to integrate the cluster with a cloud vendor’s API. (Kubernetes 2021m).

In figure 6, the “Kubernetes nodes” are the remaining worker nodes in a cluster, where the actual workload is happening. The containers deployed to a Kubernetes cluster are running in something called a “Pod”, which are introduced in chapter 3.3.1. Each worker node has a container runtime installed, making it possible to run container workloads (Kubernetes 2021m). In addition, they have a component called “kubelet”, which Kubernetes (2021a) describes as “An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.” (Kubernetes 2021a). The last component in a worker node is “kube-proxy”, which is responsible for networking (Kubernetes 2021a).

The term object in Kubernetes is used to describe units in the cluster (Kubernetes 2021b). They can be used for example to configure the cluster or running applications (Kubernetes 2021b). Objects can be configured either imperatively or declaratively (Kubernetes 2021c). The rest of this chapter introduces common objects in Kubernetes and how they can be configured.

3.3.1 Pod

The Pod object is the lowest level object in Kubernetes. Pods can consist of several containers. However, there is usually only one container in each Pod. The possibility to have multiple containers in one Pod can be useful in some situations, for example if they are highly dependent on each other and always coexist. (Kubernetes 2021d).

Pods are usually never deployed separately. Kubernetes has other objects specialized for both creating and managing Pods for different scenarios. Examples of these are: Deployment, Job, StatefulSet and DaemonSet. (Kubernetes 2021d).

Kubernetes objects can be created using only the command line. However, usually they are created using YAML files (Kubernetes 2021b). The code example below by Kubernetes (2021e) shows how a Pod manifest YAML file can look like. The kind field specifies object type, metadata works as identification, and spec specifies the desired state for the Pod (Kubernetes 2021b). In the code example below, the Pod consists of one container, running a NGINX web server container image.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
```

```
      containerPort: 80
      protocol: TCP
```

3.3.2 Deployment

The Deployment object in Kubernetes is used to deploy containerized applications in Pods. Common purposes for the object are to create, edit, or delete Pods running a particular application. In addition, the Deployment object supports rolling back to a previous version or scaling up the number of Pod instances running an application. (Kubernetes 2021f).

Below is an example of what a Deployment looks like by Kubernetes (2021f). The replicas field will create an object of type ReplicaSet that takes care of making sure that 3 Pod instances are running. The spec field decides what container image should be deployed to the Pods. The label fields under metadata and selector tells the Deployment which Pods to administer. (Kubernetes 2021f).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

3.3.3 Service

The Kubernetes Service object is “An abstract way to expose an application running on a set of Pods as a network service.” (Kubernetes 2021g). The Kubernetes cluster gives a Service object a DNS-name, that can be used to discover a group of Pods. In addition, the Service object can load-balance traffic between multiple Pods. (Kubernetes 2021g).

When Pods are deployed to a Kubernetes cluster, they get their own IP address. The problem with using these IP addresses is that Pods are mortal. If a Deployment’s desired state changes, Pods might get deleted as result since Kubernetes always has to make sure that the actual state matches the desired state. The Service object solves this problem by acting as a portal to a group of Pods. (Kubernetes 2021g).

The code example below by Kubernetes (2021g) shows what a Service object can look like. The selector field is used to tell Kubernetes that this Service belongs to all Pods with the same label key value. The “targetPort” field specifies on which TCP port the Pods are listening on, and “port” is the port the Service listens on. (Kubernetes 2021g).

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

There are different types of Services in Kubernetes. The service type can be specified by adding type field under the spec field in a manifest file. The default type is ClusterIP, which gives the Service an internal IP address that can’t be accessed from outside the cluster. The NodePort type can be used to open a port on each node that makes the Service externally accessible. The LoadBalancer type creates a load balancer for the cloud provider the cluster is using, which makes the service accessible to anyone. (Kubernetes 2021g).

3.3.4 Ingress

Ingress is a concept in Kubernetes designed to control HTTP traffic between the cluster and the outside world. It can for example be used to forward incoming traffic to a specific Service. The configuration for how and where the incoming traffic is forwarded can be done by creating Ingress rules. The Ingress rules can be used to specify where a set of URL paths and hosts should be forwarded. (Kubernetes 2021h).

In order for Ingress rules to be applied, the cluster must have at least one Ingress controller installed that takes care of actually doing what has been specified in the Ingress rule (Kubernetes 2021h). Ingress controllers are not installed by default in the cluster (Kubernetes 2021h). This makes it possible to select the most suitable Ingress controller for a specific Kubernetes cluster. According to Kubernetes (2021h), any Ingress controller should work in theory. This separation between Ingress rules and Ingress controllers abstracts away the underlying technology that takes care of doing the actual the work.

The code example below based on Kubernetes (2021h) shows what an Ingress rule manifest can look like. The rules array field allows for multiple rules to be defined. In this case, there is a rule that the domain “example.com” should point to a Service named “example-service” at port 80. The path field can be used to route a specific path only to a Service. However, in this example the path is set to “/” which means the rule is applied to all paths.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-example
spec:
  rules:
  - host: example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
```

```
service:
  name: example-service
  port:
    number: 80
```

In figure 7 the purpose of having an Ingress is demonstrated. Ingress works as the bridge between HTTP clients and the applications running on Kubernetes (Kubernetes 2021h). HTTP requests first goes through the Ingress and is forwarded to a Service based on Ingress rules. One rule could for example be to forward traffic to a specific Service based on the domain name “example.com” as the manifest example above. The Service finally forwards the traffic to one of the Pods it exposes. This example is an end-to-end demonstration of how applications running in Pods can be exposed to the outside world with the help of Service and Ingress objects in Kubernetes.

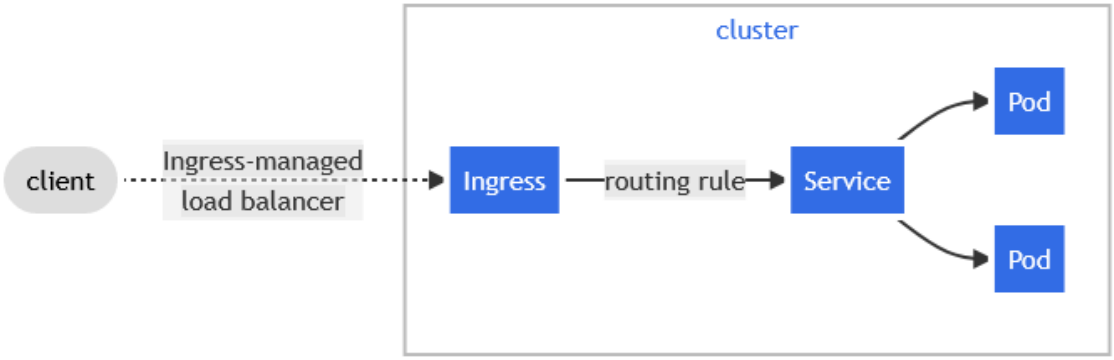


Figure 7. Ingress (Kubernetes 2021h). License: CC BY 4.0.

3.3.5 Kubectl

Kubernetes clusters can be managed using *kubectl* CLI. It can for example be used for creating or editing resources. In addition, a common use case is to get information about resources running in a Kubernetes cluster. (Kubernetes 2021i).

In order for kubectl to be able to interact with a Kubernetes cluster, a *kubeconfig* file is needed. This file is used to switch between clusters and perform the needed authentication to be able to interact with Kubernetes API running in the cluster, using kubectl. It

is possible to have multiple kubeconfig files and tell kubectl which one to use. (Kubernetes 2021j).

Kubernetes objects can be managed using kubectl either imperatively or declaratively. The imperative approach is to perform specific kubectl commands to manage resources. Below is two imperative kubectl command examples by Kubernetes (2021c). Both examples deploy the same application in two different ways. The first example uses commands only for achieving the deployment, called *imperative commands*. The second example uses both commands and a YAML manifest file where a Deployment object has been described, this is called *imperative object configuration*. Both examples are imperative since kubectl is told specifically to create something. (Kubernetes 2021c).

```
kubectl create deployment nginx --image nginx
kubectl create -f nginx.yaml
```

The declarative approach does not tell kubectl specifically what to do, this is called *declarative object configuration*. Instead, kubectl is only given YAML files or directories containing YAML files to process. Kubernetes automatically knows what to do based on the contents of the manifest files. Below is an example of a declarative approach by Kubernetes (2021c). All manifest files inside a “configs” directory will be applied by kubectl. If an object described in a manifest file does not exist, kubectl will automatically create that object. However, if the object already exists and the manifest file has changed, kubectl will automatically update that object. (Kubernetes 2021c).

```
kubectl apply -f configs/
```

All approaches for managing Kubernetes objects with kubectl have their pros and cons. Imperative commands are simple and fast to execute. However, the changes are not described anywhere and cannot be reused. Imperative object configuration solves these shortcomings by describing the actions to be taken in manifest files. The drawback of this approach is that it is more laborious compared to a few writing commands.

Declarative object configuration is better for applying folders containing manifest files and knowing what to do with them. The drawback of using the declarative approach is knowing why something is not working when a lot of changes have been applied. (Kubernetes 2021c).

4 Scaling a Kubernetes Cluster

In this chapter, different ways of scaling a Kubernetes cluster are studied. These scaling methods are studied to learn how applications can be scaled on Kubernetes. This chapter begins by studying how a Kubernetes cluster can be monitored in order to understand how metrics can be retrieved. Vertical and horizontal scaling are studied to understand different ways of scaling Pods. Cluster autoscaler is studied to understand how the server nodes can be scaled.

4.1 Monitoring

This chapter introduces how a Kubernetes cluster can be monitored. It introduces components and technologies that can be installed in a cluster for scaling purposes. These technologies are used in the scalability testing chapter.

4.1.1 Metrics Server

Kubernetes Metrics Server is a service that can be installed in a Kubernetes cluster to get information about cluster resources. The Metrics Server is able to retrieve this information from the kubelet component which is found on all nodes in a cluster. This data can be accessed via Kubernetes API server which is extended by an additional Kubernetes Metrics API. The Metrics Server is designed for autoscaling purposes only. It has to be installed in order to use Horizontal Pod Autoscaler and Vertical Pod Autoscaler scaling methods. (Oracle 2021).

4.1.2 Prometheus

Brazil (2018) describes Prometheus as "…an open source, metrics-based monitoring system". It was originally created by Sound Cloud, and today it is part of CNCF. Prometheus

is built for unifying metrics from multiple data sources. It can for example retrieve metrics from applications, servers, and other monitoring systems. In the case of Kubernetes, it can automatically find nodes and applications to retrieve metrics from. The metrics can be used as data source for visualization tools like Grafana. (Brazil 2018: 3–4).

Prometheus is able to discover Kubernetes objects and nodes to retrieve metrics from through the Kubernetes API server. All nodes in a Kubernetes cluster have a kubelet component which is used to retrieve metrics about nodes. For applications running in Kubernetes, Prometheus is able to scrape all exposed container ports inside a pod. (Brazil 2018: 159–166).

Prometheus is used in this thesis to retrieve metrics from applications running in a Kubernetes cluster. These metrics are used as data source in Grafana dashboards for visualizing load testing results.

4.1.3 Grafana

Grafana is a tool designed for data analytics at scale. It is open-source, flexible and easy to integrate with various data sources and other monitoring tools. When installed, it offers a dashboard that can be used to visualize and analyze data. (Shivang 2019).

When Kubernetes clusters are monitored using tools like Prometheus, a lot of data is collected about the cluster, nodes, and application workloads. In order to analyze this data, Grafana is used in this thesis as a data visualization tool. The Grafana dashboard is used to analyze Prometheus metrics collected when load testing is performed.

4.1.4 Helm

Helm is a tool designed for packaging applications together with their configurations for Kubernetes. This is done by creating Helm Charts that packages all needed YAML configuration files for a specific application. These charts can be versioned and distributed via repositories. This makes it easier to install applications on Kubernetes clusters. (D. Merron & T. Idowu 2020).

Helm is used in this thesis to help installing Prometheus and Grafana. These tools are often used together in Kubernetes for monitoring a cluster. There are several helm packages available for installing Prometheus and Grafana in a Kubernetes cluster.

4.2 Horizontal pod scaling

Scaling horizontally means to increase the number of compute instances. To begin with, one can start off by running only one instance, and add more instances later when there is more demand. The advantage of doing horizontal scaling is that if an instance is experiencing issues, other instances are not affected and can continue to function. (Techopedia 2021a).

When Pods are scaled horizontally, the HTTP traffic can be load balanced between the Pods. This is possible by having a Kubernetes Service in front of the Pods (Kubernetes 2021g). The benefit of sharing the load between multiple instances is that it can decrease HTTP response time (S. Jain & A. K. Saxena 2016). In addition, it can enhance throughput (D. Sharma 2018).

In the Kubernetes world, horizontal scaling is done through increasing the amount of Pod instances (Kubernetes 2021k). The first way to scale horizontally is to manually increase the number of Pods by changing the replicas field in a Deployment object for

example (Kubernetes 2021f). When Pods are scaled manually, the number of Pods is fixed.

Scaling Pods horizontally can be automated by using Horizontal Pod Autoscaler (HPA). This is done by specifying a condition for when the Pods should be scaled. The condition for when Pods should be scaled can for example be decided by how much CPU or RAM the current Pods are using. In addition, it is possible to define even more customized conditions for when to scale Pods. (Kubernetes 2021k).

In figure 8, the HPA concept is shown. It can be implemented by creating a HorizontalPodAutoscaler object. This HPA object is linked with a Deployment object through which it is able to scale the number of Pods. The HPA scales the number of Pods by editing the replicas field in the Deployment object. Since the Deployment object in Kubernetes has a controller that makes sure the actual number of Pods is equal to the desired state, the changes will automatically take effect. (Kubernetes 2021k).

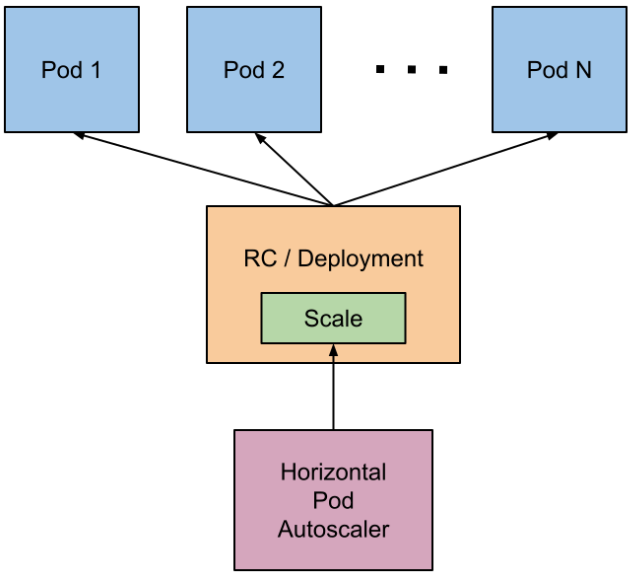


Figure 8. Horizontal Pod Autoscaler (Kubernetes 2021k). License: CC BY 4.0.

Kubernetes is able to automate horizontal scaling through a controller. This controller has a time interval for checking if Pods should be scaled. Each time the controller runs,

it compares the desired metrics in the HPA object with the actual metrics at that moment. If the condition defined in the HPA object is based on CPU or RAM usage, it retrieves the Pod metrics from Resource Metrics API. However, if the condition for scaling is a custom one, the metrics are retrieved from Custom Metrics API. (Kubernetes 2021k).

If there is not enough Pods running to meet the desired state defined in an HPA object, Kubernetes will increase the number of Pods. However, if there are more Pods then needed running, Kubernetes will decrease the number of Pods to a degree where the desired state is still fulfilled. This is achieved by continuously calculating an optimal amount of Pod replicas to meet the desired state. (Kubernetes 2021k).

4.3 Vertical pod scaling

Vertical scaling is to increase resources on a compute instance (Techopedia 2021b). This can for example mean to increase a server's RAM or CPU (Techopedia 2021b). According to Section (2020), the advantage of scaling vertically is that it is simpler than horizontal scaling in terms of not having to think about how to connect multiple compute instances. However, the disadvantage of scaling a compute instance vertically is that it often requires downtime (Section 2020).

In Kubernetes, Pods can be scaled vertically by specifying how much resources the containers in a Pod can use. Usually this is done by configuring CPU and RAM for the containers. How much resources is needed to run a container can be specified by setting a resource *request*. Kubernetes scheduler selects a node to deploy the Pod to using this information. The maximum amount of resources a container can use can be specified by setting resource *limit*. (Kubernetes 2021l).

Below is an example by Kubernetes (2021l) on how resources for containers in a Pod can be managed. This Pod runs two different containers with their own resource requests and limits. In this case, both containers have the same amount of resources. They require

a minimum of 64 MiB of RAM and 250m CPU, which is specified in the *requests* field. The containers can't use more than 128MiB of RAM and 500m CPU, specified in the *limits* field. The "m" unit for CPU stands for *millicpu*, where 1000m is equivalent to 1 CPU core (Kubernetes 2021l). Pods can be scaled vertically by changing container resources as in this example.

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

The second method for scaling Pods vertically is to use Vertical Pod Autoscaler (VPA). It can be used to automate the management of Pod resources. VPA can automatically scale the Pods vertically by changing RAM and CPU requests or limits for the containers in a Pod. It can automatically find optimal resources for Pods when the load changes. If a Pods has too little resources, VPA can automatically add more resources. If the Pod has too much resources, VPA automatically reduces Pod resources. (Github 2021a).

Below is an example by (Github 2021a) on how to create a VPA manifest. In this example the VPA object is created to control the containers created by a Deployment object. In this example the VPA runs in an "Auto" *update mode*. This mode allows the VPA object to automatically change pod resources at any time of a Pod's lifecycle. If the mode is set

to “Initial”, VPA is only allowed to change resources when a new Pod is initialized. Finally, if the mode is set to “Off”, VPA can’t change Pod resources. Instead, the “Off” mode can only provide the information about what the optimal resources would be for the Pod. (Github 2021a).

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: my-app-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment
    name:       my-app
  updatePolicy:
    updateMode: "Auto"
```

4.4 Cluster Autoscaler

Kubernetes has a Cluster Autoscaler tool designed to automatically scale a cluster when resource requirements change. The Cluster Autoscaler automatically adds new server nodes if Kubernetes fails to find a node with enough resources to run new Pods. On the contrary, Cluster Autoscaler removes unnecessary nodes when there are more node resources than needed to run the current workload. (Github 2021b).

In figure 9, the first gray box shows a scenario where Cluster Autoscaler scales up. The cluster initially consists of three nodes, with four Pods running in the first node. The remaining two nodes have three Pods each running. On the left-hand side in this scenario, there are three scheduled Pods ready to be deployed to the cluster. However, since there is not enough resources to run these Pods on the current nodes, ClusterAutoscaler deployed one of the Pods to an existing node, added a new node to the cluster, and finally deployed the two remaining Pods to the new node. When new Pods have been scheduled for Kubernetes to run and there are not enough resources on any node to run them, Cluster Autoscaler automatically adds new nodes in order to run the new Pods (Google Cloud 2020).

Figure 9. Cluster Autoscaler (Google Cloud 2020). License: CC BY 4.0.

In figure 9, the scaling down case for Cluster Autoscaler is demonstrated in the second gray box. In this case, the initial cluster consists of four nodes. The first node has four Pods running, the second node has three Pods, third node has one, and the fourth has two Pods running. The Cluster Autoscaler decreases the number of nodes if the nodes have enough unused resources (Google Cloud 2020). On the left-hand side, node 3 has a lot of unused resources since only one Pod is running. The Pod running in node 3 is able to fit inside node 4, which means that the Cluster Autoscaler can combine Pods in node 3 and 4 to the same node. In this example, the Pod on node 3 is moved to node 4 and the Cluster Autoscaler removed node 3 as shown on right-hand side. As a result, the cluster is able to run the same Pods using less resources automatically.

The Cluster Autoscaler can be used together with other autoscaling tools, for example HPA. When using HPA, the number of Pods scales automatically up or down based on

load. The Cluster Autoscaler can automatically add or remove nodes in the cluster based on the changing number of Pods controlled by HPA. (Github 2021b).

5 Test environment

In order to be able to run load tests, a test environment has to be built. This environment is a local Kubernetes cluster, meaning that the cluster is running on the same local PC used for load testing. This cluster is not a production ready cluster supposed to replicate a proper Kubernetes cluster running in the cloud. This local cluster is only used to demonstrate application scaling on Kubernetes.

5.1 Cluster architecture

The test environment used for scalability testing is a local Kubernetes cluster consisting of virtual machines. This cluster is built using Venkat Nagappan's Github project[1] that creates a local Kubernetes consisting of three nodes. The virtual machines are created using Vagrant, which is a tool for creating virtual machines.

In figure 10, the cluster architecture for the test environment is demonstrated. Starting from the bottom, the underlying infrastructure for the cluster is three virtual machines with Linux Ubuntu 20.04 OS installed on them. The first virtual machine is assigned to be the control plane, having 2 CPU cores and 2560MB of memory. The other two are computing machines, consisting of 1 CPU core and 2056MB memory each. On top of the infrastructure Kubernetes has been installed to join the nodes into one complete Kubernetes cluster. The highest level of the architecture diagram shows the Kubernetes resources used to run the example application on this cluster. The example application is deployed using the Deployment object that takes care of running the application inside Pods. The Service object load balances the traffic between the Pods and takes care of exposing the Pods. In order for outside traffic to able to interact with the application, an Ingress controller is installed and an Ingress rule is created to point the local domain

[1] Venkat Nagappan's Github project for building a local Kubernetes cluster: https://github.com/justmeandopensource/kubernetes/tree/master/vagrant-provisioning

“backend-go.fliq.test” to the Service that exposes the example application Pods. This local domain name can be used to interact with the example application, allowing the load test HTTP requests to reach the application under test.

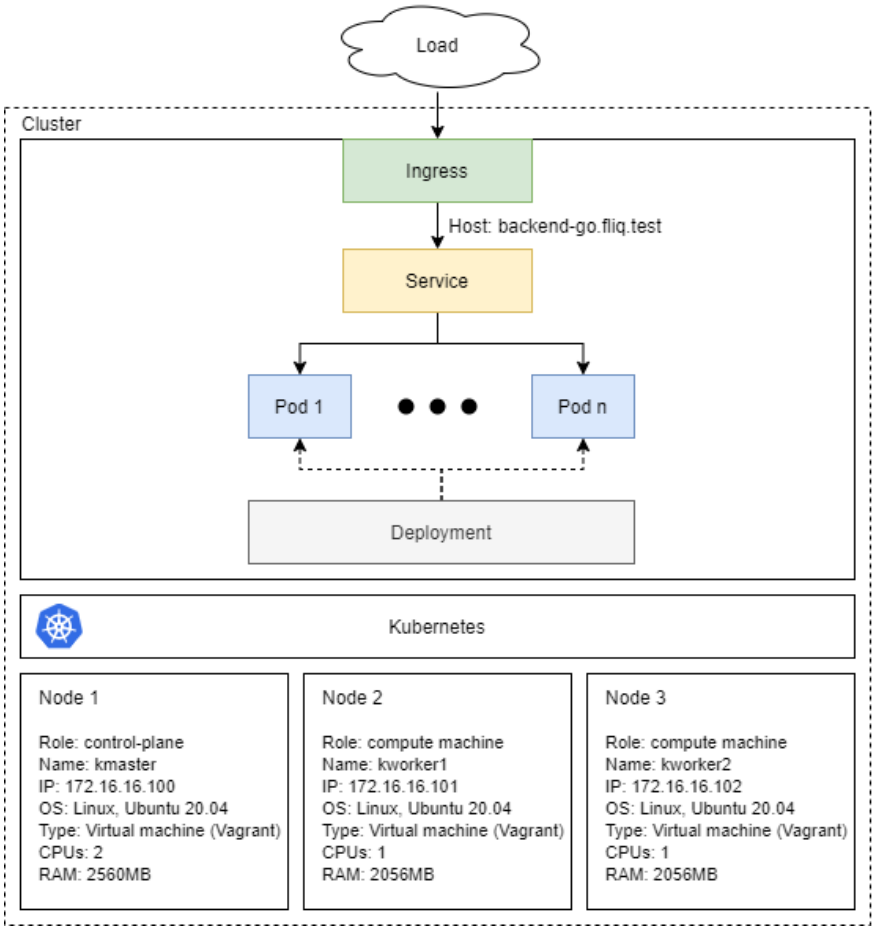


Figure 10. Cluster architecture diagram

5.2 Example application

In this chapter, the Kubernetes resources needed to run the example application are created. These are shown in figure 10, the highest level of the cluster architecture diagram. The needed resources are Deployment, Service, and Ingress.

The initial Deployment manifest is shown below. This Deployment object deploys one of Fliq's example REST API applications built using Go programming language. This Deployment deploys one Pod replica that runs the REST API in a container inside the Pod. The image field specifies that the container image should be pulled from the container registry where Fliq's example application is located. Since the REST API server is listening on port 8080, the same port on the container itself must be exposed. The container requests at least 100m of CPU and 128Mi of RAM. If there are enough resources in the cluster, the container is limited to use a maximum of 200m CPU and 256Mi RAM if needed.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fliq-backend-go
  labels:
    app: fliq-backend-go
    component: backend-go
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fliq-backend-go
      component: backend-go
  template:
    metadata:
      name: fliq-backend-go
      labels:
        app: fliq-backend-go
        component: backend-go
    spec:
      containers:
        - name: backend-go
          image: fliqreg.azurecr.io/backend-go/local
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
```

In order to expose the Pods created by the Deployment object, a Service is created as shown below. This Service is of type ClusterIP, meaning that its IP address is only reachable within the cluster. The Service itself is reachable on port 80. However, the target

port is set to the same number as the example application container port specified in the Deployment manifest, port 8080. In order for this Service to find the Pods running the example application, the same selector fields are used as in the Deployment object.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: fliq-backend-go
  labels:
    app: fliq-backend-go
    component: backend-go
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: fliq-backend-go
    component: backend-go
```

The final Kubernetes object that has to be created for the example application is the Ingress rule as shown below. This Ingress is used to allow traffic coming from outside the cluster to find the Service that exposes the example application. The correct service is detected by setting the name of the Service and its port number. This particular Ingress rule is applied to all HTTP requests for the local domain “backend-go.fliq.test”. Optional annotations have been set for the nginx Ingress controller installed in this cluster to increase default timeout.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: backend-go-ingress
  annotations:
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
spec:
  rules:
    - host: backend-go.fliq.test
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: fliq-backend-go
                port:
                  number: 80
```

6 Scalability testing

In this chapter, scalability testing is performed against one of Fliq's example REST API applications. First, the objective and scope are defined for scalability testing. The scalability testing is done by load testing the example application running in the test environment.

6.1 Objective

The goal of this scalability testing is to get a better understanding of how an application can be scaled on Kubernetes. Scalability is tested by performing load tests against an example application running in a Kubernetes cluster. Scaling methods are later applied to see if the application can be scaled when running in a Kubernetes cluster. New load tests are performed after scaling to see if the application is able to scale.

The first load tests are performed against the example application running in a single Pod. The application is later scaled vertically and horizontally to see if the scaling methods works. This scalability testing is limited in terms of only testing the scalability of a single application running in the cluster. The cluster nodes are not scaled since Cluster Autoscaler only works for specific cloud providers as of now (Github 2022). The test environment used in this thesis is a local Kubernetes cluster built using virtual machines.

6.2 Load testing

In this chapter load tests are performed against the example application deployed to the local Kubernetes cluster created as a test environment in the previous chapter. The tool selected to create and execute load tests is JMeter. For monitoring resource usage during load tests, Prometheus and Grafana is used. The first tests are executed against the initial

Deployment defined in chapter 5.2. New tests are run after scaling the application vertically and horizontally.

Since the example application is a REST API, load tests have to be executed against specific API resources, also known as API endpoints. In figure 11, the endpoint chosen for the load tests is shown. The HTTP request points to the local domain serving the example application deployed to the local Kubernetes cluster. The HTTP method is of type GET, and the selected endpoint is shown in the path input field.

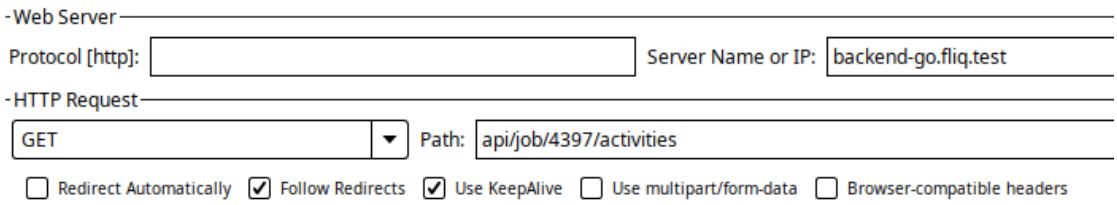


Figure 11. API endpoint used for load testing

For testing scalability, the load tests start by simulating 1000 users for the initial resource limits set for the Deployment in chapter 5.2.2. For each test the number of users is increased by 1000 until the API runs out of resources. After this initial test, the application is scaled both vertically and horizontally to see if it can handle more users. Between the initial, vertical and horizontal scaling tests, metrics given by JMeter are compared. The average response time is analyzed to compare the average time it takes for the server to send HTTP response. In addition, throughput is analyzed to compare the number of requests the server processes per second.

6.2.1 JMeter setup

In order to simulate a certain number of users sending requests to the API, a concurrency thread group is created in JMeter. Figure 12 shows the concurrency thread group used to simulate a certain number of users. In this particular example, target concurrency is set to 6000, meaning that 6000 threads or users are simulated. Ramp up time is 20 and

steps count 10, meaning that 600 new threads are created every 2 seconds up to 20 seconds. Hold target rate time is set to 10, meaning that when the number of threads reaches 6000, the load is held for 10 seconds. In the beginning, the target concurrency is set to 1000, and later increased by 1000 for each new test.

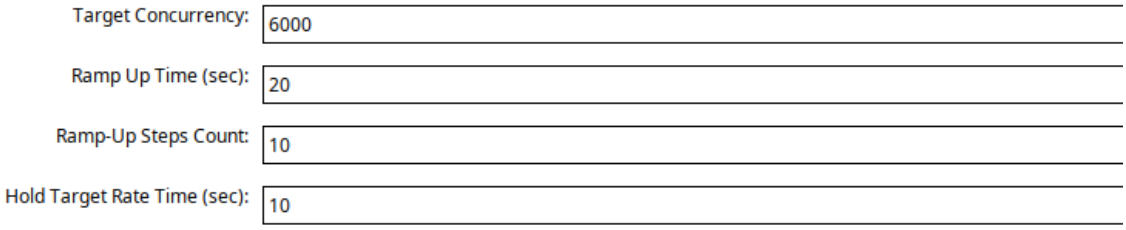


Figure 12. Concurrency thread group example

6.2.2 Initial test

This initial test is executed against the example application described in the Deployment manifest shown in chapter 5.2.2. The application requests 100m CPU and 128Mi of RAM. In addition, it has a limit of 200m CPU and 256Mi of RAM. Figure 13 shows all resources found that uses the same app selector with the help of kubectl CLI. The figure shows that the initial Deployment manifest created a deployment object. Behind the scenes, this Deployment object also created a ReplicaSet object that manages the number of Pod instances. In the Deployment manifest the replicas field was set to 1. The figure shows that 1 Pod instance is running as expected. The Service object created in chapter 5.2.2 is also shown in figure 13 since the kubectl command used to show the resource outputs searches for all Kubernetes objects with the same app selector “fliq-backend-go”.

```
$ kubectl get all --selector=app=fliq-backend-go
NAME                                  READY   STATUS    RESTARTS   AGE
pod/fliq-backend-go-5d8949f96f-94mpq   1/1     Running   0          44s

NAME                     TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/fliq-backend-go   ClusterIP   10.101.64.111   <none>        80/TCP    44s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/fliq-backend-go    1/1     1            1           44s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/fliq-backend-go-5d8949f96f    1         1         1       44s
```

Figure 13. Kubernetes resources used for initial test

Table 1 shows JMeter results for 1000-6000 users. For each new test, the users are increased by 1000 users. The average response time for the HTTP requests increases when the number of users increase. Throughput stays around 30 requests/s for all tests. However, for the final test of 6000 users 79,97% of the HTTP requests failed. These results show that the REST API endpoint fails at 6000 users for the initial resources set for the application.

Table 1. JMeter results for initial test

users	avg response time (ms)	error (%)	throughput (requests/s)
1000	18584	0	33,9
2000	38808	0	32,4
3000	63881	0	29,7
4000	81407	0	32,4
5000	106165	0	31,1
6000	101575	79,97	33,7

Figure 14 shows Pod details after the test of 6000 users with the help of “kubectl describe” command. This command shows the last state of the Pod was terminated. In addition, it shows the reason for the termination was “OOMKilled”, meaning that the Pod ran out of memory. This is the reason for why the majority of the HTTP requests failed for 6000 users. However, the Pod’s current state is running, and the restart count is 1, meaning that the Pod has automatically restarted after running out of memory.

```
$ kubectl describe pod/fliq-backend-go-5d8949f96f-x82js
Name:         fliq-backend-go-5d8949f96f-x82js
Namespace:    backend-go
Priority:     0
Node:         kworker2/172.16.16.102
Start Time:   Tue, 09 Nov 2021 01:08:23 +0200
Labels:       app=fliq-backend-go
              component=backend-go
              pod-template-hash=5d8949f96f
Annotations:  cni.projectcalico.org/podIP: 192.168.77.160/32
              cni.projectcalico.org/podIPs: 192.168.77.160/32
Status:       Running
IP:           192.168.77.160
IPs:
  IP:           192.168.77.160
Controlled By:  ReplicaSet/fliq-backend-go-5d8949f96f
Containers:
  backend-go:
    Container ID:   containerd://5e85ed05a6c4e3e36eba85e0cc0a862f695d2b9b3fa836a0f4a4d2b54f7bc309
    Image:          fliqreg.azurecr.io/backend-go/local
    Image ID:       fliqreg.azurecr.io/backend-go/local@sha256:0185f18198abca5160f2bff6f600e8fa211e948f
d97d2234de1db67562257786
    Port:           8080/TCP
    Host Port:      0/TCP
    State:          Running
      Started:      Tue, 09 Nov 2021 01:12:06 +0200
    Last State:     Terminated
      Reason:       OOMKilled
      Exit Code:    137
      Started:      Tue, 09 Nov 2021 01:08:25 +0200
      Finished:     Tue, 09 Nov 2021 01:12:04 +0200
    Ready:          True
    Restart Count:  1
```

Figure 14. Pod details after running out of memory

Figure 15 shows the max CPU usage for the initial test. The first test of 1000 users stays below the requested amount of 0,1 CPU cores. All tests between 2000-6000 users reach close to the CPU limit of 0,2 CPU cores.

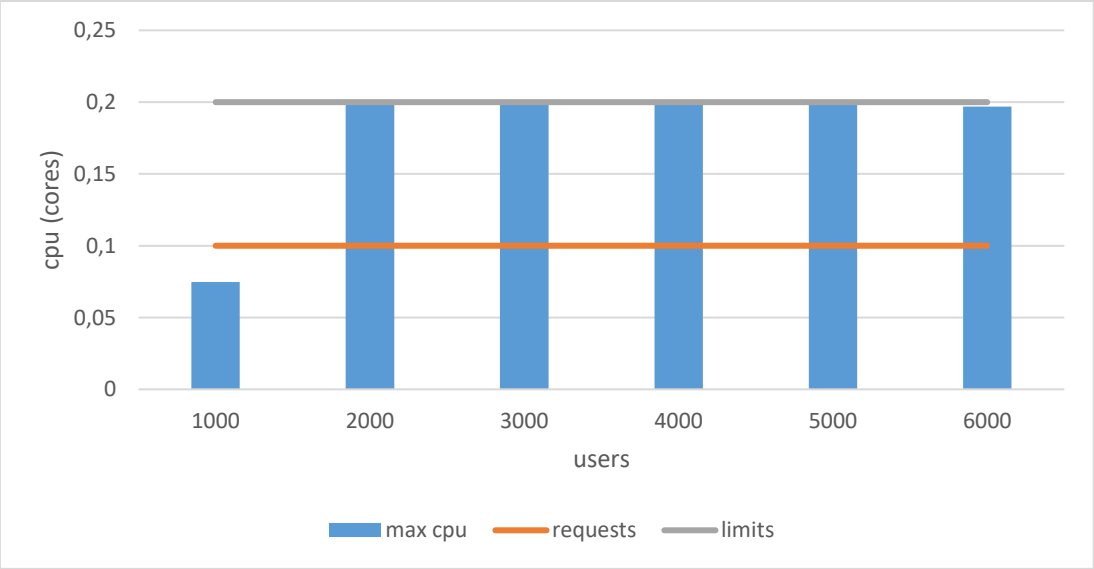


Figure 15. Max CPU usage for the initial load test

Figure 16 shows the max RAM usage for the initial test. Both 1000 and 2000 users stays below the requested amount of 128 MiB. When simulating 3000-5000, RAM usage stays between the requested and limited amount. The final test of 6000 users stays below the limited amount according to what is monitored in Grafana. However, figure 14 shows that the Pod ran out of memory after the 6000-user test.

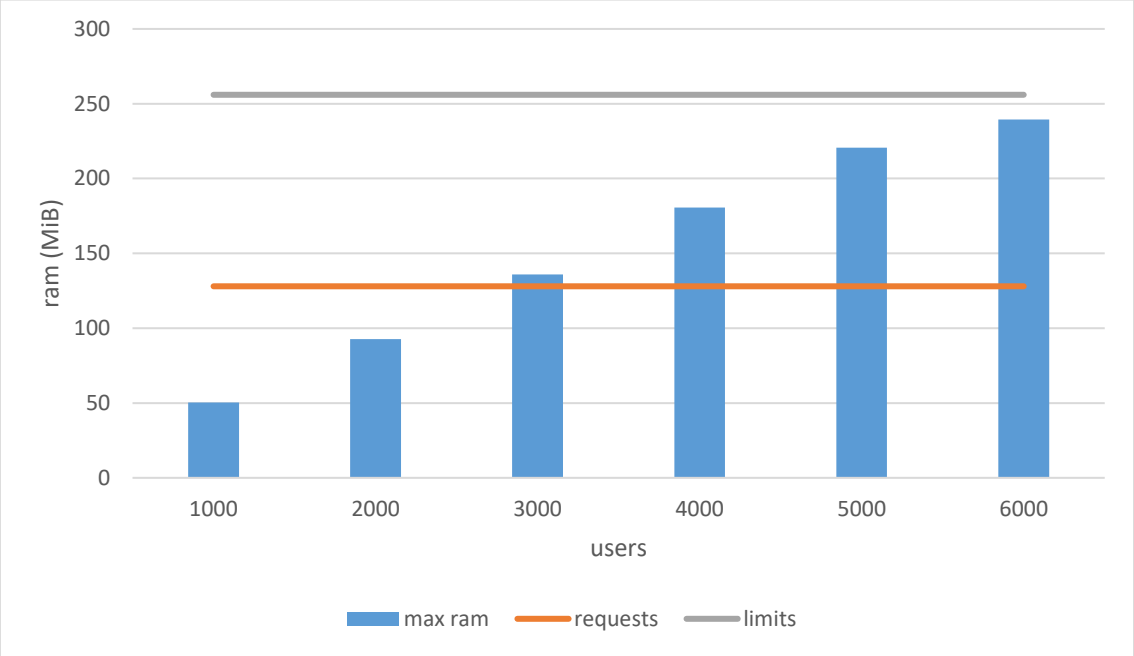


Figure 16. Max RAM usage initial test

6.2.3 Vertical scaling

For the next load test, the example application is scaled vertically. This means increasing the resources for the single instance running the example application. The YAML file below shows the Deployment manifest used for scaling vertically. This manifest is the same as for the initial test, except the resource requests and limits have changed. The requested CPU has increased from 100m to 200m, and the requested memory has increased from 128Mi to 256Mi. In addition, CPU limit has increased from 200m to 400m, and memory limit has increased from 256Mi to 512Mi.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fliq-backend-go
  labels:
    app: fliq-backend-go
    component: backend-go
spec:
  replicas: 1
  selector:
    matchLabels:
      app: fliq-backend-go
      component: backend-go
  template:
    metadata:
      name: fliq-backend-go
      labels:
        app: fliq-backend-go
        component: backend-go
    spec:
      containers:
        - name: backend-go
          image: fliqreg.azurecr.io/backend-go/local
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          resources:
            requests:
              cpu: 200m
              memory: 256Mi
            limits:
              cpu: 400m
              memory: 512Mi
```

Table 2 shows the JMeter results for the vertical scaling load tests. Similar to the initial test, the average response time inceases when the number of users increase. The througput is sligthly higher for the 1000-3000 user tests, and after it stays around 35 requests/s. For the vertical scaling tests there are test results for 1000-10000 users, since the REST API is able to scale beyond 6000 users compared to the initial test. The error percentage column shows that the REST API is able to handle 10000 users without any errors when the Pod is scaled vertically.

Table 2. JMeter results for vertical scaling test

users	avg response time (ms)	error (%)	throughput (requests/s)
1000	14913	0	44,3
2000	30436	0	41,7
3000	51885	0	37
4000	74671	0	35,6
5000	91835	0	36,6
6000	113648	0	34,9
7000	128833	0	37,1
8000	163461	0	34,4
9000	178587	0	35,1
10000	191182	0	36,4

Figure 17 shows max CPU usage between 1000-10000 users when the Pod has been scaled vertically. For the first test of 1000 users, CPU usage is below the requested amount 0,2 cores. For all tests between 2000-10000 users, CPU usage is between the requested amount (0,2 cores) and limited amount (0,4 cores). According to the metrics given by Grafan, the Pod's CPU did not reach its limit during the tests.

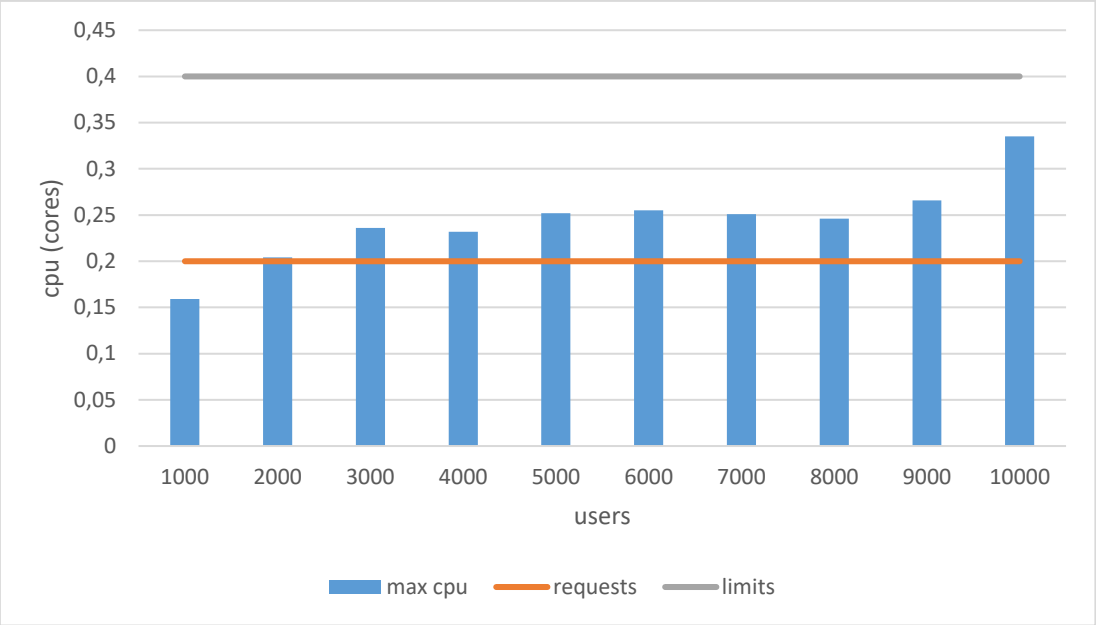


Figure 17. Max CPU usage vertical scaling test

Figure 18 shows max RAM usage between 1000-10000 users for the vertical scaling test. RAM usage stays below the requested amount 256MiB for 1000-5000 users. For 6000-10000 users, RAM usage stays between the requested (256MiB) and limited (512MiB).

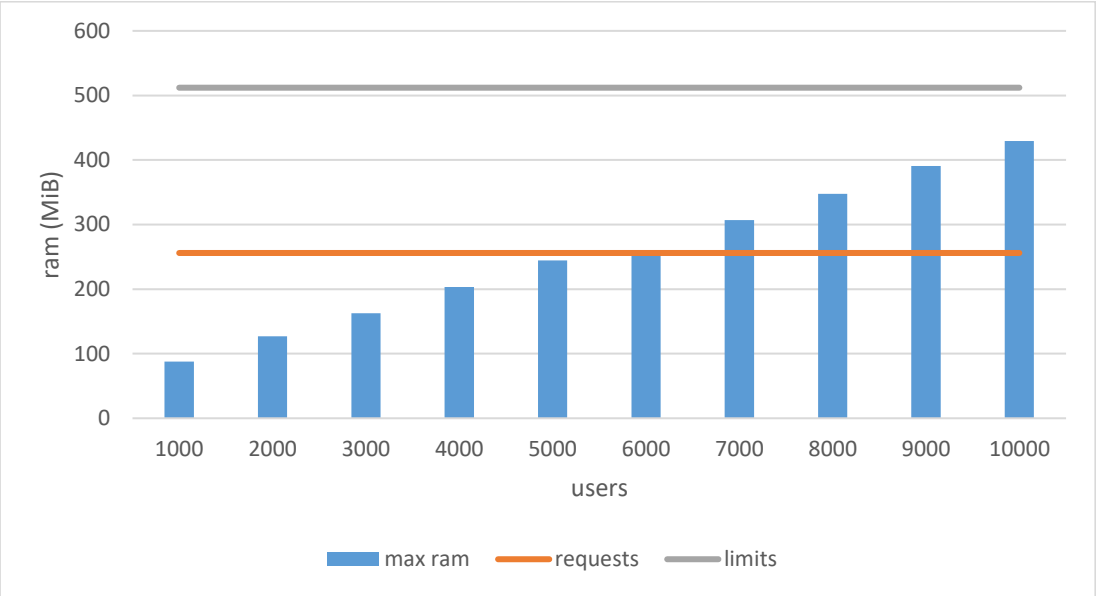


Figure 18. Max RAM usage vertical scaling test

6.2.4 Horizontal scaling

For the third and final load test, the example application is scaled horizontally instead of vertically. This means increasing the number of Pod instances running the example application. Below the Deployment manifest is shown for horizontal scaling. The difference between this manifest and the one used for the initial test is that the number of replicas has been increased from 1 to 2. This change increases the number of Pod instances to a total of two when the manifest file is applied. Each Pod has the same resources as the Pod used for the initial test. However, this time there are two Pods that can share the load since the Service load balances traffic between the Pods it exposes.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fliq-backend-go
  labels:
    app: fliq-backend-go
    component: backend-go
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fliq-backend-go
      component: backend-go
  template:
    metadata:
      name: fliq-backend-go
      labels:
        app: fliq-backend-go
        component: backend-go
    spec:
      containers:
        - name: backend-go
          image: fliqreg.azurecr.io/backend-go/local
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
```

Figure 19 shows that there are now two pod instances running the example application. The age column of the command output shows that there was only one Pod to begin

with since it has been running for 5 minutes and 56 seconds. The second Pod has only been running for 12 seconds. Kubernetes created 1 additional Pod as a result after the manifest was reapplied, after increasing the replicas field from 1 to 2. The desired state changed, and Kubernetes reacts by comparing the actual state with the desired state, and as a results notices that one additional Pod has to be created.

```
$ kubectl get all --selector=app=fliq-backend-go
NAME                                 READY   STATUS    RESTARTS   AGE
pod/fliq-backend-go-5d8949f96f-94mpq   1/1     Running   0          5m56s
pod/fliq-backend-go-5d8949f96f-chrmg   1/1     Running   0          12s
```

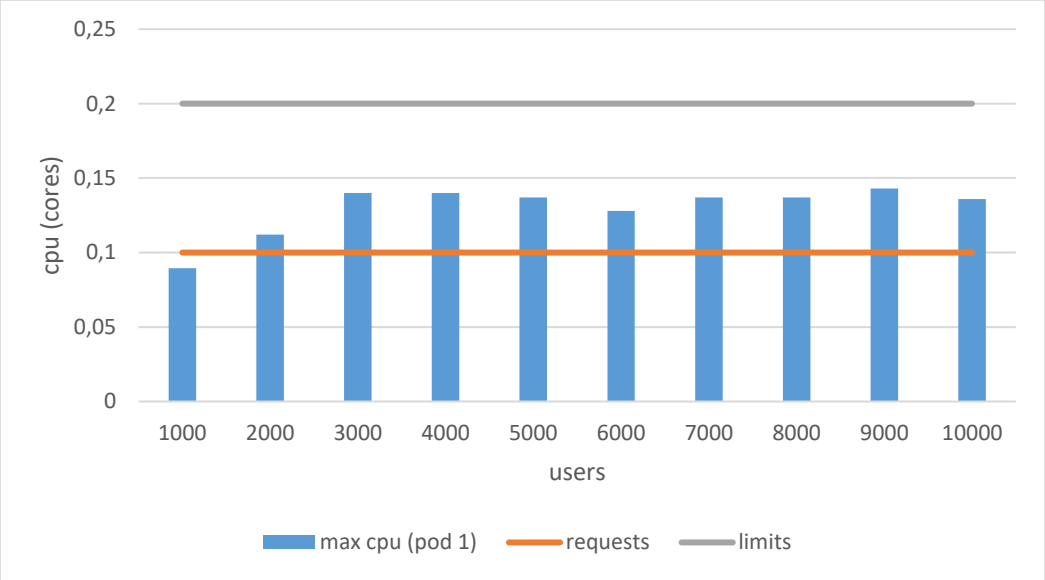
Figure 19. Pod instances for horizontal scaling

Table 3 shows JMeter results for the horizontal scaling test between 1000-10000 users. The average response time increases as the number of users increases. This time the REST API is able to scale beyond 6000 users as well, since no HTTP requests failed during any test. Throughput is around 50 requests/s for all tests, except for the last test of 10000 users decreased it down to 43 requests/s.

Table 3. JMeter results for horizontal scaling test

users	avg response time (ms)	error (%)	throughput (requests/s)
1000	11426	0	53,9
2000	22656	0	54,6
3000	32495	0	55,3
4000	42246	0	55,2
5000	49641	0	48,6
6000	71618	0	49,5
7000	90920	0	47,4
8000	102083	0	49,5
9000	120924	0	46,8
10000	151675	0	43

Figure 20 shows the max CPU usage for the horizontal scaling test. The cluster has two Pods running the REST API application, and the Service in front of the Pods load balances the traffic. This phenomenon is shown both figures 21 and 22, both Pods are sharing the load. The results from Grafana that the CPU usage is for the most part evenly distributed between the Pods. For the 1000-user test, Pod 1 used less CPU than the requested amount (0,1 cores), and Pod 2 max CPU reaches the requested amount. For 2000-10000 tests, the max CPU usage is between the requested (0,1 cores) and limited (0,2 cores).



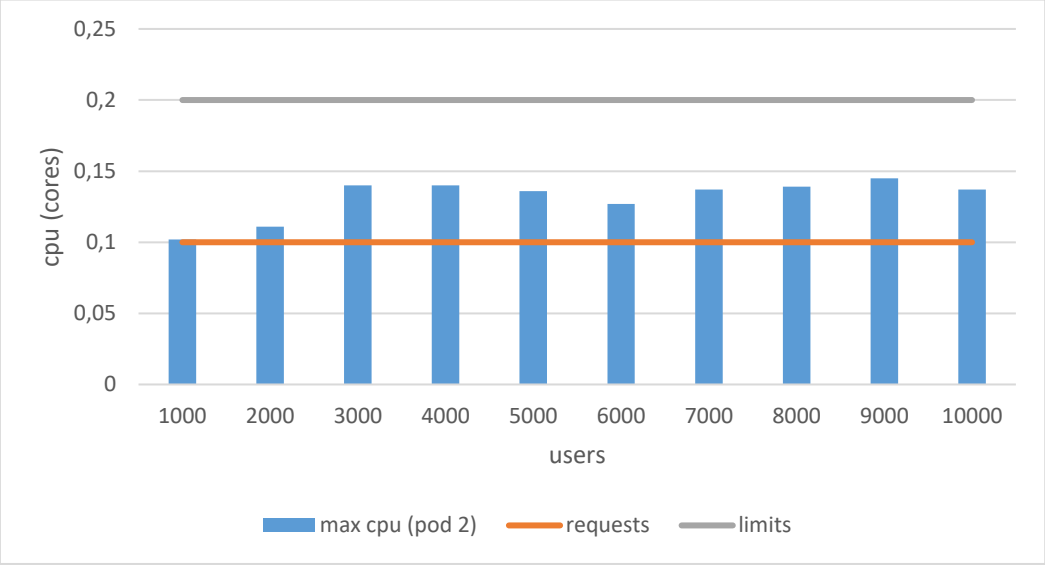


Figure 20. Max CPU usage horizontal scaling test

Figure 21 shows max RAM usage for the horizontal scaling test between 1000-10000 users. Similar to CPU usage, RAM usage is evenly distributed between the Pods. For 1000-5000 user tests, max RAM usage is lower than the requested amount 128MiB. For 6000 users and more, max RAM usage is between the requested (128MiB) and limited (256MiB).

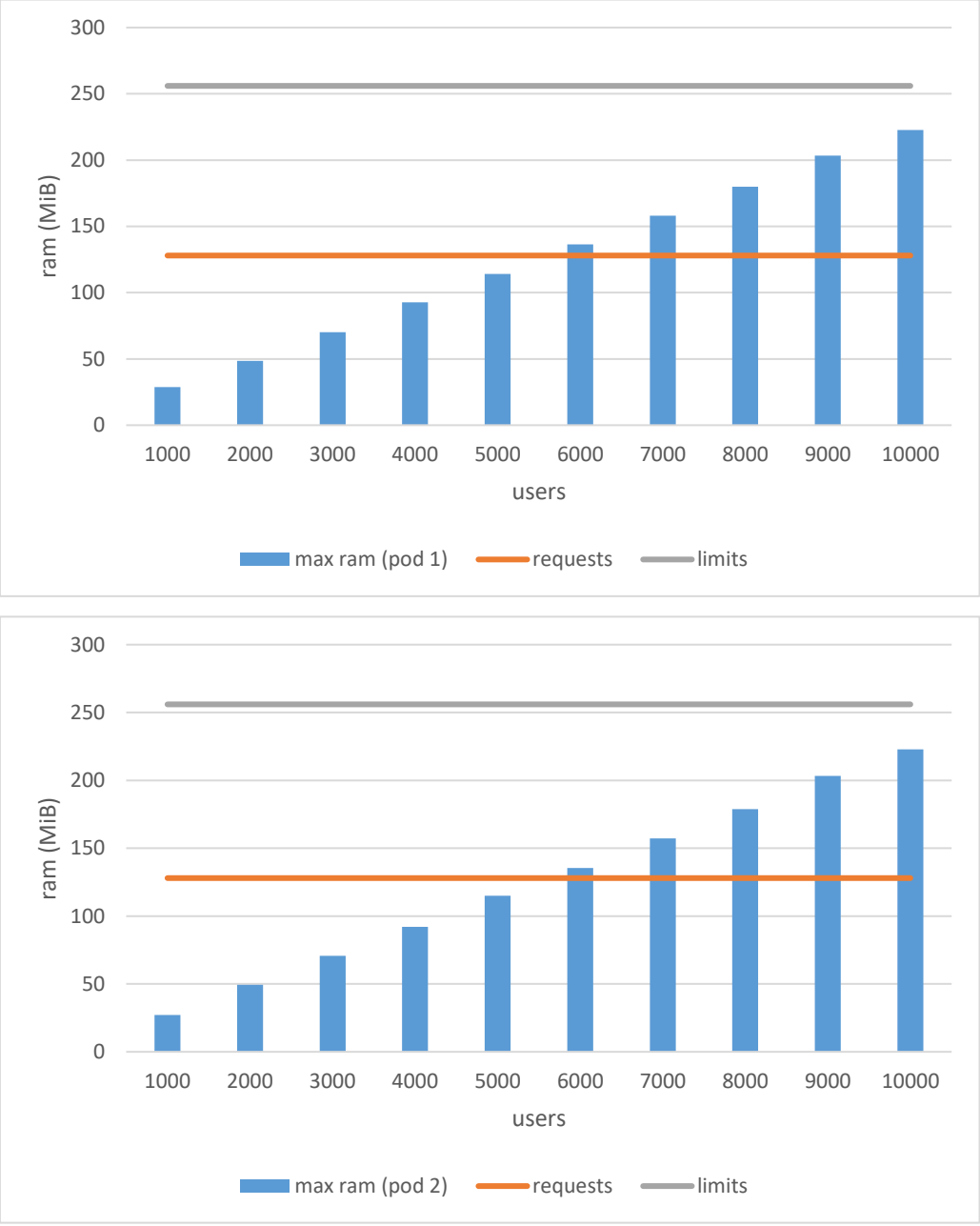


Figure 21. Max RAM usage horizontal scaling test

6.3 Evaluation

The initial load testing results show that the REST API endpoint under test failed to scale beyond 5000 users. This is further shown in figure 22, where the error percentage retrieved from JMeter results is compared between the initial, vertical, and horizontal scaling tests. The Pod ran out of memory during the 6000-user test for the initial Deployment configuration. Both the vertical and horizontal scaling test results show that the REST API was able to scale beyond 6000 users, even up to 10000 users without HTTP requests failing. This shows that both vertical and horizontal scaling can be used to provide enough resources for an application running on Kubernetes.

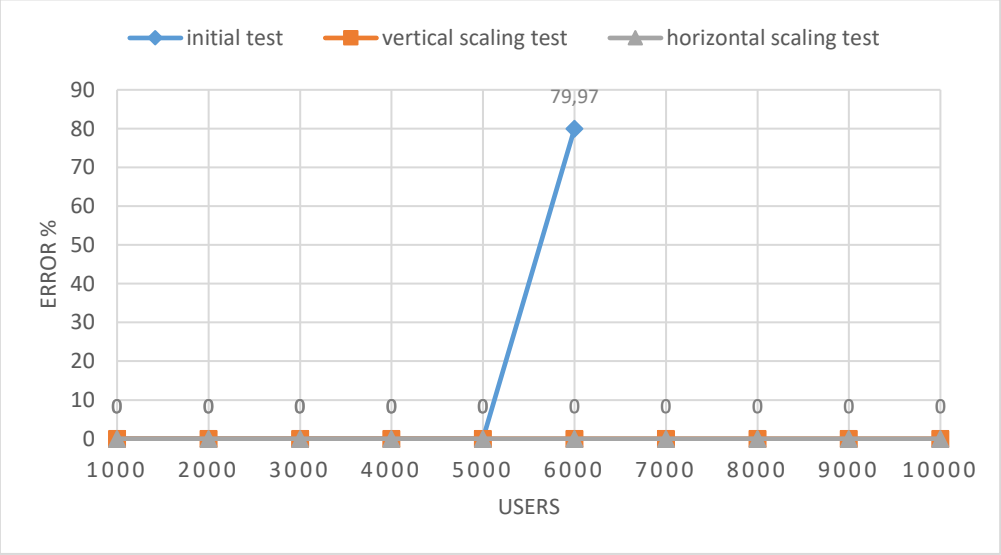


Figure 22. HTTP error % for initial, vertical, and horizontal scaling tests

The initial test caused the Pod to run out of memory during the 6000-user test. This caused the Pod to terminate, and Kubernetes automatically restarted the Pod. The reason for Kubernetes doing this, is that it noticed that the actual state is different compared to the desired state, which is that one REST API Pod replica should always be running. This aligns with Kubernetes (2021a) describing Kubernetes as self-healing.

In figure 23, average response time is compared between the initial, vertical, and horizontal scaling tests. Initial and vertical scaling tests show similar linear growth for the average response time as the number of users is increased by 1000. Horizontal scaling test results show that the average response time decreased when the number of Pods increased from 1 to 2. This aligns with S. Jain & A. K. Saxena (2016) statement about response time decreasing when HTTP traffic is load balanced. The average response is not only lower when the REST API is scaled horizontally, it also grows at a lower rate for 1000-5000 user tests.

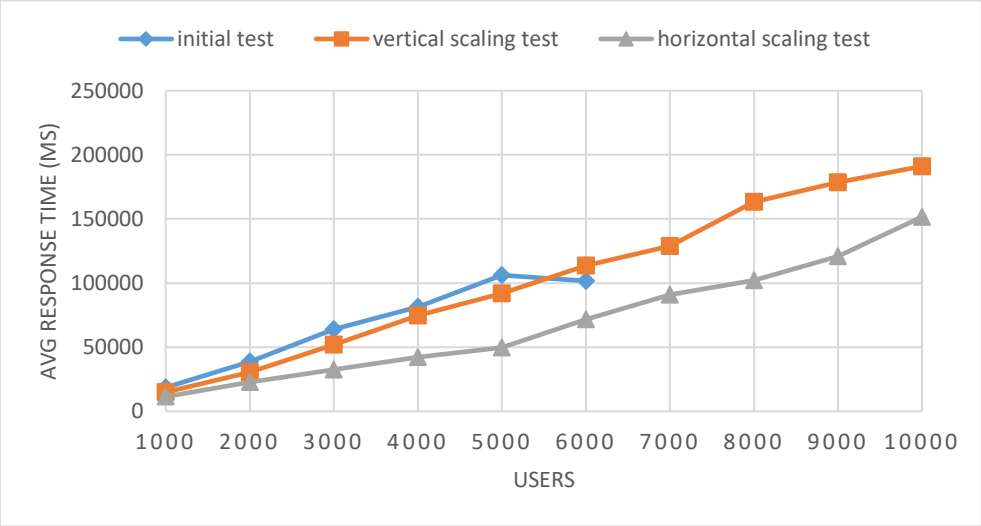


Figure 23. Average response time for initial, vertical, and horizontal scaling tests

In figure 24, throughput is compared between initial, vertical, and horizontal scaling tests. For the initial tests, throughput was consistently around 30 requests/s for 1000-6000 users. When scaling vertically, throughput started higher from 44,3 requests/s, and slowly decreased to towards about 35 requests/s and stayed there between 4000-10000 users. Horizontal scaling test resulted in higher throughput compared to the initial and vertical scaling test, for all 1000-10000 user tests. This aligns with D. Sharma (2018) saying that horizontal scaling can result in higher throughput.

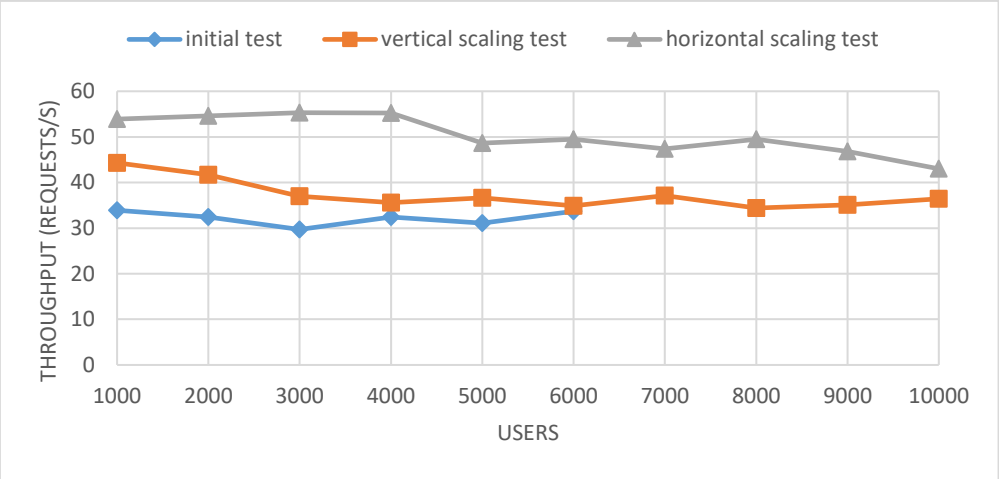


Figure 24. Throughput for initial, vertical, and horizontal scaling tests

In general, the load testing results show that applications running on Kubernetes can be scaled both vertically and horizontally. Both scaling methods solved the problem where the REST API ran out of memory when the number of users reached 6000. The resource usage metrics from Grafana also showed that the REST API needed more memory in order scale to 6000 users and beyond. In addition, Grafana metrics show that Kubernetes is able to scale horizontally by evenly distributing traffic between Pod instances.

7 Conclusion

This chapter is the thesis conclusion. First, the findings about how a Kubernetes cluster can be scaled is discussed, based on the researched scaling methods and load testing results. In addition, the limitations of this research are discussed. Finally, proposals are made for future research.

7.1 Scaling a Kubernetes cluster

The main goal of this thesis was to research how a Kubernetes cluster can be scaled with containerized applications running on it. This was done by first researching how applications can be scaled when deployed to Kubernetes. Next step was to research how the cluster itself can be scaled through Cluster Autoscaler. To get even further insight in Kubernetes scalability, a REST API deployed to a local Kubernetes cluster was load tested using JMeter.

Containerized applications running on Kubernetes can be scaled both vertically and horizontally. Vertical scaling can be achieved by increasing CPU or RAM for each container inside a Pod, either manually by changing requested and limited resource specifications, or automatically using Vertical Pod Autoscaler (Kubernetes 2021l). Horizontal scaling can be achieved by running more than one instance of a Pod, either manually by changing the Deployment replicas property, or automatically using Horizontal Pod Autoscaler (Kubernetes 2021k).

The Kubernetes cluster itself can automatically be scaled using Cluster Autoscaler. Kubernetes is able to automatically add or remove servers in the cluster depending on how much resources the Pods are using (Github 2021b). Cluster Autoscaler can be used together with HPA, where the change in number of Pods automatically change the number of nodes needed to provide enough resources for the current workload (Github 2021b).

This combination makes it possible to automatically scale both the application and the underlying infrastructure on demand.

In order to get a better understanding of scaling applications running on Kubernetes, a REST API was deployed to a local Kubernetes cluster. This cluster consisted of three Vagrant virtual machines, where one node is set as control plane and the other two as worker nodes. JMeter was used to load test one REST API endpoint. Load tests started from 1000 users, and after each test the number of users was increased by 1000 to a maximum 10000 users if the application could handle it. The same tests were run after scaling the application both vertically and horizontally in order to see how they affect results.

For the initial Deployment configurations, the REST API endpoint was only able to handle 5000 users. When simulating 6000 users, 79.97% of the HTTP requests failed. The reason for this was that the REST API ran out of memory. Kubernetes automatically restarted the Pod after termination since Kubernetes always compares the desired state with the actual state (Kubernetes 2021a). This aligns with Kubernetes (2021a) describing Kubernetes as a self-healing platform. The Grafana metrics showed that max CPU usage reached its limit of 0,2 cores during the 2000-6000 user tests. In addition, the Grafana results showed that memory usage reached closer the limit of 256MiB for each test.

The REST API Pod was scaled vertically, by increasing CPU and RAM. As a result, the application was able to scale up to 10000 users. However, scaling vertically only solved the REST API running out of memory. The average response time was similar to the initial test results. Scaling vertically increased the throughput during 1000-5000 user tests. However, between 6000-10000 users the throughput was the same as for the initial test.

Horizontal scaling was applied on the application by adding one more Pod instance. Hence, the Service exposing the Pods was able to load balance the traffic between the Pods. The Grafana metrics show that the load was distributed. JMeter results showed

that horizontal scaling decreased the average response time for all 1000-10000 user tests. In addition, it decreased the average response time growth rate. Finally, horizontal scaling results showed an increase in throughput for all tests.

Scaling horizontally is preferred for Fliq's example REST API. Vertical scaling was not able to bring more significant benefits than increasing the number of supported users. On the contrary, horizontal scaling was able to increase the number of supported users, decrease average response time, and increase throughput. In addition, horizontal scaling is able to serve clients even if one instance terminates.

7.2 Limitations and future research

The purpose of this thesis was to get a better understanding of how a Kubernetes cluster can be scaled. Kubernetes is complex container orchestration platform, that abstracts away the underlying infrastructure and its own internal components. Kubernetes can be deployed anywhere from bare metal to different cloud providers. Each cluster can have a unique setup, and different types of workloads running on it. The local Kubernetes cluster used for load testing in this thesis was not a production grade cluster. For future research, a cloud provider's production grade Kubernetes service could be used for comparison, for example Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (EKS) or Google Kubernetes Engine (GKE). A cloud provider's Kubernetes service would also allow one to test the Cluster Autoscaler feature of Kubernetes.

The JMeter load testing client and the local Kubernetes cluster used in this thesis were running on the same laptop. It would have been preferable to have them running in completely different environments. The consumers or clients of a REST API usually have separate devices and are located somewhere else physically. For future research, latency caused by users' location could be considered when load testing a Kubernetes cluster.

The purpose of the load tests in this thesis were to get a better understanding of how applications running on Kubernetes can be scaled. For this reason, the same REST API endpoint was used to test and compare the researched scalability methods. For future research, different types of applications, HTTP methods or REST API endpoints could be compared.

For future research, the impact of Kubernetes cluster's Ingress controller on scalability could be researched. The traffic coming into a cluster usually goes via the Ingress controller. Future research could investigate if the Ingress controller can be scaled, and how it affects applications scalability. In addition, Service NodePort or LoadBalancer could be compared as alternatives to using an Ingress controller as the cluster gateway.

References

A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, & J. Wilkes (2015). Large-scale cluster management at Google with Borg. *Proceedings of the Tenth European Conference on Computer Systems* (*EuroSys '15*). Association for Computing Machinery, New York, NY, USA, Article 18, 1–17. https://doi.org/10.1145/2741948.2741964

BlazeMeter (2016). *Advanced Load Testing Scenarios with JMeter Part 4 – Stepping Thread Group and Concurrency Thread Group*. Retrieved 7.12.2021 from https://www.blazemeter.com/blog/advanced-load-testing-scenarios-jmeter-part-4-stepping-thread-group-and-concurrency-thread

BlazeMeter (2019). *Performance Testing cs. Load Testing vs. Stress Testing*. Retrieved 7.12.2021 from https://www.blazemeter.com/blog/performance-testing-vs-load-testing-vs-stress-testing

Brazil (2018). *Prometheus Up & Running*. O'Reilly Media. ISBN 978-1292034148

CNCF (2018). *CNCF Cloud Native Definition v1.0*. Retrieved 6.1.2021 from https://github.com/cncf/toc/blob/master/DEFINITION.md

CNCF (2021a). *Home Page*. Retrieved 6.1.2021 from https://www.cncf.io

CNCF (2021b). *Graduated and Incubating Projects*. Retrieved 6.1.2021 from https://www.cncf.io/projects/

Docker (2021a). *What is a Container?* Retrieved 18.4.2021 from https://www.docker.com/resources/what-container

Docker (2021b). *Best practices for writing Dockerfiles*. Retrieved 6.9.2021 from https://docs.docker.com/develop/develop-images/dockerfile_best-practices

D2iQ (2018). *Brief History of Containers*. Retrieved 18.4.2021 from https://d2iq.com/blog/brief-history-containers

D. Merron & T. Idowu (2020). *Introduction to Kubernetes Helm Charts*. Retrieved 17.10.2021 from https://www.bmc.com/blogs/kubernetes-helm-charts/

D. Sharma (2018). Response Time Based Balancing of Load in Web Server Clusters. *7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 471-476, doi: 10.1109/ICRITO.2018.8748373.

Erinle (2013). *Performance Testing With JMeter 2.9*. Packt Publishing, Limited.

Github (2020). *Components of Kubernetes*. Retrieved 9.2.2022 from https://github.com/kubernetes/website/blob/main/static/images/docs/components-of-kubernetes.png

Github (2021a). *Vertical Pod Autoscaler*. Retrieved 1.11.2021 from https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler

Github (2021b). *Frequently Asked Questions*. Retrieved 2.11.2021 from https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md

Github (2022). *Cluster Autoscaler*. Retrieved 28.2.2022 from https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/README.md

Google Cloud (2022). *What are containers used for?* Retrieved 10.2.2022 from https://cloud.google.com/learn/what-are-containers

Google Cloud (2020). *Best practices for running cost-optimized Kubernetes applications on GKE*. Retrieved 2.11.2021 from https://cloud.google.com/architecture/best-practices-for-running-cost-effective-kubernetes-applications-on-gke

IBM (2020). *What is an Application Programming Interface (API)*. Retrieved 19.8.2020 from https://www.ibm.com/cloud/learn/api

Kanjilal (2013). *ASP.NET Web API: build RESTful web applications and services on the .NET framework*. Packt Publishing, Limited. https://ebookcentral-proquest-com.proxy.uwasa.fi/lib/tritonia-ebooks/reader.action?docID=1532007

Kubernetes (2015). *Borg: The Predecessor to Kubernetes*. Retrieved 2.10.2021 from https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/

Kubernetes (2020a). *Don't Panic: Kubernetes and Docker.* Retrieved 17.4.2021 from https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/

Kubernetes (2021a). *What is Kubernetes?* Retrieved 19.9.2021 from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Kubernetes (2021b). *Understanding Kubernetes Objects.* Retrieved 20.9.2021 from https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

Kubernetes (2021c). *Kubernetes Object Management*. Retrieved 2.10.2021 from https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/

Kubernetes (2021d). *Pods.* Retrieved 20.9.2021 from https://kubernetes.io/docs/concepts/workloads/pods/

Kubernetes (2021e). *Create Static Pods*. Retrieved 2.10.2021 from https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/

Kubernetes (2021f). *Deployments*. Retrieved 2.10.2021 from https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Kubernetes (2021g). *Service*. Retrieved 2.10.2021 from https://kubernetes.io/docs/concepts/services-networking/service/

Kubernetes (2021h). *Ingress*. Retrieved 10.10.2021 from https://kubernetes.io/docs/concepts/services-networking/ingress/

Kubernetes (2021i). *Overview of kubectl*. Retrieved 11.10.2021 from https://kubernetes.io/docs/reference/kubectl/overview/

Kubernetes (2021j). *Organizing Cluster Access Using kubeconfig Files*. Retrieved 11.10.2021 from https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/

Kubernetes (2021k). *Horizontal Pod Autoscaler*. Retrieved 25.10.2021 from https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

Kubernetes (2021l). *Managing Resources for Containers*. Retrieved 28.10 from https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

Kubernetes (2021m). *Components*. Retrieved 9.2.2022 from https://kubernetes.io/docs/concepts/overview/components/

Newman S. (2015). *Building Microservices* (1st ed.). O'Reilly Media.

Marinescu (2013). *Cloud computing : Theory and practice*. Elsevier Science & Technology.

Microsoft (2021). *Docker terminology*. Retrieved 10.2.2022 from https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-terminology

MuleSoft (2020). *Microservices vs Monolithic Architecture*. Retrieved 7.1.2021 from https://www.mulesoft.com/resources/api/microservices-vs-monolithic

Oracle (2021). *Deploying the Kubernetes Metrics Server on a Cluster Using Kubectl*. Retrieved 13.10.2021 from https://docs.oracle.com/en-us/iaas/Content/ContEng/Tasks/contengdeployingmetricsserver.htm

Poulton (2020). *Docker Deep Dive*. Leanpub

Puder, Römer, Pilhofer, & Romer (2005). *Distributed systems architecture : A middleware approach*. Elsevier Science & Technology.

R. Muddinagiri, S. Ambavane & S. Bayas (2019). Self-Hosted Kubernetes: Deploying Docker Containers Locally With Minikube. *International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)*, 239-243, doi: 10.1109/ICI-TAET47105.2019.9170208.

Section (2020). *Scaling Horizontally vs. Scaling Vertically*. Retrieved 28.10.2021 from https://www.section.io/blog/scaling-horizontally-vs-vertically/

Shivang (2019). *What is Grafana? Why Use It? Everything You Should Know About It*. Retrieved 17.10.2021 from https://www.scaleyourapp.com/what-is-grafana-why-use-it-everything-you-should-know-about-it/

S. Jain & A. K. Saxena (2016). *A survey of load balancing challenges in cloud environment*. International Conference System Modeling & Advancement in Research Trends (SMART), pp. 291-293, doi: 10.1109/SYSMART.2016.7894537.

Techopedia (2021a). *Horizontal Scaling*. Retrieved 19.10.2021 from https://www.techopedia.com/definition/7594/horizontal-scaling?ref=wellarchitected

Techopedia (2021b). *Vertical Scaling*. Retrieved 19.10.2021 from https://www.techopedia.com/definition/9912/vertical-scaling

Wang, Ranjan, Chen, & Benatallah (2011). *Cloud computing : Methodology, systems, and applications*. Taylor & Francis Group.