



Politechnika Łódzka

Instytut Informatyki

PRACA DYPLOMOWA MAGISTERSKA

**Tytuł angielski: KUBERNETES CLUSTER DEPLOYMENT FOR
PRODUCTION ENVIRONMENT**

**Tytuł polski: WDRAŻANIE KLASTRA KUBERNETES DLA
ŚRODOWISKA PRODUKCYJNEGO**

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej

Promotor: dr hab. inż. Aneta Poniszewska-Marańda

Dyplomant: Ewa Czechowska

Nr albumu: 227484

Kierunek: Computer Science and Information Technology

Łódź, 21.06.2020



Instytut Informatyki

90-924 Łódź, ul. Wólczańska 215, budynek B9

tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Contents

1	Introduction	7
1.1	Topic and problem description	7
1.2	Aim and scope of this study	8
1.3	Structure of this thesis	8
2	From microservices to automated orchestration	10
2.1	Microservices, DevOps, Continuous Delivery and Infrastructure as Code .	10
2.1.1	Microservices	10
2.1.2	DevOps	11
2.1.3	Continuous Integration and Continuous Delivery	12
2.1.4	Infrastructure as Code	13
2.2	AWS - The Amazon Cloud	14
2.3	Docker containers	16
2.4	Kubernetes as a Docker containers orchestration system	18
2.5	Kubernetes architecture	20
2.5.1	Kubernetes cluster	20
2.5.2	Masters and nodes	21
2.5.3	Components	22
2.5.4	Other services	24
2.5.5	The Kubernetes networking model	24
2.6	Production deployment requirements	25
2.6.1	Multiple environments in software deployment	25
2.6.2	Production deployment requirements	25
2.6.3	Monitoring as production environment requirement	26
2.6.4	Logging as production environment requirement	28
2.6.5	High Availability as production environment requirement	29
2.6.6	Automation as production environment requirements	30
2.6.7	Security as production environment requirement	32
2.6.8	Disaster recovery as production environment requirement	33
2.6.9	Testing as production environment requirement	35

2.7	Summary	35
3	Available Kubernetes cluster deployment methods	36
3.1	Managed Services	37
3.2	Command-line tools designed exactly to deploy a Kubernetes cluster . . .	39
3.3	Custom solutions	40
3.4	Deployment method: on AWS EKS Managed Service using <i>eksctl</i>	42
3.5	Deployment method: on AWS using <i>kops</i>	45
3.6	Amazon services allowing to run containers	46
3.7	Summary	47
4	Preparations for production deployment of Kubernetes cluster	49
4.1	AWS Region	49
4.2	Version of Kubernetes	51
4.2.1	Why it is important to choose a particular version?	51
4.2.2	Which version was chosen?	52
4.3	Tools and development environment	53
4.4	Selected requirements of a production deployment and acceptance cri-	
	teria needed to satisfy them	54
4.4.1	Healthy and usable cluster	55
4.4.2	Automated Operations	56
4.4.3	Central Logging	57
4.4.4	Central Monitoring	58
4.4.5	Central Audit	58
4.4.6	Backup	58
4.4.7	Capacity planning and High Availability	59
4.4.8	Autoscaling	61
4.4.9	Security	61
4.5	Expected cost	62
4.6	Summary	63
5	Deployment of Kubernetes cluster, using the two selected methods	64

5.1	Setup of a local environment	64
5.1.1	Environment variables	64
5.1.2	Source code	64
5.2	Experimental deployments using <i>kops</i>	65
5.2.1	Deployment without prerequisite steps done	66
5.2.2	Deployment with prerequisite steps done – first working cluster .	67
5.3	Experimental deployments using <i>eksctl</i>	71
5.4	Production deployment using <i>kops</i>	72
5.4.1	Generating the YAML configuration file	73
5.4.2	Creating a cluster	75
5.4.3	Testing a cluster	88
5.4.4	Deleting a cluster	91
5.4.5	Troubleshooting	91
5.5	Production deployment using <i>eksctl</i>	96
5.5.1	Generating the YAML configuration file	96
5.5.2	Creating a cluster	98
5.5.3	Testing a cluster	108
5.5.4	Deleting a cluster	108
5.5.5	Troubleshooting a cluster	108
5.6	Troubleshooting any Kubernetes cluster	111
5.7	Summary	113
6	Comparison of the used methods of Kubernetes cluster deployment	114
6.1	Production requirements which could not be satisfied	114
6.2	Time of chosen Kubernetes cluster operations	115
6.3	Additional steps needed to create a Kubernetes cluster	117
6.4	Minimal <i>EC2 instance type</i> needed for a Kubernetes worker node	118
6.5	Easiness of configuration	120
6.6	Meeting the automation requirement	121
6.7	Cost of chosen Kubernetes cluster operations	121
6.8	Summary	124

7	Conclusions	126
7.1	Lessons learned	127
7.2	Future work potential	127
	List of Figures	130
	List of Tables	132
	List of Listings	133
	References	151

Abstract

The presented master's thesis aims to **deploy Kubernetes cluster on AWS cloud, in a production environment, using two deployment methods**. The production environment is defined as an environment which satisfies selected requirements. The nine selected requirements included central logging, high availability, and automation.

Two deployment methods were chosen: using *kops* and using *eksctl*. The methods were first theoretically researched and described. Then, they were used in the empirical part to deploy Kubernetes clusters. The methods were a great help and they facilitated the clusters deployment.

These two methods of Kubernetes cluster deployment were compared. Several comparison criteria included time, cost, and the verdict whether a method satisfied all the production environment requirements. The conclusion was that each method can be used for different use cases. *Kops* allows more configuration, whereas *eksctl* is easier to configure. *Kops* method was deemed faster (in relation to cluster operations) and cheaper.

Keywords: *Kubernetes, Amazon Web Services (AWS), Amazon Elastic Kubernetes Service (EKS), eksctl, kops, deployment, operations automation.*

Streszczenie

Celem niniejszej pracy dyplomowej było **przeanalizowanie możliwości wykorzystania klastra Kubernetes w procesie produkcji oprogramowania w aspekcie instalacji i konfiguracji klastra na chmurze AWS na przykładzie dwóch wybranych metod**. Środowisko produkcyjne zdefiniowano jako takie środowisko, które spełnia wybrane wymagania produkcyjne – wyselekcjonowano dziewięć wymagań, m. in.: centralny system logowania, High Availability oraz automatyzacja operacji.

Wybrano dwie metody wdrażania klastra Kubernetes: program *kops* oraz program *eksctl*. Metody te zostały najpierw opisane teoretycznie, a następnie użyto ich w empirycznej części pracy. Obie metody stanowiły wielkie wsparcie w procesie instalowania i konfiguracji klastra.

Następnie, **obie wybrane metody wdrażania klastra Kubernetes porównano**, wykorzystując wybrane kryteria. Dotyczyły one m. in. czasu, kosztu, sprawdzenia, czy każda z metod spełnia wybrane wymogi środowiska produkcyjnego. Po tym badaniu wysnuto wniosek, że każda z wybranych metod może być użyta do wdrożenia klastra Kubernetes w zależności od konkretnych potrzeb deweloperów. Na przykład posługując się programem *kops*, dostępnych jest więcej opcji konfiguracyjnych klastra, a używając programu *eksctl* łatwiejsza staje się konfiguracja klastra. Metoda używająca *kops* okazała się szybsza (jeśli chodzi o czas wykonywania operacji typu: tworzenie, przetestowanie, usunięcie klastra) i tańsza.

Słowa kluczowe: *klaster Kubernetes, chmura Amazon Web Services (AWS), Amazon Elastic Kubernetes Service, kops, eksctl.*

1 Introduction

1.1 Topic and problem description

Microservices, Continuous Integration and Delivery, Docker, DevOps, Infrastructure as Code - these are the current trends and buzzwords in the technological world of 2020. A popular tool which can **facilitate the deployment and maintenance of microservices is Kubernetes**. Kubernetes is a platform for running containerized applications, for example: microservices. The first part of the problem, which this thesis is trying to solve is: **how to deploy Kubernetes itself**. The second part is: how to ensure that the deployment fulfills the needs of a production environment.

Any application may comprise several components, for example: a backend server, a frontend server, database. It is common knowledge, that deploying an application to a production environment, should obey a set of guidelines. It should be easy to view all the log messages generated by each of the components of the application, the application should be reachable for its end users and also, it would be nice if in a case of any component failure - that component should be available despite the failure. **Kubernetes facilitates satisfying such requirements. But, the aim of this thesis is to ensure such requirements for Kubernetes itself**. A set of requirements will be selected. Then, several ideas will be provided on how to meet each of the chosen requirements.

There are plenty methods of Kubernetes cluster deployment. Many will be described. The methods differ in relation to: how much customization they offer, which clouds they support, how much they cost. Some of the methods has existed since the Kubernetes was created, the other ones, like AWS EKS, has been invented later. The latter methods are harder to find in books tackling the Kubernetes deployment task, but there are other sources which explain how to use them (mostly: official documentation of the methods and blog posts).

There exist sources which compare several methods of Kubernetes cluster deployment. But, they are either non-formal sources (e.g. blog posts or Internet tutorials) or they do not compare the two methods selected in this thesis or they do not consider the production environment. Therefore, **this thesis has the opportunity to offer some novelty**.

1.2 Aim and scope of this study

Nowadays, the world is full of choices. The technology is exuberant. However, time, as our resource, is limited. Thus, it is often advisable to use an already existing solution instead of inventing our own. It is even better if there are many such solutions. The presented Master's thesis aims **to compare two methods of deploying a Kubernetes cluster**. Both of the methods concern AWS cloud. AWS cloud was chosen mainly because of its wide popularity and the range of provided services. Besides the two chosen methods of deployment, there are many more, including the DIY method and deploying on-premises.

This thesis attempts **to stipulate the requirements of a production environment**. Then, the requirements are used as comparison criteria to help assess the two methods of deployment. It is expected that one method could be easier to use than the other but also a method could be insufficient to satisfy all the production environment requirements. Furthermore, other criteria will be used, such as: cost of both methods and amount of problems encountered.

It is intended, that this work should **focus on the practical aspect of a Kubernetes cluster deployment**. The limitations and known issues of both methods are going to be described. Therefore, the thesis might be helpful to the engineers or consultants who are responsible for Kubernetes cluster deployment.

There are already some literature sources that compare chosen methods of Kubernetes cluster deployment. However, they are not constructed in a scientific, formal form (they are either blog posts or tutorials available in the Internet) or they do not consider a production environment.

1.3 Structure of this thesis

The presented thesis is divided into seven chapters.

The first chapter is **an introduction**. It describes briefly the topic and the problem. Then, it includes aim and scope of this study. And then, the structure of this study is presented.

The second chapter focuses on **basic information concerning: microservices, DevOps, Docker and Kubernetes**. Furthermore, AWS cloud is described there. The chapter

ends with stipulating the requirements of a production deployment.

In the next, third, chapter **the most popular methods of Kubernetes cluster deployment are described**. The two methods that will be compared in the later part are also included.

The fourth chapter is a practical one. It provides **planning and designing of the production deployment** which will be conducted later in the thesis. Some important decisions are taken here, for example: which Kubernetes version to use or which tools to use.

Then, the fifth chapter **refers to code** and it is the core part of this Master's thesis. It provides **the steps that were used to deploy Kubernetes clusters on AWS, using two methods**:

1. deployment with *eksctl* on AWS EKS service,
2. deployment with *kops* on AWS EC2 instances.

Anyone following these steps should be capable to recreate the same Kubernetes clusters as described here and thus, it should be possible to draw the same conclusions as the author of this thesis did. Furthermore, all the encountered problems and suggested solutions are provided.

The sixth chapter **presents the comparison** of the two deployment methods using chosen comparison criteria. Each subsection of this chapter deals with a single comparison criterion.

Finally, the last, seventh, chapter offers **the summary**, briefly describes the lessons learned and also provides some ideas for future work.

2 From microservices to automated orchestration

This chapter serves as an introduction to microservices, DevOps, Docker, Kubernetes and AWS. It ends with stipulating the requirements of a production deployment.

2.1 Microservices, DevOps, Continuous Delivery and Infrastructure as Code

This subchapter attempts to briefly explain a set of terms which will be used throughout this Master's thesis.

2.1.1 Microservices

Microservices may be defined as a **cloud-native architecture** which aim is to provide software systems as a package of small services. Each of these small services should be **independently deployable** and also each of them could utilize a different technological stack and platform. The services run in separate processes and communicate with each other through mechanisms like: RESTful APIs [21] because they utilize the language-agnostic protocol: HTTP [24].

Another definition of microservices is: "a microservice is simply a **self-contained service that does one thing**. If you put enough microservices together, you get an application" [8].

Microservices are often presented as an alternative to and compared to **monoliths**. For instance, the author of "Practical DevOps" [24] explains that, in comparison to monoliths, microservices have more integration points and suffer from a higher possibility of failure. Furthermore, in [8] it is written that:

- monoliths are hard to scale, both in terms of code and also in terms of teams of people,
- monoliths are easier to understand, because the code can be found in one place.

The schema presented in figure 2.1 should help to illustrate the difference between two software architecture styles.

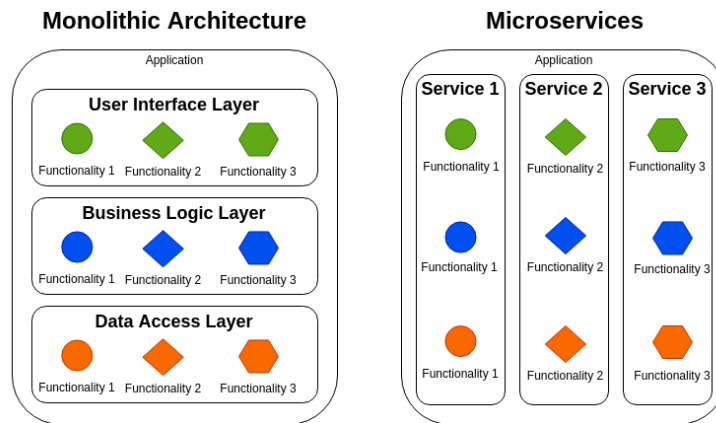


Fig. 2.1: The difference between two software architecture styles: monolith and microservices [154]

Microservices are not a panacea [8] and not a silver bullet [21], but in order to design a great software system, one should be acquainted with many solutions.

2.1.2 DevOps

DevOps and microservices have been both increasingly drawing attention since 2014, according to Google Trends. DevOps can be applied to both: monoliths and microservices, but using it for microservices promotes the importance of small teams. DevOps may be explained as a set of practices which aims **to decrease the time from the point of introducing a change to a point when the change is transferred to the production environment**. DevOps also focuses on maintaining the software quality in terms of both: code and the delivery process [21].

Moreover, DevOps can be defined as: "a movement to reduce barriers and friction between organizational silos - development, operations, and other stakeholders involved in planning, building, and running software". The definition continues, that even though the most visible aspect of DevOps may be technology, it is **culture, people and processes which have the most impact on flow and effectiveness** [23]. DevOps, as a movement, has its roots in the **Agile Manifesto**. It can be said, that DevOps obeys the first rule of the mentioned Agile Manifesto: "Individuals and interactions over processes and tools" [24].

In "Practical DevOps" [24], it is explained that DevOps spans several disciplines. Both: technical and soft skills are required to incorporate the DevOps movement. The word:

"DevOps" comes from combining: "development" and "operation". This already may indicate, that DevOps is a practice where collaboration between many teams matters and is encouraged.

2.1.3 Continuous Integration and Continuous Delivery

There are many DevOps practices: forming small teams, but also **automation and Continuous Integration (CI) and Continuous Delivery (CD)** [21, 24]. The first time that CI was written about was by Kent Beck in the book: "Extreme Programming Explained". CI was introduced to the world as Extreme Programming practice. The idea behind it was **to continuously — meaning very often — verify a source code**. CI represents a paradigm shift, because without CI, a software may be deemed broken unless somebody proves otherwise. With CI, a software is proven to work with every change added to source code [28].

There are many advantages of using CI [28]:

- CI helps to identify the change in a source code that resulted in failing tests,
- bugs may be caught earlier in the delivery process which is cheaper and faster when compared to fixing them in already deployed production environment,
- delivery process is automated, thus human error is limited,
- delivery process is automated, thus it is repeatable and easily reproducible,
- delivery process is clearly stipulated in code,
- CI helps different team members to communicate [17].

In order to incorporate the practice of Continuous Integration, a **Continuous Integration server** is needed. Examples of such a server are: Jenkins [153], GoCD [151]. The second ingredient needed is a **CI pipeline**. A CI pipeline consists of several stages which provide steps, to be performed, in order to release the software. There may be a lot more stages, for instance: the test stage can be split into many stages, each running different kind of tests (integration, functional, non-functional, acceptance, etc.) [17, 28]. The pipeline starts with a user uploading their code onto a version control system. A CI server should pick up a change and initiate the pipeline run [24]. An example pipeline is illustrated in figure 2.2.

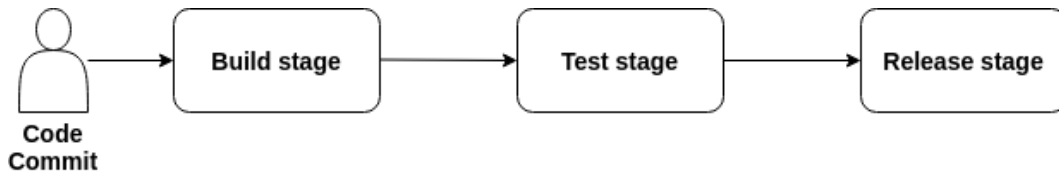


Fig. 2.2: A simple example CI pipeline, consisting of a few stages, starting with a developer uploading a change of code

Each stage of a pipeline may be either passed or failed. The failed stage indicates that some particular code change cannot satisfy the requirements of that particular stage. The stages are run in a specific order. In figure 2.2, the order is demonstrated with arrows. The first stage is build, then test, then release. Should the build stage fail, then neither the test nor the release stage will be run.

There exists an extension of continuous integration and it is called: **Continuous delivery (CD)**. CD verifies that a piece of software is always in a deployable state. CD is thus even more attractive than CI, because it offers even more automation [132].

2.1.4 Infrastructure as Code

Together with automation and DevOps there comes the term: **Infrastructure as Code (IaC)**. IaC may be understood as an approach to automate the infrastructure based on practices from software development. Consistent and repeatable routines are encouraged for infrastructure provisioning and configuration. There are three core practices required to implement IaC: **define everything as code** (e.g. the configuration can be saved as YAML files), **continuously validate the code** and, **build a system from small, loosely-coupled pieces** [23].

Back in time, developers dealt with software, while operations teams worked on hardware and the operating systems. Now, the hardware is in the cloud and thus, it can be handled in the same way as software. **The DevOps movement brings software development skills to operations**. It concerns both: tools and workflows [8]. The movement from deployment on-premises to the cloud also deserves a few words, but it will be handled in the subchapter: 2.2.

2.2 AWS - The Amazon Cloud

This subchapter offers an introduction to cloud computing and AWS cloud and also the explanation why this particular cloud was chosen.

There are three revolutions going on [8]:

- the creation of the cloud,
- the dawn of DevOps,
- coming of containers.

These revolutions are interlinked and happen all at once. This subchapter focuses on the cloud revolution. The days of "bare-metal", before cloud, are referred to as "**Iron Age**" [8, 23]. The current times (after cloud was invented) are called "**Cloud Age**" and the cloud infrastructure - **Infrastructure as a Service (IaaS)**. IaaS allows to outsource not only the hardware but also the software. The outsourced software involves: operating systems, networking scripts, monitoring logic etc. **Managed services** can take care of many non-functional requirements. There is no more grand upfront investment. Having a large-scale system to build may have cost a fortune in the past. Back then, the computing power was a capital expense and now — it is an operational one. Now, there is no fixed cost and the expense depends most often on cloud resources utilization [8, 27].

Cloud computing may be defined as a model that provides end users with access from any device, as long as the device has an Internet access, to a shared set of cloud resources. The cloud resources involve various servers and services. Taking a step away, there may be differentiated four **deployment models**: private cloud, community cloud, public cloud, hybrid cloud. Private cloud means that the software is deployed on-premises, locally. Apart from that, there are also **service models**, which the most popular are [12]:

- Infrastructure as a Service (IaaS),
- Platform as a Service (Paas),
- Software as a Service (Saas).

Public clouds like: Amazon AWS, Google Cloud Platform or Microsoft Azure represent IaaS. In the Amazon whitepaper "Architecting for the Cloud" [27], many **cloud computing benefits** are listed:

- near zero upfront infrastructure investment,
- just-in-time infrastructure — meaning that it is simple to scale the applications,
- more efficient resource utilization — resources can be reserved or required on demand (instantly),
- usage-based costing — users do not pay for allocated but unused infrastructure,
- reduced time to market — some jobs can be run parallelly.

Furthermore, AWS offers a **highly reliable and scalable infrastructure** [27], ensures **attractive SLAs** (e.g. AWS promises to keep a Monthly Uptime Percentage of at least 99.99% for compute resources) [31] and it is also **widely adopted and provides many various services** [104]. These are the reasons why AWS was chosen for this thesis. The charts in figures 2.3 and 2.4 present the popularity of AWS solutions.

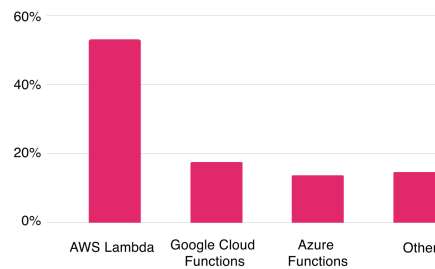


Fig. 2.3: Hosted serverless platforms preferred by CNCF community during September and October 2019 [104]

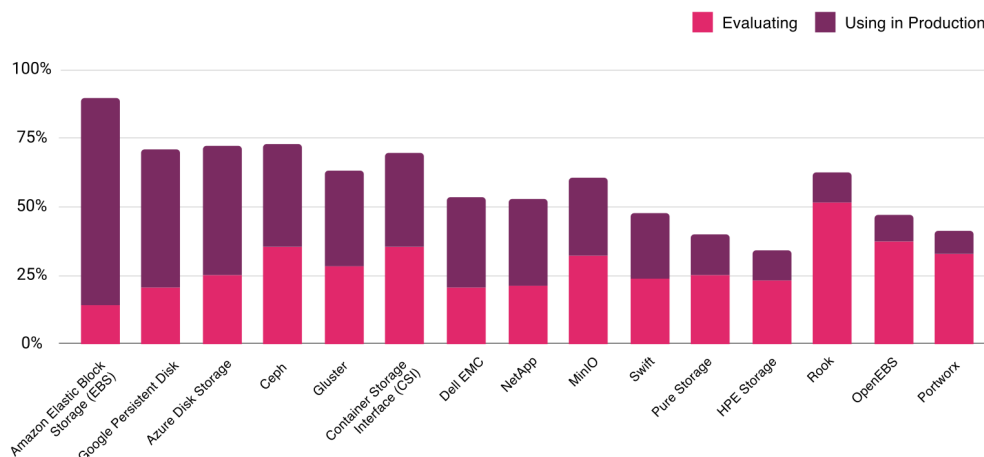


Fig. 2.4: Cloud native storage preferred by CNCF community during September and October 2019 [104]

CNCF is an abbreviation from *Cloud Native Computing Foundation*. It is a non profit organization which fosters communities to support growth and health of cloud native open source software. CNCF adopts Cloud Native projects and catalogs them, depending on their maturity level. They also regularly conduct surveys concerning the Cloud Native and container-related technologies [105].

2.3 Docker containers

This subchapter provides an introduction to Docker containers and also compares them to Virtual Machines.

In the old times, software was installed **on physical hardware directly**. Then, **virtualization and Virtual Machines (VMs)** were invented. The next great invention, after VMs, were **Docker containers**. In 2013 Docker was created as a standard way to manage containers[20].

There are many advantages of using VMs and Docker containers in comparison to physical hardware. Docker containers and VMs:

- provide safer, isolated environment — facilitate experiments,
- are easier to automate, manage and replace,
- provide reproducible environment,
- limit "works on my machine" problem,
- offer more security, e.g. an external user may have access only to a Docker container, not to a whole physical host,
- have additional features, e.g. memory replication.

It is claimed, that containers may be replacing VMs. One reason for this might be that there is a significantly lower overhead in case of deploying containers compared to VMs [22]. Furthermore, in comparison to VMs, containers provide faster resource allocation. There are many implementation of containers, but among them, Docker is probably the most adopted [18].

Docker uses some major Linux kernel features like: **namespaces and cgroups**. Namespaces control and limit the amount of resources a process can use, while cgroups manage the resources of a process group. They both help to provide isolation and resource limi-

tation [10, 20].

In order to be able to work with Docker containers, one has to know the **difference between a Docker container and a Docker image**. A Docker image is a read-only template. It has some software installed and configured. Each image has a name and a tag, which serves the same purpose as a software version. One image may be based on other image. There are many images available which are open-source and free, e.g. "debian:10.3" [146], "ubuntu:16.04" [147]. It is easy to build one's own image, basing on the available images [149].

Below, there is an example of a source code which creates (builds) a Docker image. Just one file is enough for this task: **a Dockerfile** (listing 2.1).

```
FROM ubuntu:16.04

RUN apt-get update && apt-get install -y openssh-server
RUN mkdir /var/run/sshd
RUN echo 'root:THEPASSWORDYOUCREATED' | chpasswd
RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin\
yes/' /etc/ssh/sshd_config

# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session\
optional pam_loginuid.so@g' -i /etc/pam.d/sshd

ENV NOTVISIBLE "in_users_profile"
RUN echo "export _VISIBLE=now" >> /etc/profile

EXPOSE 22
CMD ["/usr/sbin/sshd", "-D"]
```

Listing 2.1: A Dockerfile to build a Docker image with SSH server installed. Based on Docker official documentation [150]

Docker containers are claimed to be **lightweight and portable**. Thus, containers are suitable for microservices. Since microservices are loosely-coupled, failure of one microservice should not affect other microservices of an application. This architectural style is characterized by fine granularity and therefore, it makes scaling more flexible and efficient [13].

Docker containers are utilized by big companies like: **Netflix or Twitter** [15]. Docker is a technology described thoroughly in many literature sources and therefore there is no

sense in repeating it. A curious reader is referred to read the official Docker documentation [149].

2.4 Kubernetes as a Docker containers orchestration system

This subchapter describes what Kubernetes is and what problems it solves. Furthermore, the chapter acknowledges Kubernetes popularity and briefly introduces chosen Kubernetes objects.

The process of deploying multiple containers of one application can be optimized through automation. This kind of automation is referred to as **orchestration** [10].

There was a need to automatically deploy and manage a large number of services at global scale on millions of servers. Thus, Google developed Borg. Borg is a private orchestration system. It is very powerful, but also very much coupled to Google's own internal and proprietary technologies. Therefore, in 2014 Google launched a new project called: **Kubernetes**. The name comes from a Greek word which means "helmsman, pilot". Apart from Kubernetes, there are alternative solutions invented, other orchestrators, but Kubernetes **won the orchestration wars** [8, 22].

"Kubernetes does the things that the very best system administrator would do: automation, failover, centralized logging, monitoring. It takes what we've learned in the DevOps community and makes it the default, out of the box." These are the words uttered by Kelsey Hightower — a Google employee with legendary contributions to Kubernetes [8].

Kubernetes was built **to make deployments easy, to reduce the time and effort** which needs to be poured into them. To achieve this goals, Kubernetes offers **a wide variety of features**[6, 8, 18]:

- creating, destroying, replicating containers,
- rolling updates of containers,
- built-in health checks (liveness and readiness probes),
- autoscaling,
- redundancy and failover — to make the applications deployed on top of Kubernetes

more resilient and reliable,

- being provider-agnostic — Kubernetes can be deployed on-premises and in the cloud and in both cases it will provide the same set of features, thus, it may be said that Kubernetes unifies the underlying infrastructure,
- utilizing provider-specific (sometimes referred to as: vendor-specific) features, e.g. AWS load balancer or Google Cloud load balancer,
- self-healing,
- service discovery,
- load balancing,
- storage orchestration.

Kubernetes is the first project of the CNCF [18]. Kubernetes is already **used by some well-known entities**, e.g. NASA [9], F-16 jets [106], Zalando [3], ING [2], booking.com [1]. Apart from that, the **Kubernetes popularity is growing**, basing on the information from CNCF Survey 2019, which states that 78% of respondents use Kubernetes in production and that this is a huge jump from 58% using in the previous — 2018 year [104]. Even the older CNCF Survey from year 2017 shows Kubernetes as number one cloud management platform — it is presented in figure 2.5.

The **basic working unit in Kubernetes is a pod**. A pod is an abstraction and it represents one application, a set of containers. Two things are crucial to know about pods [22]:

- all the containers defined in a pod will be deployed on one machine,
- a pod has one IP address assigned and all the containers defined in this pod share the same IP address.

A pod is all that is needed to deploy a set of containers. Pod is a Kubernetes object and there are many more objects, for example: deployment, replica sets, service, ingress. The important thing to note is that these **objects can be configured in a declarative way**, thanks to YAML files [4].

Kubernetes provides mechanisms for maintaining, deploying, healing and scaling containerized microservices. Thanks to that, Kubernetes hides the complexity of microservices orchestration. It is much easier to satisfy non-functional requirements of microser-

vices deployment by using Kubernetes. An example of such requirements may be: availability, healing, redundancy or autoscaling [13].

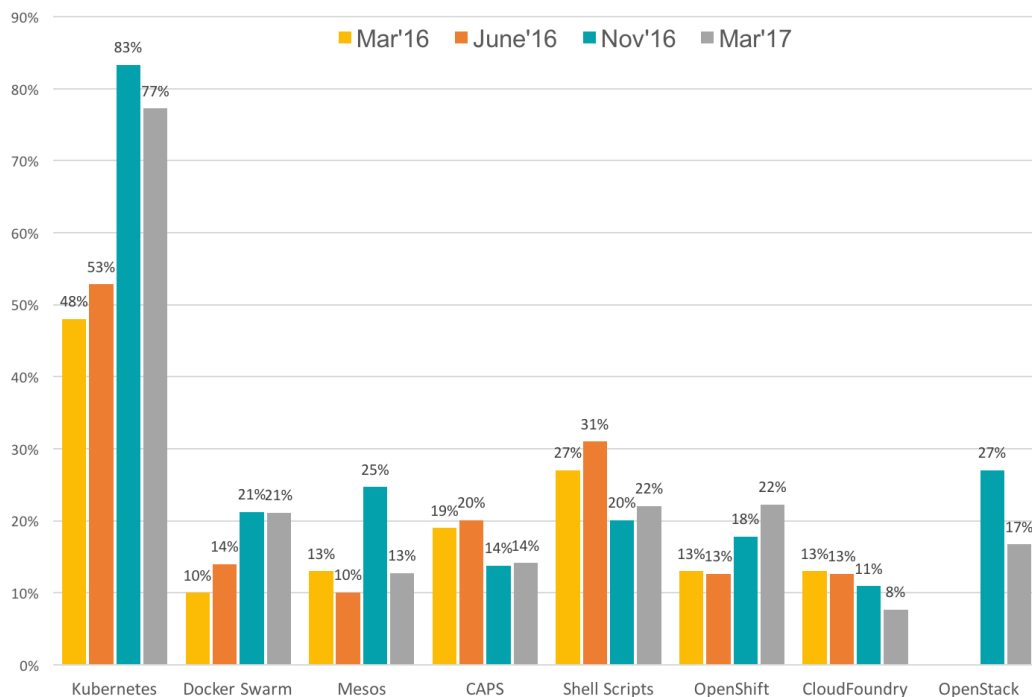


Fig. 2.5: Results of CNCF Survey from year 2017 showing Kubernetes as number one cloud management platform [103]

2.5 Kubernetes architecture

This subchapter contains a high level description of the Kubernetes architecture. The focus is on the responsibilities of each Kubernetes component.

2.5.1 Kubernetes cluster

A **Kubernetes cluster** may be defined as a collection of storage and networking resources which are used by Kubernetes to run various workloads [14]. Another definition states that a Kubernetes cluster is a single unit of computers which are connected to work together and which are provisioned with Kubernetes components [126].

A cluster consists of two kinds of instances: masters and nodes. An instance can be a virtual machine or a physical computer [13, 126]. This is depicted in figure 2.6. Kubernetes

nodes communicate with Kubernetes master.

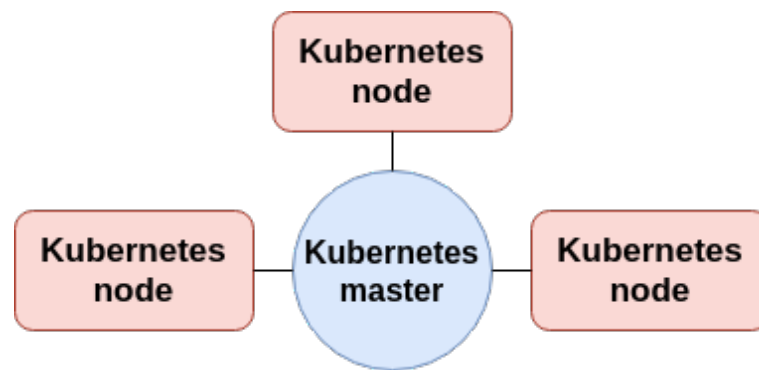


Fig. 2.6: Kubernetes cluster diagram depicting a master-slave architecture

2.5.2 Masters and nodes

The role of the master is to manage the cluster. This means that master: schedules applications, maintains their desired state, scales them, handles events, manages nodes. **Nodes serve as the worker machines.** They are responsible for running containers and handling container operations. Masters schedule containers to run on nodes [14, 126]. At least one node and one master is needed in a Kubernetes cluster. In order to provide fault-tolerance and high availability in production environments, multiple master and multiple node instances are run [119].

The instances in a Kubernetes cluster (masters and nodes) are hosts to several **Kubernetes components**. There are **master components**, used to control the cluster and there are also **node components**, run on each node. All the components are presented in figure 2.7:

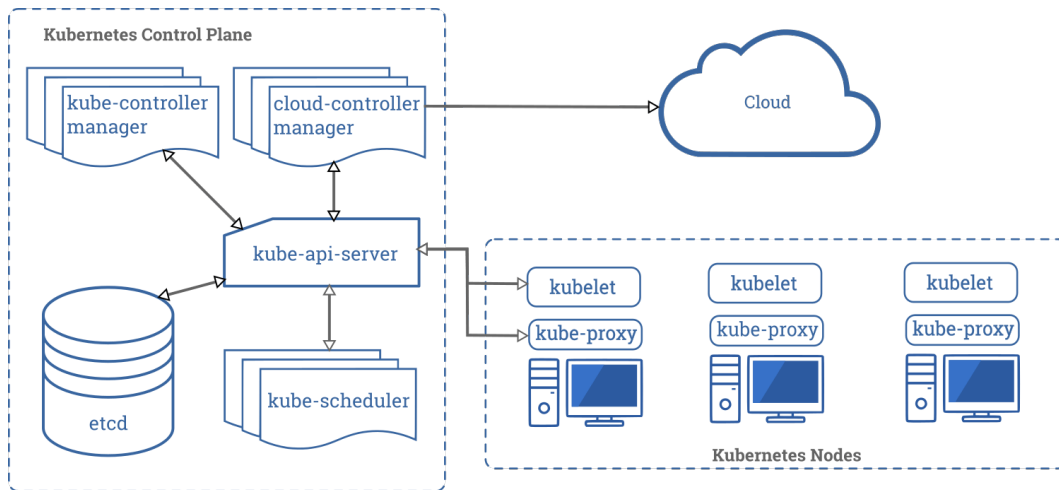


Fig. 2.7: Kubernetes components including master and node components [119]

2.5.3 Components

The master components are also known as the control plane's components. They are as follows [14, 119]:

- Etcd (a key-value store),
- API server,
- Scheduler,
- Controller manager and Cloud Controller manager.

Etcd stores the entire cluster state. It is a highly-available key-value store. It is enough, for a test Kubernetes cluster, to deploy one instance of Etcd. However, for the purposes of high availability and redundancy, a 3-node or even 5-node Etcd cluster is typical. It is recommended to have a back up plan for the data stored in Etcd[14, 119]. The name *etcd* follows a naming convention within the UNIX directory structure. In UNIX, all system configuration files are contained in a folder called `"/etc"`. The last letter "d" stands for "distributed" [131].

API server exposes the Kubernetes REST API. It allows the nodes to communicate with the master and it also allows end users to interact with the cluster. Thanks to the fact that API server is stateless and that all its data is stored in etcd, API server can easily scale horizontally. The main implementation of a Kubernetes API server is kube-apiserver[14,

119, 126].

Scheduler is responsible for assigning containers to nodes. Scheduler selects a node for a container to run on. It considers a range of factors: resource requirements, various constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines. The implementation is known as kube-scheduler [14, 119].

Controller manager runs controller processes such as: watching the shared state of the cluster and making changes needed to move the current state into the desired state. Controller manager is a collection of separate managers, but they are all compiled into a single binary and run in a single process in order to reduce complexity. The controllers consist of: Node Controller, Replication Controller, Endpoints Controller, Service Account and Token Controllers. The implementation of Controller manager is kube-controller-manager [14, 119].

Cloud Controller manager interacts with a specified underlying cloud provider (e.g. Amazon Web Services or Google Cloud Platform). It is implemented by cloud-controller-manager and therefore, it allows the Kubernetes code and the cloud vendor's code to evolve independently [119].

As aforementioned, there are also node components. They run on both: masters and nodes. They are as follows [14, 119]:

- Kubelet,
- Proxy,
- Container Runtime.

Kubelet oversees the communication with the master components (by monitoring API server for changes) and makes sure that containers, described by a Pod, are running and healthy (so it manages a Pod lifecycle). A Pod is a simple object from a Kubernetes API and it represents a set of containers. Containers which were not created by Kubernetes are not managed by Kubelet [14, 119].

Proxy is implemented by kube-proxy. It is a network proxy and it implements a part of the Kubernetes Service concept, which means that it is responsible for exposing an application as a network service and it provides load balancing [14, 119].

Container Runtime is the software which is responsible for operating containers. Sev-

eral container runtimes are supported: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface). It is a design policy of Kubernetes that is ought to be decoupled from a specific container runtime. Under some circumstances it should be possible to switch from one container runtime to another or to use multiple of them at once. Originally, Kubernetes was designed to manage only Docker containers [14, 119].

2.5.4 Other services

Apart from master and node components, there are also **add-ons, extensions, and third party tools** which communicate with Kubernetes by the API server and which provide additional functionality. Examples of such services are: DNS, Vertical Pod Autoscaler, Cluster Autoscaler, Istio, Kubernetes Dashboard, kube-ops-view, node-problem-detector, etc [8].

2.5.5 The Kubernetes networking model

Kubernetes states a few **networking requirements** [112]:

- pods on a node must be able to communicate with all pods on all nodes without Network Address Translation (NAT),
- agents on a node (e.g. system daemons, kubelet) must be able to communicate with all pods on that node.

There are many available options that help with satisfying these requirements, e.g. AWS VPC CNI for Kubernetes, Azure CNI for Kubernetes, Flannel, OpenVSwitch, Project Calico, etc [112]. CNI stands for Container Networking Interface and it is a specification and also a set of libraries for writing network plugins to configure network interfaces in Linux containers. A CNI container is bound to have its own IP address. For Kubernetes, **each pod has its own IP address**, so the pod is the CNI container [14]. **Containers that belong to a one pod share the same IP address**, which means that these containers can reach all reach each other's ports on localhost and also that none two containers should expose the same port. This model is known as "IP-per-pod" [112].

2.6 Production deployment requirements

This subchapter explains what a production deployment is and what requirements it must met.

2.6.1 Multiple environments in software deployment

First, it is helpful to distinguish between the terms: '**infrastructure stack**' and '**environment**'. They both may be defined as a collection of infrastructure resources. The difference is that **an environment is conceptual, while a stack is concrete**. A stack is defined with code, particularly when Infrastructure as Code is applied, and managed using tools. However, an environment serves to fulfill a predetermined purpose. Multiple environments can run an instance of the same system [23].

Typically, there are **two reasons for which multiple environments are in use**: to support a release delivery process and to run multiple production instances of the system. The first reason allows to have a particular build of an application (e.g. a git commit or a specified version of code) well tested. Such a build has to go through many different environments, e.g.: testing, staging and production. When a build does not pass all the stages in the former environments, it will not be promoted to the production environment[23, 28].

To briefly explain the second reason for multiple environments: they are used in order to ensure fault-tolerance (when one environment fails, the other can take over), scalability (the work can be spread among many clusters), segregation (it may be decided to handle a group of customers using one environment and the other group with the other environment, e.g. for latency purposes) [23]. Well-known examples of running multiple production deployments can be **Blue-green deployments or Canary deployments** [17].

2.6.2 Production deployment requirements

Throughout this work a production deployment means such a deployment which targets the production environment. A list of **requirements for a production deployment**, gathered through the literature, is provided below:

- **Central Monitoring** — this is helpful when troubleshooting a cluster [8, 14, 184, 187].
- **Central Logging** — this is a fundamental requirement for any cluster with number of nodes or pods or containers greater than a couple [14, 20, 184].
- **Audit** — to show who was responsible for what action [187].
- **High Availability** — authors of [14] go even further and state that the cluster should be tested for being reliable and highly available **before** it is deployed into production [8, 14].
- **Live cluster upgrades** — it is not affordable for large Kubernetes clusters with many users to be offline for maintenance [14].
- **Backup, Disaster Recovery** — [8, 14, 187].
- **Security, secrets management, image scanning** — security at many levels is needed (node, image, pod and container, etc.) [8, 14, 184, 187].
- **Passing tests, a healthy cluster** — 'if you don't test it, assume it doesn't work' [8, 14].
- **Automation and Infrastructure as Code** — in production environment a versioned, auditable, and repeatable way to manage the infrastructure is needed [14][187].
- **Autoscaling** — if application deployed on a Kubernetes demand more resources, then a new Kubernetes node should be automatically created and added to the cluster. However, autoscaling, even though nice to have, is not that important [8].

2.6.3 Monitoring as production environment requirement

Monitoring helps to ensure that a cluster is operational, correctly configured and that there are enough resources deployed. Monitoring is also indispensable for debugging and troubleshooting [14]. The third reason for using a monitoring system is that historical data is needed for planning purposes. The monitoring strategy should cover four areas [28]:

- configuring the infrastructure in such a way that it is possible to collect the data,
- storing the data,
- providing dashboards, so that data is presented in a clear way,

- setting up notifications or alarms to let people know about certain events.

Monitoring provides various **metrics**, e.g. CPU usage, memory utilization, I/O per disk, disk space, number of network connections, response time, etc. Thus, it is helpful at many different levels: at hardware, operating system, middleware and application level. There is a wide range of **available open source and commercial tools** to take care of monitoring: Nagios, OpenNMS, Flapjack, Zenoss, Tivoli from IBM, Operations Manager from HP, Splunk, etc. [28]. Solutions recommended for a Kubernetes cluster are: Heapster combined with InfluxDB as backend and Grafana as frontend and also cAdvisor [14]. A nice feature of Grafana are its dashboards. Example Grafana dashboard is presented in figure 2.8.

Grafana also works well with Prometheus, which is a monitoring system and a time series database. Prometheus is also a CNCF graduated project [166, 167]. If a system (like Kubernetes) is deployed on AWS, another solution for monitoring and logging may be: Amazon CloudWatch [30].

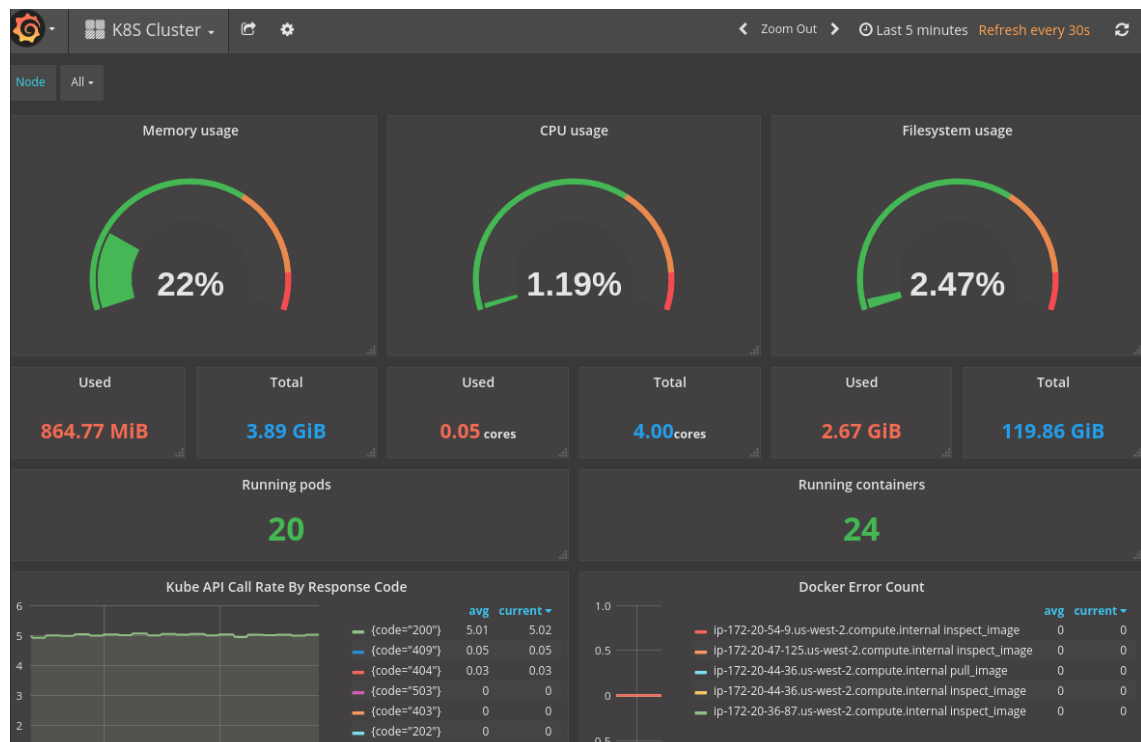


Fig. 2.8: Example Grafana dashboard for a Kubernetes cluster, showing among others: Memory, CPU and File system usage [159]

Another solution for monitoring is *Kubernetes dashboard*, which is a built-in solution and doesn't require any customization. Heapster, InfluxDB and Grafana are great for heavy-duty purposes, whereas Kubernetes dashboard is probably able to satisfy the majority of monitoring needs of a Kubernetes cluster [14, 20]. Example dashboard provided by Kubernetes dashboard is depicted in figure 2.9.

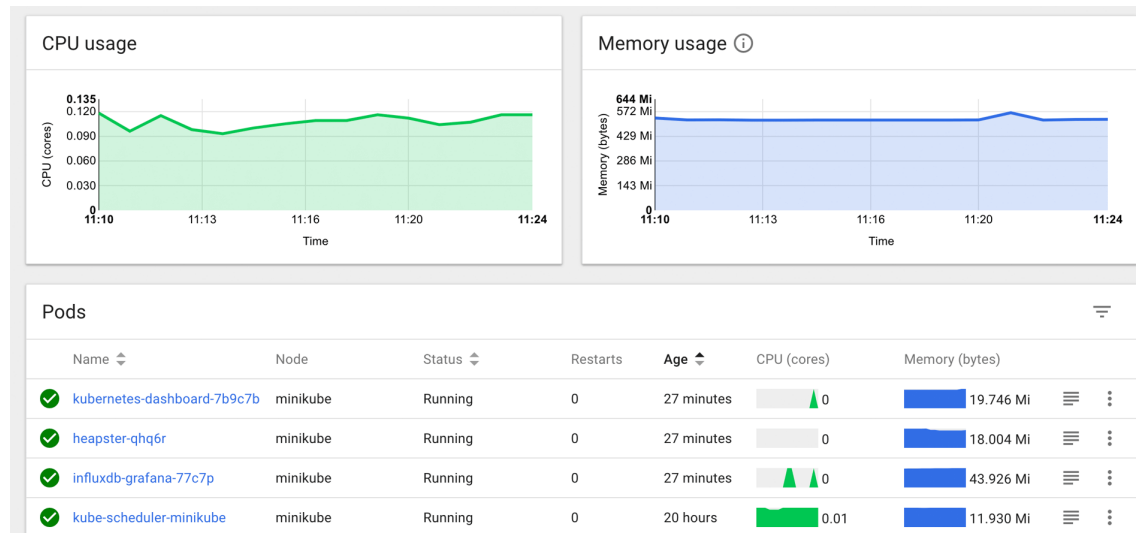


Fig. 2.9: Kubernetes dashboard depicting CPU and Memory usage by Kubernetes pods[127]

2.6.4 Logging as production environment requirement

Kubernetes dashboard has also a feature which makes it able to show **log messages** of a single container deployed on Kubernetes [14]. **Centralized logging** is essential for a production cluster, because usually there are a lot of pods (and containers) deployed, each generating many log messages. It is impossible to require a Kubernetes administrator to login into each container for the purpose of getting the logs.

The second reason for the importance of centralized logging is that **containers are ephemeral** — the log messages kept inside the containers would be lost after a container is redeployed. **Popular solutions** are: Fluentd, Elasticsearch, Kibana [14], Logstash [20] and Graylog [98, 135]. It is also important to consider that log messages in Kubernetes cluster are generated from many sources: from end-user applications, nodes, Kubernetes system containers and there are also **audit logs** in the form of e.g. API server events [136].

For the purposes of auditing, when deploying on AWS, one can use AWS CloudTrail [52].

2.6.5 High Availability as production environment requirement

While administering a Kubernetes cluster, there is a high probability that something will go wrong. Components, network can fail, configuration can be incorrect, people make mistakes and software has bugs. Failure classification has been described in [29]. This has to be accepted and a system should be designed in such a way that it is **reliable and highly available (HA)** despite of many problems. The ideas of how to ensure high availability is as follows [14]:

- **Redundancy** — means having a spare copy of something. Kubernetes uses Replica Sets or Replication Controllers to provide redundancy for applications deployed on Kubernetes. Five redundancy models were summarized in [26]. Some of them require an active replica (running) and other passive (or standby).
- **Hot Swapping** — can be explained as replacing some failed component on the fly, with minimal or ideally zero down-time. Actually, hot swapping is quite easy to implement for stateless applications. For stateful applications, one has to keep a replica of a component (see redundancy).
- **Leader election** — it is a pattern used in distributed systems. Whenever there are many servers fulfilling the same purpose to share the load. One of the servers must be elected a leader then and certain operations must go through it. When the leader server experiences a failure, other server can be selected as new leader. This is a combination of redundancy and hot swapping.
- **Smart load balancing** — used to share and distribute the load.
- **Idempotency** — means that one request (or some operation) is handled exactly once.
- **Self-healing** — means that whenever a failure of one component happens, it is automatically detected and steps are taken (also automatically) to get rid of the failure.
- **Deploying in a cloud** — a goal is to be able to physically remove or replace a piece of hardware, either because of some issues or because of preventative maintenance

or horizontal growth. Often this is too expensive or even impossible to achieve [29]. Traditional deployments on-premises forced administrators to do a capacity planning (to predict the amount of computing resources). Thanks to the on-demand and elastic nature of the clouds, the infrastructure can be closely aligned to the actual demand. It is also easy to scale applications deployed on a cloud, because of the fundamental property of the cloud: elasticity [27].

Generally speaking, **"highly available systems are fault tolerant systems with no single point of failure"** [26]. In order to introduce HA for the Kubernetes cluster the following ideas could be incorporated [14]:

- Deploy etcd as a cluster, not just one instance of etcd.
- Ensure redundancy for API server.
- Deploy multiple master instances and ensure a load balancer in front of them.
- Ensure that node instances are reliable: the Docker daemon and the Kubelet daemon (the node agent which runs on each node [117]) should restart automatically in case of failure.
- Apply RAID to ensure redundancy of data storage or apply Key-Value Multi-Device (KVMD), a hybrid data reliability manager[7] or let cloud provide storage availability. The intention of RAID is to spread the data across several disks, such that a single disk failure will not lose that data [177].

Furthermore, it may be needed to test high availability. This can be done by inducing a predictable failure and verifying if the system behaves as expected [14]. Such a kind of testing, where one or more cluster nodes or pods is killed is called Chaos Monkey, after the tool developed by Netflix. There are also ready to use tools basing on the idea of Chaos Monkey: chaoskube, kube-monkey, PowerfulSeal [8].

2.6.6 Automation as production environment requirements

When it comes to **automation**, many guidelines can be found in [28]. Below are some of them listed:

- **Every Change Should Trigger the Feedback Process** — means that every change in code should trigger some pipeline and should be tested (including unit tests,

functional acceptance tests, nonfunctional tests). The tests should happen in an environment which is as similar as possible to production. Some tests may run in production environment too [23, 28].

- **The Feedback Must Be Received as Soon as Possible** — this also involves another rule: fail fast. This guideline suggests that faster tests (or less resource-intensive tests) should run first. If these tests fail, the code does not get promoted to the next pipeline stages, which ensures optimal use of resources [28].
- **Automate Almost Everything** — generally, the build process should be automated to such extent where specific human intervention or decision is needed. But there is no need to automate everything at once [23, 28].
- **Keep Everything in Version Control** — this means that not only application source code but also tests, documentation, database configuration, deployment scripts, etc. should be kept in version control and that it should be possible to identify the relevant version. Furthermore, any person with access to the source code should be able to invoke a single command in order to build and deploy the application to any accessible environment. Apart from that, it should be also clear which version in version control was deployed into each environment [28].
- **If It Hurts, Do It More Frequently, and Bring the Pain Forward** — if some part of the application lifecycle is painful, it should be done more often, certainly not left to do at the end of the project [28].
- **Idempotency** — the tools used for automation should be idempotent, which means that no matter how many times the tool is invoked, the result should stay the same [23].

Together with automation, there are two inextricably entwined terms: Infrastructure as Code and DevOps. As these two terms have been already explained in this work, now let us focus on the essential tools needed to introduce the automated application lifecycle. First, a framework for **Configuration Management** is needed. Examples involve: Puppet, CFEngine [23, 28], Chef [148], Ansible [168], SaltStack [170], etc. These tools help to declaratively define what packages should be installed and how should they be configured in a virtual machine or a container or a physical server [28]. They can help

prevent **configuration drift** in a large number of computers [20]. A configuration drift is a difference across systems that were once identical. It can be imposed by a manual amendment and also by automation tools which propagated a change only to some of the instances[23]. There are also stack-oriented tools, which follow the same declarative model: Terraform [141] and CloudFormation [23]. Another type of needed tools is a building server, where the examples are: Jenkins, GoCD, Travis - they were already mentioned earlier.

2.6.7 Security as production environment requirement

Security is another essential aspect of production deployment and, as mentioned above, it touches many levels. A node breach is a very serious problem and it can happen by someone logging to the instance or having physical access to it. The latter is easily mitigated by not deploying on bare-metal machines but on a cloud instead [14]. The former demands some hardening done. There are several ideas that can be implemented for a Kubernetes cluster specifically listed below:

- Ensuring that data is encrypted in transit by using secure api server protocol (HTTPS instead of HTTP) [14].
- Ensuring proper user and permissions management by configuring authentication, authorization, security accounts and admission control in API server [14]. When setting up authorization, it is wise to apply **the principle of least privilege**. This principle recommends that only the needed resources or permissions should be granted [8].
- Utilizing Role-Based Access Control (RBAC) to manage access to a cluster [8].
- Ensuring security keys management and exchange [14] by implementing for example automated key rotation.
- Ensuring that used Docker images are neither malicious (deliberately causing some harm) nor vulnerable (allowing some attacker to take control) by keeping them up-to-date and maintaining them instead of using the publicly available ones or by using a private Docker registry [14].
- Using minimal Docker images because the fewer programs there are installed in an

image, the fewer potential vulnerabilities there are [8].

- Maintaining a log or audit system [14].
- Utilizing network policies which act in a whitelist fashion and can open certain protocols and ports [14].
- Using secrets. Kubernetes has a resource called: secret, but the problem is, that Kubernetes stores secrets as plaintext in etcd. This, in turn, means that steps should be taken in order to limit direct access to etcd [14].
- Preferring managed services, because they will have many security measures already implemented [8].
- Avoid running processes as root user in Docker containers [8].
- Using available programs for security scanning [8].

Figure 2.10 security features provided by Kubernetes API server.

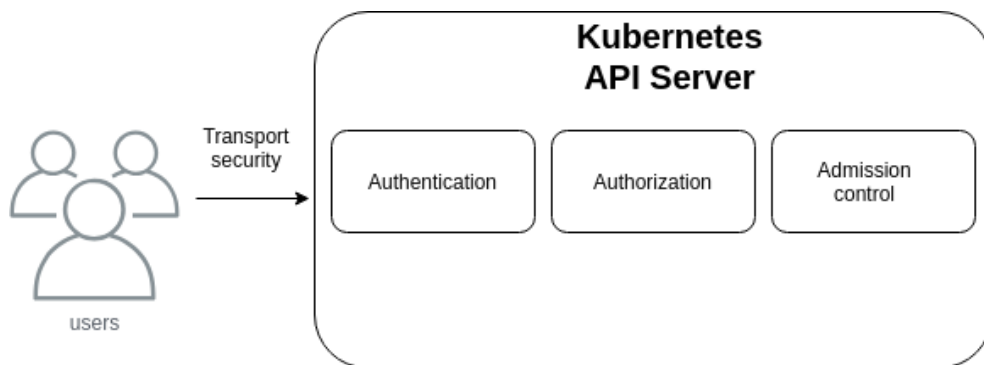


Fig. 2.10: Security features which a request sent to API Server goes through [5]

There are many more security measures that Kubernetes administrators and end-users could apply [8, 14].

2.6.8 Disaster recovery as production environment requirement

Disaster recovery can be understood as the process which an organization has to undergo after a service disruption happened in order to resume normal services. It is vital to know what actions are necessary to overcome the disaster. This set of predefined procedures is known as **Disaster Recovery Plan**. Furthermore, disaster recovery is not the same as fault tolerance. The latter ensures that a system will withstand and resist the failure [16].

Disaster recovery is an essential requirement of any business where continuity matters. In order to plan disaster recovery well the following key parameters should be considered: the initial cost, the cost of data transfers and the cost of data storage. Significant costs may be a reason why, in the past, around 40-50% of small businesses had no DRP and did not intend to change this. However, cloud computing provides affordable solutions, because of the employed model "pay-for-what is used". Another **advantage of the cloud is that it is fairly easy to use resources deployed in multiple geographical areas**. This is desired, because one the major concepts in a DRP is the geographical separation of the system servers and its backup [25].

Key metrics that can be taken into consideration while planning disaster recovery are [16, 25]:

- Recovery Point Objective (RPO),
- Recovery Time Objective (RTO).

RPO can be defined as the time between two successive backups. Thus, the time between the last backup and the the disaster, which is de facto the time of data loss, is maximally equal to RPO. **RTO** can be understood as the time needed to recover from the disaster, when the server experiences downtime [25]. RPO and RTO are illustrated in figure 2.11.

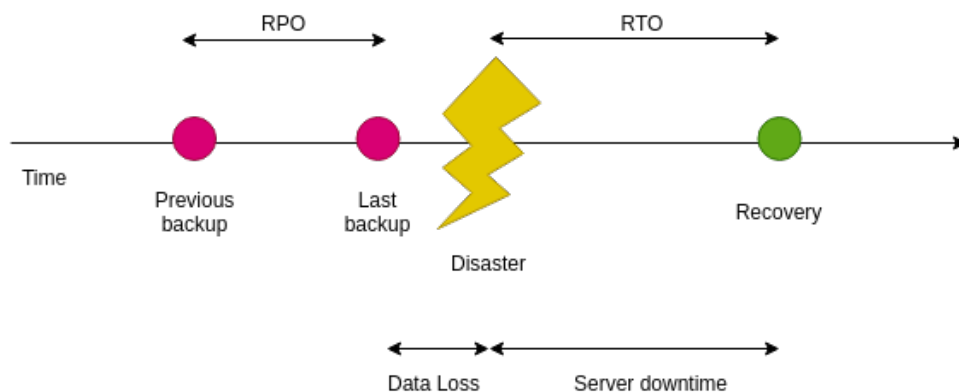


Fig. 2.11: RPO and RTO illustrated in relation to time

Cloud services mitigate some risks that persistent data storage has. For example: cloud services provide high-available data storage by replicating it across different geographical locations. However, replication is not the same as backup and it does not protect against accidentally deleting a volume or against a misconfigured application over-

writing the data. Thus, backup is still needed. In order to backup Kubernetes, the etcd database has to be backed up. Apart from that, each application deployed on top of Kubernetes should be backed up on its own [8]. There are already available services that help with Kubernetes backup: *Velero* [8].

2.6.9 Testing as production environment requirement

Before the production Kubernetes cluster is ready for end-users, it must be verified that **it works and is healthy**. Every component and every node should be tested proving that it is working as expected. Sometimes, applications expose a custom **health endpoint**[20]. E.g. kube-scheduler does that. Thanks to that, it is possible to verify regularly that a service is performing correctly by sending a request to the health endpoint. Usually, a HTTP response code of 200 indicates correct status of the service [158]. Kubernetes can monitor the health endpoints with **liveness probes**. Based on a specified health endpoint response, Kubernetes can restart the faulty container [113].

The tests should be incorporated into a CICD pipeline. "Having a comprehensive test suite is essential to continuous integration" and test-driven development is a vital practice in this context [28]. There is even a possibility to implement **test-driven** changes to deployment environments. The recipe for achieving that is described in "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" [28].

2.7 Summary

This chapter described the production environment requirements found in the literature. The example ideas of how to satisfy most of the requirements were provided. Therefore, it is now possible to choose the requirements for Kubernetes cluster deployment.

3 Available Kubernetes cluster deployment methods

Here multiple methods of a Kubernetes cluster deployment are presented. Two chosen methods are described in a more detailed way.

Kubernetes is **not trivial to deploy**. In order to deploy a usable cluster, there are at least two machines needed: one master and one node. On each of the machines several components must be installed. Apart from that, there are also requirements concerning networking to be met (described in subchapter: 2.5.5) and a bunch of non-functional requirements (described in subchapter: 2.6.2) to be satisfied.

Fortunately, Kubernetes is a popular tool and many methods of deploying it are already described in the literature. The **available methods may be divided into three categories**:

- self-hosted solutions, on-premises,
- deployment in a cloud, but not using Managed Services,
- deployment in a cloud, using Managed Services.

Furthermore, different categorization may be applied. For instance, the deployment methods may be categorized **by the tools used**:

- using web interface of a particular cloud, e.g. AWS Management Console (supported by AWS),
- using command-line tools officially supported by a particular cloud, e.g. *awscli* or *eksctl* (supported by AWS),
- using command-line tools designed exactly to deploy a Kubernetes cluster, but not limited to one particular cloud, e.g. *kops*,
- using command-line tools, designed for managing computer infrastructure resources, e.g. Terraform, SaltStack.

There was an interesting study conducted by Cloud Native Computing Foundation (CNCF) in 2019. A question was asked about **what tools are used in the respondents' organization or company to manage containers**. According to CNCF's Cloud Native Landscape, there are more than 109 such tools and 89% of them are using different forms of

Kubernetes. The **top 10 tools** are presented in figure 3.1 [104]. Based on this chart, it is observable that **the tool used the most often is Amazon Elastic Kubernetes Service (AWS EKS)** and the second most popular tool is Google Kubernetes Engine (GKE).

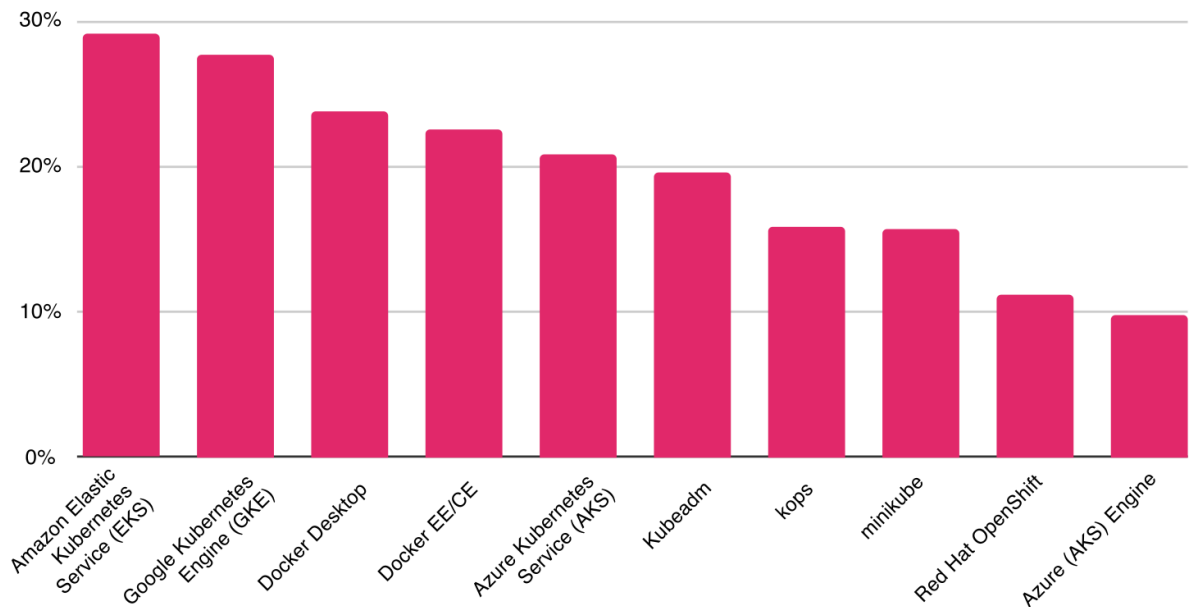


Fig. 3.1: Top 10 tools used by organizations and companies to manage containers [104]

Further in this chapter, the methods categories (mentioned before) are briefly explained and a few methods are described in more detail. The focus is on **the two particular methods, which are used in the practical part of this work:**

- deploying on AWS, using AWS Managed Service (AWS EKS), using *eksctl* which is a AWS supported official tool,
- deploying on AWS, not using any Managed Service, using *kops* which is a command-line tool, not officially supported by any cloud, but designed exactly to deploy a Kubernetes cluster.

3.1 Managed Services

This subchapter explains the term: Managed Services and lists some examples.

The complexity of Kubernetes infrastructure has a steep learning curve. Thus, a new market of services emerged **to free the Kubernetes users of the burden of having to configure and maintain the not trivial Kubernetes infrastructure**. They provide a version of Kubernetes which is hosted and managed by a cloud. They are called Managed Services [11].

Managed services **offer a ready to use cluster**. The **popular Managed Kubernetes Services**, offered by major cloud providers, are (Fig. 3.2):

- Elastic Kubernetes Service (*Amazon EKS*, also referred to as *AWS EKS*) [34]
- Google Kubernetes Engine (*GKE*) [133]
- Azure Kubernetes Service (*AKS*) [157]



Fig. 3.2: The logos of three Managed Kubernetes Services, offered by the major cloud providers

Managed Services are **deeply integrated with other resources offered by a particular cloud**. For example *AWS EKS* is claimed to integrate with AWS services like *Amazon CloudWatch*, *Auto Scaling Groups*, *AWS Identity and Access Management (IAM)*, *Amazon Virtual Private Cloud (VPC)*, and *AWS App Mesh* [34]. A nice feature of *AKS* is integrated continuous integration and continuous delivery (CI/CD) experience [157]. Moreover, *GKE* is advertised as offered with "integrated Cloud Monitoring with infrastructure, application, and Kubernetes-specific views" [133].

Managed Services are relatively new. For example, the initial release of *AWS EKS* happened at 05.06.2018 [57]. Thus, some literature may not acknowledge its existence, e.g. the book "Mastering Kubernetes" [14]. Another effect of the Managed Services being so young is that, to the best of this work's author knowledge, there is no other formal study which compares the two chosen methods of Kubernetes cluster deployment, from a perspective of satisfying production environment requirements. There was, however, an interesting study conducted which compares three Managed Kubernetes Services (the

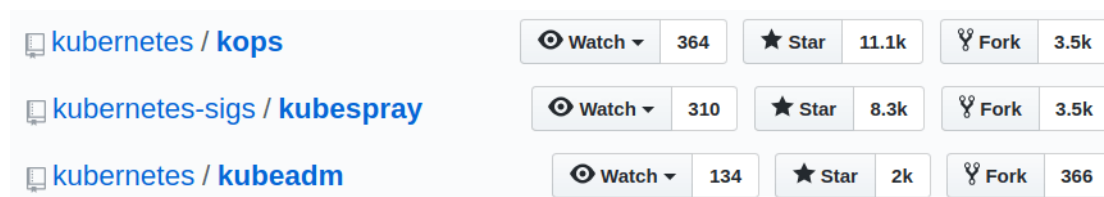
three mentioned above: AWS EKS, AKS and GKE) from a performance perspective [11].

3.2 Command-line tools designed exactly to deploy a Kubernetes cluster

In this subchapter, several opensource command-line tools are discussed. These tools are not supported by any particular cloud provider, but they are designed exactly to fulfill one aim: to deploy a Kubernetes cluster.

Github.com is a popular (used by many people) website which hosts code repositories. Many of those repositories are opensource. *Github.com* uses a star-rating system: whenever someone likes a particular project, one can give it a star. Thus, **the number of stars given to a project may be used as a popularity measure** of a particular project.

Searching through *Github.com*, there are several **command-line tools available which exist only to fulfill one aim: to deploy a Kubernetes cluster**. The names of a few of the most popular of these projects, together with the amount of stars they were acclaimed with, are presented in figure 3.3.

A screenshot of the GitHub website showing three repositories. Each repository entry includes a repository icon, the owner and repository name, a 'Watch' button with a dropdown arrow and a count, a 'Star' button with a star icon and a count, and a 'Fork' button with a fork icon and a count.













 kubernetes / kops	 Watch ▾ 364	 Star 11.1k	 Fork 3.5k
 kubernetes-sigs / kubespray	 Watch ▾ 310	 Star 8.3k	 Fork 3.5k
 kubernetes / kubeadm	 Watch ▾ 134	 Star 2k	 Fork 366

Fig. 3.3: Selected opensource tools which deploy a Kubernetes cluster, found on *Github.com*, state for date: 20.04.2020

The most popular tool amongst the four above is: **kops**. *Kops* stands for Kubernetes Operations and on its *Github.com* page it is advertised as "the easiest way to get a production grade Kubernetes cluster up and running" [85]. It is a command-line tool which allows to create, destroy, upgrade and maintain production-grade, highly available, Kubernetes clusters. It supports multiple clouds: AWS (officially), GCE and OpenStack (in beta support) and VMware vSphere (in alpha) [85]. Furthermore, it can be used from multiple operating systems: Linux, Mac and Windows [81]. *Kops* is one of the recom-

mended ways to setup a Kubernetes cluster and it is a tool which can be used to create a production environment [19].

The second most popular tool is **kubespray**. It also supports multiple clouds: AWS, GCE, Azure, OpenStack, vSphere, Oracle Cloud Infrastructure (Experimental), and Baremetal [94]. Furthermore, it also supports Highly Available deployment. The main difference between *kubespray* and *kops* is that **Kubespray uses Ansible** (an opensource tool to provision infrastructure) while ***kops* performs the provisioning and orchestration itself**. *Kops* also provides more features tightly integrated with specified clouds [93].

There is also **kubeadm**. In contrast to *kops* and *kubespray*, *kubeadm* helps to get a minimum viable cluster "in a user friendly way". Furthermore, its scope is limited to the local filesystem [116]. Kubernetes (and *kubeadm*) maintainers state that *kubeadm* is supposed to become a building block for all Kubernetes deployments. They also want to identify the common phases of a cluster deployment and make *kubeadm* an easy-to-use and configurable set of commands for each of those phases. An example of a common phase could be: *certificate distribution* [156].

Other than these three briefly described tools, **there are many more**, for instance: *kube-aws* [130] or *minikube* [95]. But neither listing them all nor describing them is needed for the purpose of this work. In the empirical part of this work, only one of the described tools is used: *kops*.

3.3 Custom solutions

In this subchapter, custom solutions for a Kubernetes cluster deployment are presented.

There is always a way to do (almost) everything on one's own. Here, one can split the deployment into two phases: infrastructure creation and, then, provisioning. To create infrastructure, meaning: virtual machines (compute resources), network and storage resources, one can use the following tool: **Terraform**. It is a tool made by Hashicorp, which incorporates **Infrastructure as Code (IaC)**, in order to manage infrastructure. It has a command line interface (CLI) and thus it is fairly easy to use with a Continuous Integration server. Thanks to declarative configuration files, the resultant infrastructure is easy

to reproduce [141]. IaC facilitates future repetition of any work done with Terraform and thus any evaluations of this tool (done by experimenting with it) may be repeated in the future.

Terraform supports many providers, meaning that it can manage infrastructure of different public and private clouds (like: AWS, GCP, Azure, OpenStack) and, it can also handle local operations [141]. There are also some alternative solutions, e.g. **Heat or CloudFormation**. They support declarative configuration files too. But they are not cloud-agnostic. Heat is a solution for OpenStack cloud and CloudFormation – for AWS. Sometimes, there is a need to use many various providers in order to build an infrastructure and then, one may view it as a nice feature, to be able to use a unified syntax, which Terraform provides [142].

After the infrastructure is created, the next step is to provision it. This step involves installing needed software (in this case: Kubernetes) and configuring it. However, since Kubernetes cluster is not just one machine, it would be tremendously tedious to provision each machine one by one manually. Thus, a chosen Configuration Management tool should be applied. There are any opensource tools available, some have been already presented in the subchapter 2.6.6.

There was a time, when **Kubernetes source code contained SaltStack code**, in order to provision a cluster. The code can be still found on *Github.com*: <https://github.com/kubernetes/kubernetes/tree/release-0.13/cluster/saltbase/salt>. However, now this method is deprecated (as can be read e.g. on this page: <https://github.com/kubernetes/kubernetes/tree/v1.18.2/cluster/>).

The advantage of such a custom solution is that it is very customizable and that, by making it work, one can learn Kubernetes in depth. On the other hand, it is not nearly as fast to design and first run as Managed Services. If one would like to immerse into the ocean of Kubernetes knowledge base, another great idea could be to try Kubernetes The Hard Way [144].

3.4 Deployment method: on AWS EKS Managed Service using *eksctl*

Here is briefly described the first of the two methods, which are used in the empirical part of this work.

The goal of this method is to have the Kubernetes control plane managed by AWS. This means that it is AWS which is responsible for managing Kubernetes master components, such as: API Server, kube-scheduler, kube-controller-manager and also Etcd. The **control plane should be already highly available**, thanks to being deployed across **multiple Availability Zones (AZ)** [69]. Availability Zone is an isolated geographically place to host Amazon data centers. Several AZs create a Region. AZs in a Region are connected through low-latency links [63].

The control plane being already highly available means that there should be at least two API server nodes and three etcd nodes that run across three Availability Zones within a Region. In addition, AWS EKS is responsible for automatically detecting and replacing unhealthy control plane instances. Furthermore, version upgrades should be provided automatically and applied [69]. There exists a **Service Level Agreement (SLA)** especially concerning AWS EKS. It is a policy which governs the use of the Amazon Elastic Container Service for Kubernetes. For example, the SLA states what is the Monthly Uptime Percentage during any monthly billing cycle (at least 99.95%) [41]. Thanks to such a SLA, AWS EKS is claimed to be **reliable and recommended for production workloads** [69].

As far as the master components are concerned, they are run in an account managed by AWS. The Kubernetes API is exposed via the *Amazon EKS* endpoint which is associated with the cluster. Each *Amazon EKS* cluster control plane runs on its own set of *Amazon EC2 instances* [36].

Furthermore, AWS EKS also manages the Kubernetes (worker) nodes. The nodes are run in one's AWS account and AWS connects them to the already deployed control plane via the cluster API server endpoint. The worker nodes are grouped in an *AWS EC2 Auto Scaling group*. The latter fact has some consequences, namely all these nodes have to [70]:

- use the same *Amazon EC2 instance type*,

- be instantiated from the same *Amazon EC2* image (called: *AMI*, an abbreviation from *Amazon Machine Image* [47]),
- use the same *Amazon EKS* worker node *IAM* role.

Fortunately, an *AWS EKS* cluster can contain multiple node groups, therefore there is a possibility to utilize nodes of various *AMIs* [70]. This is an important information, because a Kubernetes administrator may be interested in choosing a particular operating system for a node. There exist such *AMIs*, which are specifically designed for *EKS*. This *AMI* is built on top of *Amazon Linux 2* and has some essential tools installed and configured: *Docker*, *kubelet*, and the *AWS IAM Authenticator*. The way in which the *AMI* is built is coded and made opensource. Thanks to this solution, everyone can build their own *AMI*, basing on the opensource version [40]. There is even an option to have *Windows* worker nodes [60].

The cluster control plane is fronted by an **Elastic Load Balancing Network Load Balancer** and also all the networking is taken care of (elastic network interfaces are provisioned in *VPC subnets*) to provide connectivity from the control plane instances to the worker nodes. Thanks to that, the *AWS EKS* user can access the **Kubernetes API Server** [36].

In order to use *AWS EKS*, there are two ways supported:

- using *eksctl* CLI,
- using *AWS* Management Console and *AWS CLI*.

Both of them demand installing *AWS CLI*. In this work, **the method with *eksctl* CLI is applied**. *Eksctl* is officially the CLI for *AWS EKS*, endorsed by *AWS*, though it was launched by *WeaveWorks*. With *eksctl*, a simple command can be used to instantiate the whole cluster [100]:

```
$ eksctl create cluster
```

Listing 3.1: A command of *eksctl* CLI tool used to create a Kubernetes cluster

The cluster can be **configured with a *YAML* file and also by setting *eksctl* CLI flags**. The documentation describing how to do it, is presented on the official *eksctl* website [183]. There is also an example *YAML* configuration file attached on that website. This file is used to customize a cluster. It is also appended below.

```

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: basic-cluster
  region: eu-north-1

nodeGroups:
  - name: ng-1
    instanceType: m5.large
    desiredCapacity: 10
  - name: ng-2
    instanceType: m5.xlarge
    desiredCapacity: 2

```

Listing 3.2: An example YAML file used to customize a Kubernetes cluster created with *eksctl* CLI tool [183]

This example configuration file sets a name for the cluster (*basic-cluster*), chooses an Amazon Region in which all the AWS resources will be deployed (*eu-north-1*) and also configures the details of Kubernetes nodes. Many more options can be set.

After the cluster is created and after a connectivity with a Kubernetes cluster endpoint is established, it is now possible to deploy applications on top of the cluster. The figure 3.4 depicts the stages of working with AWS EKS.



Fig. 3.4: A schema presenting the stages of working with AWS EKS

Much more information on how to deploy AWS EKS Kubernetes cluster can be found in the Internet. The official websites of AWS EKS [69] and *eksctl* [183] are a great source of help. More examples of cluster configuration can be found on the official *eksctl* page on *Github.com* [181]. A sample Kubernetes use cases scenarios, for example: creating a CI pipeline to deploy a sample Kubernetes service, can be found in the Internet [56].

3.5 Deployment method: on AWS using *kops*

Here is briefly described the second of the two methods, which are used in the empirical part of this work.

Kops was already introduced in the subchapter: 3.2. Similarly to *eksctl*, *kops* can also create a *Highly Available Kubernetes cluster*. However it demands more work than *eksctl*. The commands which allow to create a cluster on AWS are given in listing 3.3 [14].

```
$ kops create cluster ${CLUSTER_NAME} --state \
"s3://${K8S_EXP_KOPS_S3_BUCKET}" --cloud=aws \
--zones=us-east-1c
$ kops update cluster ${CLUSTER_NAME} --state \
"s3://${K8S_EXP_KOPS_S3_BUCKET}" --yes
```

Listing 3.3: The commands of *kops* CLI tool used to create a Kubernetes cluster configuration and then to deploy the cluster

But, before these commands can be run, one has to provide some minimal DNS configuration via Route53 (an AWS resource responsible for networking), set up a *S3 bucket* (another AWS resource, responsible for storage) to store the cluster configuration [14] and also configure the AWS IAM user (an AWS resource responsible for access management) and create a SSH key pair [76]. Amazon S3 is an AWS service which provides object storage. AWS IAM is an abbreviation from *AWS Identity and Access Management* and it is responsible for managing access to AWS services and resources. The instructions explaining how to set up the AWS resources are provided on the official *kops* website [76].

The configuration of *kops* is kept in an S3 bucket. In order to change the configuration, the command shown in listing 3.4 has to be used [76].

```
$ kops edit cluster ${NAME}
```

Listing 3.4: A command of *kops* CLI tool used to edit a Kubernetes cluster configuration

A configuration specification file is generated during the create phase and uploaded to a S3 bucket. The configuration can be also kept in YAML file. All the configuration options are available on a *kops* documentation website [83]. A cluster can be created using the command provided in listing 3.5 [91]:

```
$ kops create -f $NAME.yaml
```

Listing 3.5: A command of *kops* CLI tool used to create a Kubernetes cluster using a YAML configuration file

The stages of working with a Kubernetes cluster deployed on AWS with *kops* are similar to the stages of working with *eksctl*, but there is the additional first stage (fig. 3.5).

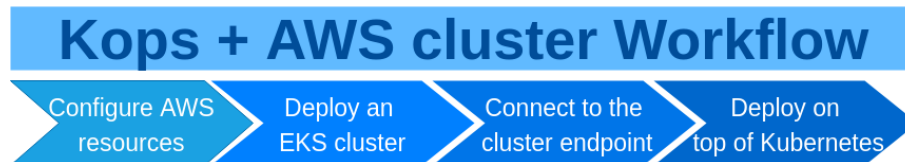


Fig. 3.5: Stages of working with a Kubernetes cluster deployed on AWS with *kops*

The next stage, after instantiating a cluster is to connect to it through an endpoint. Such configuration is automatically generated and written to `/.kube/config` (on Linux Operating System) [76]. *eksctl* also automatically generates this configuration file.

Kops has been around a long time as an AWS-specific tool, but now it also supports other clouds [8, 85]. Its main features such as: automated Kubernetes cluster deployment, highly available master or adding a variety of custom Kubernetes addons [89] indicate that *kops* is an attractive tool, worthy to at least try out.

There are many Internet sources available to broaden one's knowledge about *kops*: the official *kops* website [84], the *kops* project website on *Github.com* [85] and there are also some comparisons of *kops* versus alternative tools available, e.g. "A Multitude of Kubernetes Deployment Tools: *kubespray*, *kops*, and *kubeadm*" [178].

3.6 Amazon services allowing to run containers

In order to exhaust the topic of the Amazon services which allow to run containers, this subchapter was created. This chapter shortly characterizes each such AWS Service.

All the AWS Services, used to manage containers, are as follows:

- ECS (Elastic Container Service),

- ECR (Elastic Container Registry),
- Fargate,
- AWS EKS (Elastic Kubernetes Service).

AWS ECS is a service that provides a **highly secure, reliable, and scalable way to run containers**. Secure, because the containers are run in a custom VPC (*Virtual Private Cloud*). Thanks to that, custom firewall rules can be applied to the containers (by using AWS Security Groups). Furthermore, each of the containers uses IAM, which means that access to each container can be restricted and granular access permissions can be assigned. Running containers on ECS is reliable thanks to the SLA, which guarantees a Monthly Uptime Percentage of at least 99.99% [44].

AWS ECR is not a service which strictly manages the containers, but it manages images. It is a **fully-managed Docker container registry**. AWS ECR eliminates the need to run such a registry on-premises or worry about scaling the underlying infrastructure. The images are hosted in a highly available and scalable architecture. Similarly to AWS ECS, AWS ECR also integrates with IAM [43].

AWS Fargate is a serverless service to run containers. This service is responsible for allocating the right amount of compute, eliminating the need to choose instances and scale cluster capacity. It can work with both: *AWS ECS* and *AWS EKS*. *AWS Fargate* runs each task or pod in its own kernel, therefore the tasks and pods are provided with their own isolated compute environment. Without Fargate, the containers are run on *EC2 instances*, so the end user has to pay for both: an *EC2 instance* and for an *ECS container* [53]. Running serverless Kubernetes Pods Using *AWS EKS* and *AWS Fargate* is very new, a blog post informing about this possibility was created at 3rd of December 2019 [65].

The last service on the list above, **AWS EKS**, was already described in this chapter.

3.7 Summary

There are **many methods of deploying a Kubernetes cluster**. According to the authors of "Cloud Native DevOps with Kubernetes" [8], the best solution is to use **Managed Services**. The authors argue that, thanks to this method, one can get a fully working, secure, highly available, production-grade cluster in a few minutes and for a little price. Man-

aged Services are certainly a good way to try Kubernetes out. Then, if one wants to do something non-standard or to experiment with the cluster, then one could choose **a custom or a hybrid solution**. The self-hosted, deployed on-premises way is recommended if the following qualities are of a great importance: price, low-network latency, regulatory requirements and total control over hardware [14].

Managed Services offer many features like: built-in autoscaling, security, high availability, having serverless options. However, they may be more expensive (for example, when using *AWS EKS*, one has to pay \$0.10 per hour for each *Amazon EKS* cluster [38]) and less customizable. Custom solutions allow the Kubernetes administrator to broaden their knowledge and **grasp the deep understanding** on what is going on under the Kubernetes hood (i.e. one can customize how the Kubernetes control plane is deployed or set up a custom network topology).

There is **no one right answer** which fits all the use cases. It is always advised to do one's own research and to try and experiment with the existing methods.

4 Preparations for production deployment of Kubernetes cluster

This is a practical chapter. It includes planning and designing the production deployment, considering: capacity planning, choosing which requirements to satisfy and taking any other deployment and infrastructure related decisions. No AWS resources were created here, but AWS CLI was used to get some information.

4.1 AWS Region

The first decision to take is to choose the AWS Region in which all the resources should be deployed.

The Kubernetes cluster deployment constitutes the empirical work of this study. The deployment will target the AWS cloud. This cloud provides many data centers, spread across multiple physical locations around the world. AWS calls these locations: Regions, Availability Zones, and Local Zones. AWS offers many resources (for example: S3, EC2 or EKS). Some of them are global, whereas the others are tied to a Region, an Availability Zone, or a Local Zone [63].

While it matters not which Availability Zone will be chosen, **choosing a Region has some consequences**. For example, AMIs are tied to a Region. AMIs may be copied between Regions. It may happen, that some officially supported AMIs are available only in a limited number of Regions and choosing some other Region would incur more **time and money** needed to copy the image. There is a charge for data transfer between Regions [63].

AWS Local Zones are a new type of AWS infrastructure deployment. Thanks to them, AWS resources can be put closer to large population, industry, and IT centers where no AWS Region exists today. AWS Local Zones help to run latency sensitive applications closer to end users. Even single-digit millisecond latency can be achieved. The use cases for such latency are for example: media entertainment content creation, real-time gam-

ing, reservoir simulations, electronic design automation, and machine learning [54]. AWS Local Zones are available since 9th March 2020 and there is currently one AWS Local Zone, available in Los Angeles, California [50]. For this work, single-digit millisecond latency is not needed.

In this work, *AWS EC2 instance type* will be used (Elastic Compute Cloud resources). The *AWS CLI* command presented in listing 4.1 was run to list all the available regions for the AWS account of the author of this work.

```
$ aws ec2 describe--regions
```

Listing 4.1: A command of *AWS CLI* tool used to list all the available regions (for an AWS account)

The returned list of Regions was: "eu-north-1", "eu-west-3", "eu-west-2", "eu-west-1", "eu-central-1", "ap-south-1", "ap-northeast-2", "ap-northeast-1", "sa-east-1", "ca-central-1", "ap-southeast-1", "ap-southeast-2", "us-east-1", "us-east-2", "us-west-1", "us-west-2". Considering the geographical proximity, the Region should be in Europe. The Regions' names are mapped to geographical locations below (limited to Europe) [64]:

- "eu-north-1" – Europe (Stockholm),
- "eu-west-1" – Europe (Ireland),
- "eu-west-2" – Europe (London),
- "eu-west-3" – Europe (Paris),
- "eu-central-1" – Europe (Frankfurt),
- "eu-south-1" – Europe (Milan).

The next criterion of choosing the Region is **price**. The pricing for *EC2 instance types* were compared for 5 AWS Regions (available for a particular AWS account, in Europe). A few *EC2 instance types* were considered, but from the General Purpose family of types. Only Linux/UNIX usage was taken into consideration. The comparison is presented in the table 4.1. The data comes from the official AWS EC2 pricing website [33]. The prices are noted in **USD per hour of an EC2 instance running**.

Table 4.1: Comparison of the price, noted in USD, of 1 hour running of an *EC2 instance* on AWS, considering a few selected AWS Regions. Based on the official AWS EC2 pricing website [33]

Instance Type	London	Ireland	Frankfurt	Paris	Stockholm
t2.nano	0.0066	0.0063	0.0067	0.0066	not given
t2.micro	0.0132	0.0126	0.0134	0.0132	not given
t2.small	0.026	0.025	0.0268	0.0264	not given
t2.medium	0.052	0.05	0.0536	0.0528	not given

Basing on the three factors (Region being available for an AWS account, in Europe and the cheapest) - **the Region chosen is: Europe (Ireland) - "eu-west-1"**. Then, the next thing to decide upon was choosing availability zones. All the AZs available in the chosen region can be itemized with the command shown in listing 4.2 [76]:

```
$ aws ec2 describe-availability-zones --region eu-west-1
```

Listing 4.2: A command of *AWS CLI* tool used to list all the available AZs in the chosen AWS Region)

The command output listed the following AZs: "eu-west-1a", "eu-west-1b", "eu-west-1c".

4.2 Version of Kubernetes

In this subchapter, it is decided which Kubernetes version to use.

4.2.1 Why it is important to choose a particular version?

In order to compare two methods of Kubernetes deployment in a reasonable way, **both methods should deploy the same Kubernetes version**. Also, the version should be one of the latest released ones, for the sake of keeping this comparison up-to-date. Furthermore, using a specified version is important for the production environment, because:

- the environment should be easy to recreate,

- the experiment (the empirical work of this study: deploying a Kubernetes cluster) should be possible to be repeated,
- using specified versions helps to incorporate Infrastructure as Code and DevOps best practices and to automate the deployments.

4.2.2 Which version was chosen?

At the time of writing this work, in **May 2020**, the latest released versions of the needed software are:

- Kubernetes version 1.18 [118].
- *Kops* version 1.16.2, Kubernetes versions supported by *kops* are: 1.16.9, 1.15.11, 1.14.10, 1.13.12, 1.12.10 [82, 86, 87].
- Kubernetes versions supported by EKS: 1.16.8, 1.15.11, 1.14.9, 1.13.12. However, Kubernetes version 1.13 is deprecated [39]. There are also particular EKS Platform versions for each supported Kubernetes version. For example, for Kubernetes 1.16.8 there is one EKS Platform version: eks.1 [62].
- *Eksctl* version 0.20.0 [182].

EKS documentation recommends that unless a specific Kubernetes version is required, the latest supported version should be chosen [39]. **The maximum version of Kubernetes, supported by *kops* is 1.16.9 and by *eksctl*: 1.16.8.** When it comes to the AWS AMIs, there are some EKS-optimized Linux AMIs – built for this exact purpose – to be deployed in a Kubernetes cluster. They are built on top of **Amazon Linux 2** [40]. Besides the official Amazon EKS-optimized AMIs, Canonical (the commercial support provider for Ubuntu) has partnered with Amazon EKS to create worker node AMIs built on top of **Ubuntu** [66]. On the other hand, the default Operating System for *kops* is Debian whereas Amazon Linux 2 support is still experimental, but should work [80].

Amazon Linux 2 is a Linux Server Operating System, provided by Amazon. Among the features it provides are: optimized performance (to help ease integration with AWS Services), long term support, bleeding edge software updates support, on-premises use, systemd support, etc. [46]. However, Amazon Linux 2 is a RPM-based Linux distribution (which can be judged based on information that many applications developed on Cen-

tOS (and similar distributions) run on Amazon Linux) [45]. RPM-based Linux distributions include: CentOS, Fedora, Red Hat Linux, etc.

Considering all the information presented in this subchapter, the **version of Kubernetes chosen was: 1.16.9 for *kops* and 1.16.8 for *eksctl*.**

4.3 Tools and development environment

This subchapter can be treated as a checklist of what tools should be gathered before deploying a Kubernetes cluster. It also characterizes a development environment.

The following tools were selected and deemed essential to deploy a Kubernetes cluster:

- Linux,
- *Bash*,
- *kubectl*,
- AWS CLI 2,
- *eksctl*,
- *kops*,
- *Helm*,
- *Bats-core*.

When it comes to **development environment – the Ubuntu 18.04 workstation** was used. In order to facilitate repetition of the empirical work, all the deployment was performed in a Docker container. Thus, the local development environment was just obliged to: have Docker installed, Internet access and AWS credentials configured.

The **Docker container** used, was created from a Docker image: *kudulab/k8s-aws-dojo* [128]. This Docker image already has all the needed tools installed. In order to run this Docker container, *Dojo* was used. *Dojo* is a CLI program, designed to provide portable development environments [129]. Thanks to using *Dojo* and Docker, one can ensure that the same development environment is present on their laptop and also on a CI server.

Bash is a shell language, available on Linux distributions like Ubuntu or Debian. *kubectl* is a CLI program needed to access and manage a Kubernetes cluster [96]. AWS CLI allows

to communicate directly with AWS. *Eksctl* is a CLI program that facilitates Kubernetes deployment on AWS EKS [183]. *Kops* is also a CLI program that facilitates Kubernetes deployment on AWS (but does not use AWS EKS) [84]. *Helm* is a package manager for applications which are deployed on top of Kubernetes [71]. *Bats-core* is a CLI program, used to run tests using *Bash* shell [172].

4.4 Selected requirements of a production deployment and acceptance criteria needed to satisfy them

Here a set of production deployment requirements is presented. Each of them is shortly described. Acceptance criteria, needed to satisfy the requirements are provided. This subchapter helps to plan the deployment.

There are numerous requirements for a production deployment of a Kubernetes cluster. Some of them were gathered throughout the available literature and presented in the subchapter: 2.6.2. It is common knowledge that companies, which deploy Kubernetes and similar systems, obey some set of best practices, dedicated to these companies only. Thus, the requirements presented in this work do not exhaust the topic.

Below, there is a list of the selected production deployment requirements. The most important one were selected (but it was a subjective opinion of the author of this work). In the empirical part of this study, the requirements were attempted to be satisfied.

- healthy cluster,
- automated operations,
- central logging,
- central monitoring,
- central audit,
- backup,
- high availability,
- autoscaling,
- security.

The set entails **9 requirements**. Each of them was already described theoretically, but

further in this chapter, each requirement is explained from a practical point of view. The **acceptance criteria** needed to satisfy each of the requirements are presented. Basing on this, a plan of a Kubernetes cluster deployment is created. The work done is described in the next chapter, together with any problems encountered and troubleshooting sessions.

4.4.1 Healthy and usable cluster

Unless any of the Kubernetes control plane components is healthy, it will need to be fixed. It should not be a problem with a managed Kubernetes services, but for self hosted clusters, this should be checked. It should be noted that, since the **health checks** ought to be run frequently, they shouldn't do anything too expensive. To monitor the health of a Kubernetes cluster, **the following could be checked: number of nodes, node health status, number of pods per node and overall, resource usage/allocation per node, and overall** [8].

When all the cluster health checks are passed, then the cluster is healthy. But, the cluster should be also usable for end users. End users will want to deploy applications on top of the cluster. Thus, they have to have the possibility to reach the cluster endpoint, which in practice means, that they have to **be able to reach the Kubernetes API Server endpoint**. Such an endpoint may be an IP address or a domain name. Since the cluster will be deployed in a production environment, a domain name is preferred. Furthermore, many sources claim that a top-level domain or a subdomain is required to create a Kubernetes cluster with *kops*. Alternatively, gossip-based DNS can be used instead, and then, the cluster domain name would have to end with *.k8s.local* and the API server would be available under the AWS load balancer address [76, 77, 107, 139].

In order to verify that an application can be deployed on top of a Kubernetes cluster, a simple Apache server will be deployed. A *Helm* Chart will be used [99]. Then, *Bats-core* tests will be run to check that the Apache server works (more precisely: that the Apache endpoint is reachable from a remote machine).

To summarize, in order for a cluster to be *healthy and usable*, the following acceptance criteria must be met:

- the cluster should be tested with *Bats-core* (i.e particular number of worker nodes

- should be deployed, particular Kubernetes version should be used),
- the Kubernetes API Server endpoint should be reachable for the end users under a domain name (i.e. an end user can connect with the cluster using *kubectl*),
- it should be tested that an application can be deployed on top of a Kubernetes cluster (i.e. an Apache server will be deployed and tested).

4.4.2 Automated Operations

In order to be able to perform automated operations, **a Kubernetes cluster configuration and commands needed to manage the cluster should be stored in a version control system**. This is the goal of Infrastructure as Code. Such a solution will also allow cluster changes to be applied through reviewable commits and in turn it will allow to avoid possible collisions from simultaneous changes being made. Moreover, a history of the version controlled source code may act as an audit trail [74, 92]. Both: *eksctl* and *kops* support **cluster configuration through YAML files**. In this work, all the code needed to create a Kubernetes cluster will be stored in **a Git repository**.

Furthermore, **to automate the operations, a Bash script will be used**. It should be possible to use this script in order to: create, test and delete a cluster. This *Bash* script will be named *tasks*. There will be one such script provided for *kops* cluster and one for *eksctl* cluster. It will be possible to run the commands presented in Listing 4.3.

```
$ ./tasks _create  
$ ./tasks _test  
$ ./tasks _delete
```

Listing 4.3: Commands provided by tasks file – a *Bash* script which automates a Kubernetes cluster operations

It is essential to obey the Infrastructure as Code policy, because it facilitates repeating the operations and the whole experiment of a cluster deployment. Although, in this work no CI server is going to be used, **it should be possible to invoke the automated commands (by using the *Bash* script) in a potential CI pipeline**.

Moreover, it should be also possible to **differentiate between two deployment environments: testing and production**. The testing environment should be very similar to the production environment. But, there may be more tests run in the testing environment

and more resources used in the production environment. Another thing to note is, due to the automation requirement, there should be no need to perform any operations using AWS Management Console, but it is desirable to view any potential dashboards (e.g. with metrics) through a web browser.

It is recommended to set up a **package manager for reusable deployments** which happen on top of Kubernetes [138]. In this work *Helm* will be used and it was already described in the subchapter: tools. Applications deployments which happen on top of Kubernetes are already facilitated thanks to many resources available by Kubernetes API, such as: Deployment [115] or DaemonSet [114], etc.

Furthermore, it has to be possible to create a cluster in a testing or production environment. Thus, some kind of templating mechanism is needed. Both *eksctl* and *kops* use YAML configuration files. Whenever a YAML file template will be used and it will contain *Bash* variables, the file name will end with *.tpl.yaml*. But, when a file will be a YAML file template generated by *kops*, the file name will end with *.kops-tmpl.yaml*

To summarize, in order to satisfy the Automated Operations requirement, the following measures will be applied:

- all the code and configuration needed to deploy a cluster will be stored in a Git repository,
- cluster will be configured with YAML file,
- *Bash* script will be used to create, test and delete a cluster,
- *Helm* will be used to automate a test application deployment,
- it will be possible to choose between two deployment environments: testing and production (a templating mechanism with *Bash* variables will be used).

4.4.3 Central Logging

The plan here is to simply **use AWS CloudWatch service**. It should be easy to enable when using *eksctl*. It is not enabled by default due to data ingestion and storage costs [179]. There should be also a possibility to enable logging to AWS *CloudWatch* when using *kops*.

4.4.4 Central Monitoring

To satisfy the Central Monitoring requirement, it is planned to use **AWS CloudWatch**. There also exist a solution – a program called: Kubernetes Metrics Server. And even though it is a server which provides source of container resource metrics, it should not be used as an accurate source of resource usage metrics. Its main usage is for CPU/Memory based horizontal autoscaling and for automatically adjusting/suggesting resources needed by containers. Thus, it will not be used for Central Monitoring. But one should be aware of its existence [121].

4.4.5 Central Audit

This requirement will be simply met by utilizing **the AWS CloudTrail service**. This service should show us, which user is responsible for what changes made to the AWS resources used. Besides, *AWS CloudTrail* can be also used to detect unusual activity in the AWS accounts [52].

4.4.6 Backup

In order to implement and test backup, **Velero will be used**. First, *Velero* will be used to create a backup. Then a disaster will be simulated: a whole Kubernetes namespace will be deleted with all the resources deployed inside. The last phase will be the backup restore phase. It is expected that the namespace and all the resources in it will be restored [55, 97].

Let's now apply a broader perspective, to understand **why backing up a Kubernetes cluster is needed**. Since a cluster configuration will be kept in a Git repository (in *YAML* files) and all the commands needed to operate the cluster (create, test, delete) will be automated, then it should be possible to recreate the cluster using the mentioned Git repository. There are even opinions which claim that if you are asking how to backup a Kubernetes cluster (like EKS), you are using it wrong [143]. However, it should be remembered that the Git repository needed to operate the Kubernetes cluster (using Infrastructure as code) contains only the configuration and operations. It does not contain the cluster state. The cluster state is represented by deployed containerized applications and work-

loads, their associated network and disk resources, and other information [120]. Thus, anytime one deploys an application on top of the cluster, the cluster's state changes. All the cluster state is stored in Etcd and even the official Kubernetes documentation recommends to have a back up plan for those data [119].

The other thing is **backing up all the applications deployed on top of Kubernetes**. Their deployment should also be automated (e.g. with *Helm* charts), but this automation is also not sufficient in terms of backup. In order to backup such applications, one may have to backup both: their configuration and data. The data may be kept on various volumes (e.g. AWS Elastic Block Store). This problem is, however, out of scope of this work, but one should be aware that it should be handled as well.

4.4.7 Capacity planning and High Availability

The needed capacity, meaning: number of machines and IT resources (CPU, Memory) must be discussed. The important thing is, that the to-be-deployed Kubernetes cluster, should have the minimal amount of capacity, but still it should be representative for a production environment. Here **chicken-counting** (0, 1, many) comes to the rescue. Applying the chicken-counting technique means that if a production site has 250 web servers, 2 should be enough to represent it [28]. Since a Kubernetes cluster consists of a master node and worker nodes, then it was decided that **a representative cluster would be made of two worker nodes**. One worker node will be created together with cluster creation and the second worker will be added automatically by the Autoscaler. When it comes to number of master nodes needed, one is enough for the minimal deployment. But in order to achieve high availability at least two master nodes are needed and they should span multiple AZs [27]. However, having an odd number is recommended for master nodes to avoid issues with consensus/quorum. Thus, **at least three master nodes are needed** [78, 174].

The capacity of a cluster means the sum of CPU and memory of all the cluster worker nodes. Multiple ways are possible to achieve the desired target capacity. Example given: if a cluster with a total capacity of 8 CPU cores and 32 GB of RAM is needed, then it can be achieved by, for example, having two nodes (each with 4 CPU and 16 GB RAM) or having four nodes (each with 2 CPU and 8 GB RAM) [188]. Now there are two questions:

- It is better to have a few bigger nodes or many smaller nodes?
- What is the minimal capacity needed for a production cluster?

Table 4.2 summarizes the advantages and disadvantages of having a few bigger nodes. More information about the pros and cons are presented in "Architecting Kubernetes clusters — choosing a worker node size" [188].

Table 4.2: Comparison advantages and disadvantages of having a few bigger nodes in a Kubernetes cluster instead of having many smaller nodes [188]

Advantages	Disadvantages
Less management overhead (less number of machines to manage)	Large number of pods per node (each pod introduces some overhead, if there are too many pods, the system may slow down)
Resource-hungry applications allowed	Large scaling increments
Potential lower costs per node (the price increase for a more powerful may not be linear)	Limited replication (each replica of a pod may be requested to be deployed on a different node. If a HA application requires 5 replicas, but there are only 2 nodes in a cluster, then the effective degree of replication is reduced to 2)
Large numbers of nodes can be a challenge for the Kubernetes control plane (there are many network communication paths and also also more load on the etcd database)	The impact of one failed node is big
Better resources utilization	—

However, in the Internet, there is contradictory information about the limited replication. The online source cited above [188] states that there can be maximally as many replicas as worker nodes in a cluster, while another source – a blog post [165] claims that there can be more pod replicas than nodes. This information was not verified in practice in this work. Perhaps the replicas limit depends on Kubernetes version – but this is just a speculation of this work's author.

It is officially claimed, that Kubernetes supports up to 5000 nodes and up to 100

Pods per node [110, 188]. However, in practice, 500 nodes may already lead to non-trivial challenges. Furthermore, as far as AWS EKS is concerned, there are some hard limits for number of pods allowed for a particular EC2 instance type. For example, for a t2.medium instance, the maximum number of pods is 17, for t2.small it's 11, and for both: t2.micro and t2.nano it's 4 [59, 188].

There was no official statement found stating the minimal capacity of a Kubernetes cluster. Thus, in the empirical part, the work will be started from using the smallest (and cheapest EC2 instance type). Should there be any problems, instance types with more resources will be used. Generally, the EC2 resources type should depend on the total quantity of resources needed by pods. (Some pods will be deployed in order to test whether the cluster is healthy). Some overhead must be also considered, because the amount of compute resources that are available for pods, **the allocatable**, is smaller than the capacity of the whole node [122].

To summarize: the production cluster should have at least three master nodes and two worker nodes. The experiments will start from the smallest (and cheapest EC2 instance type) available.

4.4.8 Autoscaling

The idea here is to test whether a cluster, which consists of master nodes and worker nodes, will be able to, by itself, create and delete another worker node. This decision, whether to scale in or out, should be taken according to how many resources there are available for pods. The demand for more resources will be artificially created by increasing the value of *replicaCount* in the Apache Helm Chart [99].

4.4.9 Security

There are many security measures that could be applied on a cluster. In order to stay secure, access to the Kubernetes cluster should be limited. Any Kubernetes deployment on AWS will have to deal with networking.

VPC stands for Virtual Private Cloud and it is an AWS resource, providing the networking services. When a VPC is created, it means that a virtual private network is created and

assigned to a particular AWS account. AWS resources can be launched into that VPC. Each VPC has a CIDR block assigned. The VPC cannot span more than one AWS Region. The VPC can be divided into subnetworks (or subnets). One subnet cannot span more than one Availability Zone. **Subnets can be either public or private.** If a subnet's traffic is routed to an internet gateway (which is an AWS resource), the subnet is known as a public subnet. Otherwise, a subnet is private [68]. This means that public subnets have access to the Internet.

By default, both *kops* and *eksctl* deploy Kubernetes clusters using public subnets. This means, that anyone who knows the URL of the Kubernetes master node, can access its API Server. Additionally, if one has a private SSH key, then they can SSH login into the master node. A simple way to restrict access to the cluster is to use **Security Groups**. Security Groups rules will be used to limit SSH access to the master and worker nodes and also to limit access to API Server endpoint [75]. In result, only one IP address should be able to access the endpoints.

The deployments described in this work are operated by one person only. Thus, the IP address of the author's laptop will be used. However, when working in a bigger team of many Kubernetes cluster administrators, one could use a NAT Gateway or NAT instance IP address as a single allowed IP address. Furthermore, a bastion host could be used to securely SSH login into the master node[61].

To summarize: in order to make the cluster more secure, access to the cluster should be restricted. Only 1 IP address will be allowed to connect.

4.5 Expected cost

The cost of running the Kubernetes cluster is expected to be low. The clusters will be run for a short while - just to run some tests and then deleted. This should never be longer than 0.5 hour. Experiments may take longer, around several hours. However, the AWS resources which will be used are cheap and their cost depends on how for long they will exist. All the AWS resources which will be used are: *S3 buckets*, *EC3 instances*, *VPC subnets*, *Security Groups*, *Internet Gateway*, *EBS volumes* (attached to *EC2 instances*), *Autoscaling groups*, *Keypairs*, *Load balancers*.

The chosen approach was to read about AWS resources pricing and more or less estimate how much a single experiment would cost. Since it was a small number (less than 1 USD), it was decided to use Cost Allocation Tags [67]. One custom tag will be set to distinguish a Kubernetes cluster deployment from other AWS resources.

4.6 Summary

This chapter was needed to gather all the requirements and to establish acceptance criteria for each of the requirements. Apart from that the AWS Region was set to *eu-west-1* and the Kubernetes version was chosen to be 1.16.9 for *kops* and 1.16.8 for *eksctl*. The chapter also attempted to state how many master and worker nodes are needed. There should be 3 master and 2 worker nodes deployed.

Another topic touched in this chapter were the development tools. They were all listed here. It was decided that a Docker image will be used with all the development tools already installed. This solution was chosen, because it was easy to recreate such an environment (such a Docker container) and it helped with automating the deployment.

5 Deployment of Kubernetes cluster, using the two selected methods

This is a practical chapter and it was written together with performing the empirical work and writing the source code. Main steps of Kubernetes cluster deployment will be described. Also, encountered problems will be listed and potential solutions will be presented.

5.1 Setup of a local environment

Here the setup, common to all the deployments performed in this work, is explained.

5.1.1 Environment variables

For each deployment the following environment variables were set:

- `K8S_EXP_REGION` to choose an AWS Region. It was always set to: `eu-west-1`.
- `K8S_EXP_ENVIRONMENT` to choose a deployment environment. In this work two such environments were supported: `testing` and `production`.
- `K8S_EXP_CLUSTER_NAME` to choose a Kubernetes cluster name. For AWS EKS cluster this was a combination of the prefix: either `eks-` and with the value of the variable `K8S_EXP_ENVIRONMENT`. Example value: `eks-testing`. For `kops`: this was set to:

`${K8S_EXP_ENVIRONMENT}.${K8S_EXP_KOPS_S3_BUCKET}`

(this was needed by `gossip-dns`, a chosen networking solution).

All these environment variables start with the same prefix: `K8S_EXP_`, therefore it is easier to differentiate them from all the other environment variables.

5.1.2 Source code

Source code is publicly available on: <https://github.com/xmik/MastersThesis>. The code contains the text of this thesis and all the files needed to deploy Kubernetes clusters. The source code is kept as a Git repository and it contains the following directories:

- `figures` contains the images used in this thesis,
- `presentation` contains the files needed for presentation for the Final Project Seminar subject,
- `conspect-tex` contains the files needed for drafting the thesis topic, scope and aims,
- `thesis-tex` contains this thesis text,
- `src` contains the source code needed to deploy Kubernetes clusters,
- `src/eks` contains the files needed to deploy a Kubernetes cluster on AWS EKS,
- `src/kops` contains the files needed to deploy a Kubernetes cluster using *kops*.

In the `src` there is a file called: *Dojofile*. It was used in order to provide a development environment to operate Kubernetes clusters. The file is needed by the program: Dojo, described in the previous chapter. Another *Dojofile* is kept in the root directory of the source code. It was used to generate a PDF file with this thesis text.

Throughout this work, such a convention was applied, that each command typed into a *Bash* terminal starts with "\$" sign, output from a command is not prepended with any special sign, and whenever there is a word written before the dollar sign – it indicates a current directory which was set when running a command. Whenever a command starts with hash "#", it means that it is a text comment. Last but not least, backslash "\" indicates that a command was split to a new line. An example is presented in listing 5.1.

```
# this is a comment
kops$ ./this-is-a-command-run-from-kops-directory
this is output
kops$ ./this-is-a-long-command-so-long-that-it-was-split \
    —some-option —some-other-option
```

Listing 5.1: An example code listing explaining the format

5.2 Experimental deployments using *kops*

In this subchapter first experiments of deploying a Kubernetes cluster with kops on AWS are described.

5.2.1 Deployment without prerequisite steps done

Although the aim of this work is to create a production Kubernetes cluster, it is always welcome, when there is a possibility to start working with a program easily. It is nice to have a simple working proof of concept (POC). Thus, it was decided to start with *kops* without performing any prerequisite steps (listing 5.2).

```
$ kops create cluster --state \
  "s3://dummy-k8s-kops-state-store"
--master-zones=eu-west-1a --master-count=1 \
--master-size=t2.nano \
--zones=eu-west-1a --node-count=1 \
--node-size=t2.nano dummy-k8s-kops.k8s.local
```

Listing 5.2: Command used to create a cluster with *kops*, without prerequisite steps performed

The command *kops create cluster* instructs *kops* how to create a cluster. The flag `--master-count=1` says that there will be one master node created, `--master-size=t2.nano` sets the *EC2 instance type* and `--master-zones=eu-west-1a` configures the AZs in which master nodes will be deployed. Similar flags are used to configure worker nodes. The `--state` flag sets which S3 bucket to use. **As expected and what is aligned with the *kops* documentation [76], the above commands failed**, because the S3 bucket was not created (listing 5.3).

```
error reading cluster configuration "dummy-k8s-kops.k8s.local":
error reading s3://dummy-k8s-kops-state-store/dummy-k8s-kops.k8s.local/config:
Could not retrieve location for AWS bucket dummy-k8s-kops-state-store
```

Listing 5.3: Output of the commands used to create a cluster with *kops*, without prerequisite steps performed

The *kops* documentation [76] informs that the following goals should be first accomplished, before deploying a Kubernetes cluster:

- AWS CLI tools should be installed,
- AWS credentials should be set,
- AWS IAM user and its permissions should be set,
- DNS should be configured,

- An S3 bucket should be created for storing a cluster state,
- AWS Region and Availability Zones should be chosen.

Further in this chapter, after all the prerequisites will have been met, *kops* will be used to create a production Kubernetes cluster.

5.2.2 Deployment with prerequisite steps done – first working cluster

First, all the prerequisites were met. In order to make things simpler, an AWS user with administrator permissions was used. SSH keypair was already set. Then, it was decided to use the gossip-based DNS. Then, an S3 bucket was created to keep the *kops* cluster configuration. Both versioning and server side encryption of the S3 bucket were enabled. Versioning was strongly recommended, because thanks to it, one may revert or recover a previous cluster state store. S3 bucket encryption is not required, but may be needed for compliance reasons [76]. **Setting the S3 bucket** was done by the commands presented in listing 5.4:

```
$ export K8S_EXP_REGION="eu-west-1"
$ export K8S_EXP_KOPS_S3_BUCKET=\
  "k8s-kops-for-masters-thesis.k8s.local"
$ export K8S_EXP_ENVIRONMENT="testing"
$ export K8S_EXP_CLUSTER_NAME=\
  "${K8S_EXP_ENVIRONMENT}.${K8S_EXP_KOPS_S3_BUCKET}"
$ aws s3api create-bucket --bucket ${K8S_EXP_KOPS_S3_BUCKET} \
  --region ${K8S_EXP_REGION} \
  --create-bucket-configuration \
    LocationConstraint=${K8S_EXP_REGION}
$ aws s3api put-bucket-versioning \
  --bucket ${K8S_EXP_KOPS_S3_BUCKET} \
  --versioning-configuration Status=Enabled
$ aws s3api put-bucket-encryption \
  --bucket ${K8S_EXP_KOPS_S3_BUCKET} \
  --server-side-encryption-configuration \
    '{"Rules":[{"ApplyServerSideEncryptionByDefault":\
      {"SSEAlgorithm":"AES256"}}]}'
```

Listing 5.4: Commands used to set an AWS S3 bucket for *kops*

Then, it was decided to create a template configuration file (listing 5.5).

```
$ cluster_name="kops.minimal.k8s.local"
$ create_s3_bucket ${K8S_EXP_KOPS_S3_BUCKET} ${K8S_EXP_REGION}
set -x
```

```
# generate a cluster configuration
$ kops create cluster --state "s3://${K8S_EXP_KOPS_S3_BUCKET}" \
  --cloud=aws --kubernetes-version=1.16.9 \
  --cloud-labels="deployment=kops-${K8S_EXP_ENVIRONMENT}" \
  --master-zones="eu-west-1a" --master-count=1 \
  --master-size=t2.micro \
  --zones=eu-west-1a --node-count=1 --node-size=t2.micro \
  ${cluster_name}
# export the configuration to a YAML file
$ kops get -o yaml --name ${cluster_name} --state \
  "s3://${K8S_EXP_KOPS_S3_BUCKET}" > cluster-minimal.kops-tmpl.yaml
```

Listing 5.5: Commands used to generate *kops* configuration

The created file *cluster-minimal.kops-tmpl.yaml* was a Go template [73]. But, in section 4 it was decided to use templates with *Bash* variables. Therefore the created file was copied as *cluster-minimal.tmpl.yaml* and these steps were taken:

- replaced string: testing with string: `${K8S_EXP_ENVIRONMENT}`,
- replaced string: kops.minimal.k8s.local with string: `${K8S_EXP_CLUSTER_NAME}`,
- replaced string: k8s-kops-for-masters-thesis.k8s.local with string: `${K8S_EXP_KOPS_S3_BUCKET}`.

The output of *kops create cluster* command listed the actions which *kops* will perform on the AWS account, e.g. creating EBS volumes for Etcd, configuring IAM, creating keypairs for Kubernetes services, configuring network and setting *EC2 instances*. Details of the to-be-created resources were also provided, for example, the command output informed that the EC2 image will be: *kope.io/k8s-1.16-debian-stretch-amd64-hvm-ebs-2020-01-17*. Apart from printing the output, *kops* created a directory named the same as the cluster name (*testing.k8s-kops-for-masters-thesis.k8s.local*) in the S3 bucket. Among the files automatically created by *kops*, there is a configuration file named: *config* and it contains the cluster settings. **The cluster configuration can be edited from command line** with the command presented below. Running the editing command starts a *vim* session (listing 5.6).

```
$ kops edit cluster ${K8S_EXP_CLUSTER_NAME} \
  --state "s3://${K8S_EXP_KOPS_S3_BUCKET}"
```

Listing 5.6: Command used to edit a Kubernetes cluster managed by *kops*

After editing the configuration, **the cluster can be created or updated** (if it was created earlier) with the next command. This command deploys a cluster on AWS and prints a helpful output (listing 5.7).

```
$ kops update cluster ${K8S_EXP_CLUSTER_NAME} \
  --state "s3://${K8S_EXP_KOPS_S3_BUCKET}" --yes
# some output lines omitted
Cluster is starting. It should be ready in a few minutes.

Suggestions:
* validate cluster: kops validate cluster
* list nodes: kubectl get nodes --show-labels
* ssh to the master: ssh -i ~/.ssh/id_rsa \
  admin@api.testing.k8s-kops-for-masters-thesis.k8s.local
* the admin user is specific to Debian. If not using Debian\
  please use the appropriate user based on your OS.
* read about installing addons at: \
  https://github.com/kubernetes/kops/blob/master/docs/\
  operations/addons.md.
```

Listing 5.7: Command used to deploy a Kubernetes cluster with *kops*

Unfortunately, the suggested above commands, did not work (listing 5.8).

```
$ kops validate cluster ${K8S_EXP_CLUSTER_NAME} \
  --state "s3://${K8S_EXP_KOPS_S3_BUCKET}"
Validating cluster testing.k8s-kops-for-masters-thesis.k8s.local
unexpected error during validation: error listing nodes: \
Get https://api-testing-k8s-kops-for-l9puut-394396927.\
eu-west-1.elb.amazonaws.com/api/v1/nodes: EOF

$ ssh -i ~/.ssh/id_rsa \
  admin@api.testing.k8s-kops-for-masters-thesis.k8s.local
ssh: Could not resolve \
  hostname api.testing.k8s-kops-for-masters-thesis.k8s.local: \
Name does not resolve
```

Listing 5.8: Commands run to connect with a cluster created by *kops* together with output

It was expected that the latter command should fail, because *api.testing.k8s-kops-for-masters-thesis.k8s.local* is not a public domain name and thus, it is not available from remote locations (such as this work author's computer). But the former command should have worked. In practice, there was no way to connect to the *EC2 instances*. Thus, as a

solution – bigger *EC2 instances* were used: *t2.micro* instead of *t2.nano*. Using this particular instance type, *t2.micro*, was observed in several online sources [160, 162, 174]. This time the command succeeded. It was possible to list the worker nodes with: *kubectI get nodes*. It was doable thanks to *kops* creating an AWS Classic LoadBalancer. It exposed the following DNS A record that was publicly reachable: *api-testing-k8s-kops-for-l9puut-1371087518.eu-west-1.elb.amazonaws.com*.

It is also worth mentioning that the kubeconfig (*/.kube/config*), a Kubernetes configuration file needed to connect to a cluster, was generated automatically. Another thing to notice is that the command, which deploys a Kubernetes cluster, returned immediately, without waiting for the cluster to be ready. In the next deployments, some waiting mechanism must be applied, so that the cluster creation and verification can be automated. Listing 5.9 presents command used to request information about the cluster:

```
$ kubectI cluster-info
Kubernetes master is running at \
  https://api-testing-k8s-kops-for-l9puut-1371087518.\
  eu-west-1.elb.amazonaws.com
KubeDNS is running at \
  https://api-testing-k8s-kops-for-l9puut-1371087518.\
  eu-west-1.elb.amazonaws.com/api/v1/namespaces/kube-system/\
  services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use \
  'kubectI cluster-info dump'.
```

Listing 5.9: Command used to request information about a running Kubernetes cluster

Then, **the cluster was deleted**. There were no problems with deleting the cluster. It took several minutes, but the following command succeeded and all the AWS resources (except for the manually created S3 bucket) were deleted.

```
$ kops delete cluster --name ${K8S_EXP_CLUSTER_NAME} \
  --state "s3://${K8S_EXP_KOPS_S3_BUCKET}" --yes
```

Listing 5.10: Command used to delete a Kubernetes cluster created with *kops*

The steps described in this section proved that it is possible to deploy a Kubernetes cluster on AWS with *kops*. It was a POC. The next cluster will attempt to satisfy the production deployment requirements.

5.3 Experimental deployments using *eksctl*

In this subchapter first experiments of deploying a Kubernetes cluster with eksctl on AWS are described.

In order to be consistent, the similar first experiment was performed using *eksctl*. It was decided to store the configuration locally in a YAML configuration file. The alternative was to set many command line flags. **The configuration file and the *eksctl* CLI command used to create a cluster are presented in listing 5.11:**

```
$ cat cluster.yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: eks-testing
  region: eu-west-1
  tags:
    deployment: eks-testing

nodeGroups:
- name: ng-1
  labels: { role: worker, cluster: eks-testing }
  instanceType: t2.nano
  desiredCapacity: 1
  ssh:
    allow: true
$ eksctl create cluster -f cluster.yaml
```

Listing 5.11: Commands used to create a cluster with *eksctl*, without prerequisite steps performed

This resulted in a successful creation of a cluster in "eu-west-1" AWS region with one worker node. It took 19 m 23.379s. Apart from that, the configuration file needed to access the remote cluster (remote, because deployed on AWS) was automatically created and written to: */.kube/config* (same as done with *kops*). Commands needed to **verify that the worker nodes were running** were shown in listing 5.12.

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
ip-192-168-13-139.eu-west-1.compute.internal	Ready	<none>	7m18s	v1.16.8-eks-e16311

Listing 5.12: Command used to list Kubernetes worker nodes to verify that one such node was running

This experiment was successful. **It was easy to deploy a Kubernetes cluster using *eksctl*. No prerequisite steps were needed.** Besides, it was also easy to set up the YAML configuration file, basing on the *eksctl* documentation [180]. However, later it was discovered, that even though all the tests worked using the *t2.nano* worker instance, they should not have worked. They worked, because one pod from the kube-system namespace was not deployed, because there was too little resources for it. Thus, there were just 3 pods running on the worker node and thus, there was a place to deploy the test pod.

The cluster was then deleted with the command presented in listing 5.13.

```
$ eksctl delete cluster -f cluster.yaml --wait
```

Listing 5.13: Command used to delete Kubernetes cluster with *eksctl*

The `--wait` CLI flag was applied. Without it, a delete operation would have been only requested but not waited for. In some cases it happens that the deletion fails, and, without this flag, the errors would not have been propagated back as the CLI command output. Then, one would be forced to delete the AWS resources manually [180].

Any special characters in *eksctl* output, such as ticks, crosses and letter "i" were replaced in this thesis accordingly with: "ok", "no", "info". This was done due to problems with formatting.

5.4 Production deployment using *kops*

This subchapter briefly presents all the steps performed that lead to a Kubernetes cluster deployment on the AWS cloud using kops. Here an attempt was made to satisfy all the production environment requirements selected in chapter 4.

5.4.1 Generating the YAML configuration file

Commands was used to generate a cluster YAML configuration file. Creating the YAML file is not recommended to be done by hand. Rather, one should first run *kops create cluster* command (which creates a cluster configuration in a S3 bucket state store) and then export the configuration from the state store with: *kops get -o yaml*. There are plans to change it. Using YAML instead of CLI has another advantage: more options can be set [92] and addons can be deployed [72].

The S3 bucket was reused from previous experiments. The cluster YAML configuration file was generated with the command presented in listing 5.14.

```
$ my_ip=$(curl https://ipinfo.io/ip 2>/dev/null)
kops create cluster --state="s3://${K8S_EXP_KOPS_S3_BUCKET}" \
--kubernetes-version=1.16.9 \
--master-zones="eu-west-1a,eu-west-1b,eu-west-1c" \
--master-count=3 --master-size=t2.micro \
--zones=eu-west-1a --node-count=1 --node-size=t2.micro \
--ssh-access=${my_ip}/32 \
${K8S_EXP_CLUSTER_NAME}
```

Listing 5.14: Commands used to generate a cluster configuration with *kops*

The options used to create a production *kops* cluster were described in table 5.1.

Table 5.1: Options set to *kops* when creating a Kubernetes cluster

Option	Description
--state="s3://\${K8S_EXP_KOPS_S3_BUCKET}"	set a S3 bucket name
--kubernetes-version=1.16.9	choose a particular Kubernetes version
--master-zones="eu-west-1a,eu-west-1b,eu-west-1c"	choose AZs for master instances (to ensure HA)
--master-count=3	deploy 3 master nodes
--master-size=t2.micro	choose <i>EC2 instance type</i> for master nodes
--zones="eu-west-1a"	choose AZs for node instances
--node-count=1	deploy 1 worker node
--node-size=t2.micro	choose <i>EC2 instance type</i> for worker nodes
--ssh-access=\${my_ip}/32	apply more security

Then, the configuration was exported to local filesystem, as configuration templates:

```
$ kops get -o yaml --name ${K8S_EXP_CLUSTER_NAME} \
  --state "s3://${K8S_EXP_KOPS_S3_BUCKET}" > cluster.kops-tmpl.yaml
```

Listing 5.15: Command used export *kops* configuration from S3 to a local file

Afterwards, the same step was performed as for the experimental deployment, namely, the configuration template file was copied and saved as *cluster.tmpl.yaml*. Next, the latter file was edited in the following way:

- *kubernetesApiAccess* and *sshAccess* were set in order to ensure more security - only one IP was allowed to communicate with the Kubernetes cluster,
- replaced string: testing with `${K8S_EXP_ENVIRONMENT}`; now, it is possible to decide on the environment at the moment of cluster deployment,
- replaced string: `kops.minimal.k8s.local` with `${K8S_EXP_CLUSTER_NAME}`,
- replaced string: `k8s-kops-for-masters-thesis.k8s.local` with `${K8S_EXP_KOPS_S3_BUCKET}`,
- *cloudLabels* were specified; now, it is easier to identify which AWS resources belong to this Kubernetes deployment [90],
- AWS IAM permissions were added, so that Kubernetes logs may go to *AWS CloudWatch*.

The added AWS IAM permissions are presented in listing 5.16:

```
additionalPolicies:
  node: |
    [
      {
        "Effect": "Allow",
        "Action": ["logs:CreateLogGroup", "logs:CreateLogStream", "logs:PutLogEvents",
          "logs:DescribeLogGroups", "logs:DescribeLogStreams"],
        "Resource": ["*"]
      }
    ]
  master: |
    [
      {
        "Effect": "Allow",
        "Action": ["logs:CreateLogGroup", "logs:CreateLogStream", "logs:PutLogEvents",
          "logs:DescribeLogGroups", "logs:DescribeLogStreams"],
        "Resource": ["*"]
      }
    ]
```

```
}
```

Listing 5.16: AWS IAM permissions added to *kops* cluster template needed for logging

Most of this work is implemented in the *Bash* script and can be run as in listing 5.17.

```
kops$ ./tasks _gen_config_full
# then perform the tinkering described above (set IAM permissions, labels and
# security settings)
```

Listing 5.17: *Bash* commands automating cluster configuration generation

5.4.2 Creating a cluster

All the work described in this section, needed to deploy a Kubernetes cluster is fully automated. In the future it may be used in a *Continuous Integration (CI)* pipeline. This step can be executed with one command, presented in listing 5.18.

```
kops$ ./kops/tasks _create
```

Listing 5.18: *Bash* command automating cluster creation

This command utilizes the *Bash* variables set: *K8S_EXP_ENVIRONMENT*, *K8S_EXP_CLUSTER_NAME*, *K8S_EXP_KOPS_S3_BUCKET* and creates *cluster.yaml* file (out of the *Bash* template file: *cluster.tmpl.yaml*). Then, a *kops* secret with SSH key is created. Then it updates the configuration on S3 using the local file *cluster.tmpl.yaml*. Then, it creates a cluster. It creates AWS resources needed for Kubernetes cluster (*EC2 instances*, *VPC*, etc.).

It was decided that this command should stop only after the Kubernetes cluster is ready. Thus, a simple *Bash* loop was implemented, which checks every 1 second if a cluster is ready. The loop is presented in listing 5.19.

```
while [ 1 ]; do
  kops validate cluster ${K8S_EXP_CLUSTER_NAME} --state \
    "s3://${K8S_EXP_KOPS_S3_BUCKET}" && break || sleep 30
done;
```

Listing 5.19: A waiting mechanism that waits until a *kops* cluster is ready

Thanks to the custom cloud label: *deployment*, it is now possible to list all the AWS *EC2 instances*, which have this AWS tag (cloud label) set (listing 5.20).

```
$ aws ec2 describe-instances --filters "Name=tag-key,Values=deployment" \
--query "Reservations[*].Instances[*].{PublicIP:PublicIpAddress,\
Name:Tags[?Key=='Name']|[0].Value,Status:State.Name}"
[
  [
    {
      "PublicIP": "34.242.4.10",
      "Name": "master-eu-west-1a.masters.testing.k8s-kops-for-masters-thesis.k8s.local",
      "Status": "running"
    }
  ],
  [
    {
      "PublicIP": "34.247.37.124",
      "Name": "nodes.testing.k8s-kops-for-masters-thesis.k8s.local",
      "Status": "running"
    }
  ]
]
```

Listing 5.20: Listing all AWS EC2 instances by AWS tag

All the deployed Kubernetes components are listed in listing 5.21.

```
$ kubectl get -n kube-system pods
```

NAME	READY	STATUS	\
RESTARTS AGE			
dns-controller-776cdf4ff4-lzb8f	1/1	Running	\
o 2m28s			
etcd-manager-events-ip-172-20-39-5.eu-west-1.compute.internal	1/1	Running	\
o 2m20s			
etcd-manager-main-ip-172-20-39-5.eu-west-1.compute.internal	1/1	Running	\
o 2m24s			
kops-controller-49hsh	1/1	Running	\
o 89s			
kube-apiserver-ip-172-20-39-5.eu-west-1.compute.internal	1/1	Running	\
3 82s			
kube-controller-manager-ip-172-20-39-5.eu-west-1.compute.internal	1/1	Running	\
o 100s			
kube-dns-autoscaler-594dcb44b5-7psf9	1/1	Running	\
o 2m31s			
kube-dns-b84c667f4-4wvsx	3/3	Running	\
o 2m32s			
kube-dns-b84c667f4-kgdgb	3/3	Running	\
o 61s			
kube-proxy-ip-172-20-39-5.eu-west-1.compute.internal	1/1	Running	\
o 2m20s			

kube-scheduler-ip-172-20-39-5.eu-west-1.compute.internal	1/1	Running	\
o	93s		

Listing 5.21: Kubernetes components in a *kops* cluster

So far, all the necessary work was performed in order to have a working cluster. The next thing is to deploy logging. The code presented in listing 5.22 was needed to attain this goal.

```
$ aws logs create-log-group --log-group-name k8s-kops-${K8S_EXP_ENVIRONMENT}
$ helm install "kube2iam" --namespace="default" --wait --atomic --set rbac.create=true \
  stable/kube2iam
$ helm repo add incubator https://kubernetes-charts-incubator.storage.googleapis.com
$ helm install "fluentd-cloudwatch" --namespace="default" --wait --atomic \
  --set awsRegion=${K8S_EXP_REGION},rbac.create=true,\
  logGroupName=k8s-kops-${K8S_EXP_ENVIRONMENT} incubator/fluentd-cloudwatch
```

Listing 5.22: Commands providing the logging solution

An AWS CloudWatch LogGroup was created first. Then, two *Helm* charts were deployed. The *kube2iam* chart handles the permissions of Kubernetes pods and the *fluentd-cloudwatch* chart is responsible for transporting the logs from Kubernetes services to AWS CloudWatch. As a result, the log messages may be viewed on AWS Management Console. (Fig. 5.1 and 5.2).

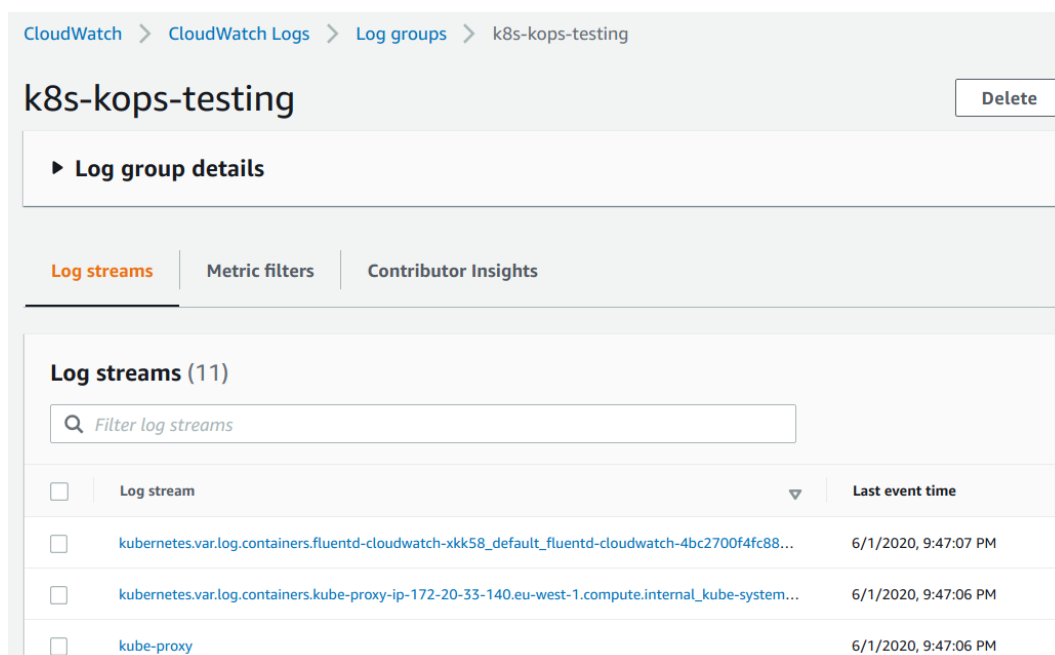


Fig. 5.1: AWS CloudWatch with logs from a Kubernetes cluster

Log events		
<input type="text" value="Filter events"/>		
▶	Timestamp	Message
	There are older events to load. Load more.	
▶	2020-06-01T21:46:12.132+02:00	{"log": "2020-06-01 19:46:12 +0000 [info]: gem 'fluent-plugin-multi-format-parser' version '1.0.0'\n",
▶	2020-06-01T21:46:12.132+02:00	{"log": "2020-06-01 19:46:12 +0000 [info]: gem 'fluent-plugin-prometheus' version '1.0.0'\n",
▶	2020-06-01T21:46:12.132+02:00	{"log": "2020-06-01 19:46:12 +0000 [info]: gem 'fluent-plugin-record-modifier' version '1.0.0'\n",
▶	2020-06-01T21:46:12.132+02:00	{"log": "2020-06-01 19:46:12 +0000 [info]: gem 'fluent-plugin-rewrite-tag-filter' version '1.0.0'\n",
▶	2020-06-01T21:46:12.132+02:00	{"log": "2020-06-01 19:46:12 +0000 [info]: gem 'fluent-plugin-systemd' version '1.0.0'\n",

Fig. 5.2: AWS CloudWatch with logs from a Kubernetes cluster

An example log message is presented in listing 5.23:

```
{
  "log": "2020-06-01 19:46:12 +0000 [info]: gem 'fluent-plugin-multi-format-parser' version '1.0.0'\n",
  "stream": "stdout",
  "docker": {
    "container_id": "4bc270of4fc88530b4e03b2ed794c1f1b366be6931d8500a8ccf21503d2c5b97"
  },
  "kubernetes": {
    "container_name": "fluentd-cloudwatch",
    "namespace_name": "default",
    "pod_name": "fluentd-cloudwatch-xkk58",
    "container_image": "fluent/fluentd-kubernetes-daemonset:v1.7.3-debian-cloudwatch-1.0",
    "container_image_id": "docker-pullable://fluent/fluentd-kubernetes-daemonset@sha256:9b8b2f99ea884853205150364eceaac9fff5ea97fc330occo080f48c3eac8b8a",
    "pod_id": "9fc38bd1-ff7b-4d29-9593-15796a7e5f94",
    "host": "ip-172-20-33-140.eu-west-1.compute.internal",
    "labels": {
      "app": "fluentd-cloudwatch",
      "controller-revision-hash": "68f7dc7cc",
      "pod-template-generation": "1",
      "release": "fluentd-cloudwatch"
    },
    "master_url": "https://100.64.0.1:443/api",
    "namespace_id": "4044bfd2-e836-43a1-bda1-71dae985b9a"
  }
}
```

Listing 5.23: An example Kubernetes log message, presented on AWS CloudWatch

It can be also verified, that the custom labels applied to AWS resources by the cluster YAML configuration file, are indeed visible. The custom label here is *deployment: kops-testing* (Fig. 5.3).

Key	Value
KubernetesCluster	testing.k8s-kops-for-masters-thesis.k8s.local
Name	master-eu-west-1a.masters.testing.k8s-kops-for-masters-thesis.k8s.local
aws:autoscaling:groupName	master-eu-west-1a.masters.testing.k8s-kops-for-masters-thesis.k8s.local
deployment	kops-testing
k8s.io/cluster-autoscaler/node-template/label/kops.k8s.io/instancegroup	master-eu-west-1a
k8s.io/role/master	1
kops.k8s.io/instancegroup	master-eu-west-1a

Fig. 5.3: AWS Management Console showing tags applied on EC2 instance which is a master node

When it comes to monitoring the cluster resources, some metrics are available by default on AWS EC2 dashboard. The information concerning CPU utilization or incoming network quantity in bytes can be viewed without any additional configuration. The example charts are presented in figures 5.4 and 5.5.

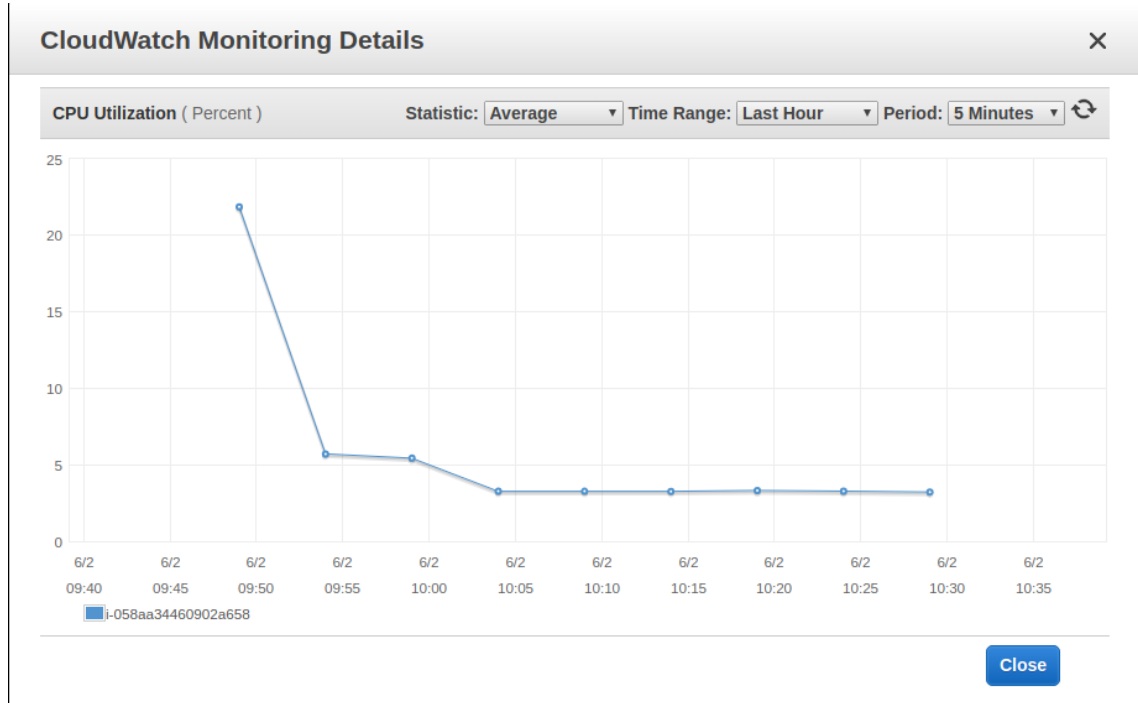


Fig. 5.4: AWS CloudWatch metrics of a worker node, CPU utilization

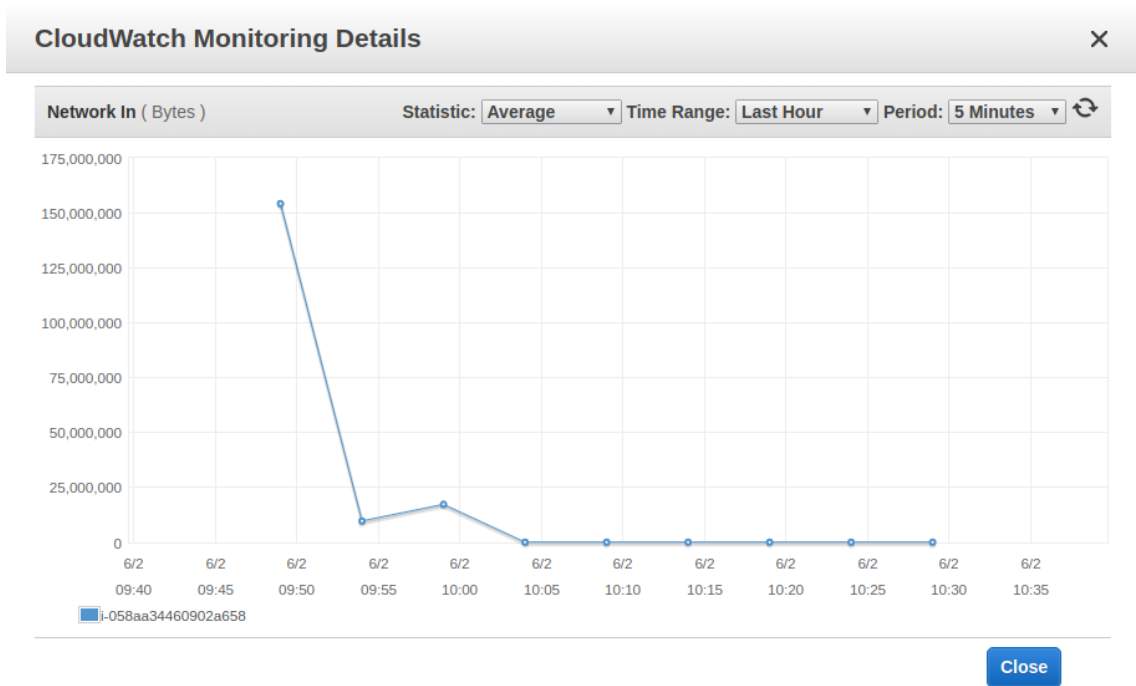


Fig. 5.5: AWS CloudWatch metrics of a worker node, Network input

When it comes to **Central Audit** requirement, no manual steps were needed. One can just log into *AWS Management Console* through a web browser, go to the *AWS CloudTrail* service and view the events history. Each event can be viewed and in return, information in *JSON* format is received. A table with events is presented in figure 5.6.

Filter:	Read only	false	Time range: Select time range		
	Event time	User name	Event name	Resource type	Resource name
▶	2020-06-02, 12:52:28 PM	ewa-admin	DeleteSubnet	EC2 Subnet	subnet-0fca2b6d15a0d3096
▼	2020-06-02, 12:52:28 PM	ewa-admin	DeleteSecurityGroup	EC2 SecurityGroup	sg-0d4889d85db65a8c8
<div><div><div>AWS access key</div><div>AKIA6GSJD56SM6WV5COI</div></div><div><div>AWS region</div><div>eu-west-1</div></div><div><div>Error code</div><div></div></div><div><div>Event ID</div><div>87ca7ec4-2ea9-4b2d-98c0-8cce612e6d2e</div></div><div><div>Event name</div><div>DeleteSecurityGroup</div></div><div><div>Event source</div><div>ec2.amazonaws.com</div></div><div><div>Event time</div><div>2020-06-02, 12:52:28 PM</div></div><div><div>Read only</div><div>false</div></div><div><div>Request ID</div><div>6e5d7b3d-5648-4571-8e7e-93b5a254ae45</div></div><div><div>Source IP address</div><div>31.183.242.155</div></div><div><div>User name</div><div>ewa-admin</div></div></div>					
Resources Referenced (1)					
Resource type			Resource name		Config timeline
EC2 SecurityGroup			sg-0d4889d85db65a8c8		⏮
View event					
▶	2020-06-02, 12:52:25 PM	i-06a35e1c7c60fae92	UpdateInstanceInformation		
▶	2020-06-02, 12:52:17 PM	ewa-admin	DeleteInternetGateway	EC2 InternetGateway	igw-0f216f4449b32704a

Fig. 5.6: AWS CloudTrail events of a kops cluster

The next operation is to backup a cluster namespace. *Velero* was chosen as an application to facilitate backup. In order to install *Velero* CLI, one can run the command presented in listing 5.24.

```
kops$ ./tasks _install_velero_cli
```

Listing 5.24: Automated command to install *Velero* CLI

This command installed *Velero* client. A *Velero* Server must be also deployed on a Kubernetes cluster. Before it is done, AWS IAM user must be created and proper permissions must be assigned to it. The instructions how to deal with AWS IAM for *Velero* were available at *Github.com* [176]. In order to be able to repeat the steps easily, all the instructions were saved to a *Bash* script as: *src/velero/generate-velero-credentials.sh*. Running it should result in generating two credentials and generating a file called *credentials-velero* in the current directory (listing 5.25).

```
[ default ]
aws_access_key_id=<REPLACE-ME-WITH-SECRET>
aws_secret_access_key=<REPLACE-ME-WITH-SECRET>
```

Listing 5.25: Contents of AWS credentials file

Afterwards, a *Velero* Server was deployed, using a *Helm* chart (listing 5.26).

```
$ kubectl create namespace velero
$ helm repo add vmware-tanzu https://vmware-tanzu.github.io/helm-charts
$ helm install velero --namespace velero -f ${PWD}/../velero/velero-chart-values.yaml \
--set configuration.backupStorageLocation.bucket=${K8S_EXP_VELERO_S3_BUCKET} \
--set configuration.backupStorageLocation.prefix=${K8S_EXP_ENVIRONMENT} \
--set configuration.backupStorageLocation.config.region=${K8S_EXP_REGION} \
--set-file credentials.secretContents.cloud=credentials-velero \
--wait --atomic vmware-tanzu/velero
```

Listing 5.26: Deploying a vs *Helm* chart

Then, a backup location for *Velero* was configured (listing 5.27).

```
$ velero backup-location create default \
--provider aws \
--bucket ${K8S_EXP_VELERO_S3_BUCKET} \
--prefix ${K8S_EXP_ENVIRONMENT} \
--config region=${K8S_EXP_REGION}
Backup storage location "default" configured successfully.
```

Listing 5.27: Choosing a backup storage location for *Velero*

And then, finally, the steps from *Velero* documentation was followed [97]. A test application (provided by *Velero*) was deployed and backed up. The test application was deployed to a Kubernetes namespace: *nginx-example*. It was verified also that the Kubernetes resources were created (listing 5.28).

```
$ kubectl apply -f ../velero/velero-example.yaml
namespace/nginx-example created
deployment.apps/nginx-deployment created
service/my-nginx created
$ kubectl get -n nginx-example all
```

NAME	READY	STATUS	RESTARTS	AGE
pod/nginx-deployment-7cd5ddccc7-9drfn	1/1	Running	0	26s
pod/nginx-deployment-7cd5ddccc7-hr9zg	1/1	Running	0	7s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/my-nginx	LoadBalancer	100.64.170.74	a64fbc3f7b55c45aead4f113025d8a34-1346547608.eu-west-1.elb.amazonaws.com	80:31927/TCP	7s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/nginx-deployment	1/2	2	1	7s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/nginx-deployment-7cd5ddccc7	2	2	1	8s

```
$ velero backup create nginx-backup1 --include-namespaces nginx-example
Backup request "nginx-backup1" submitted successfully.
Run 'velero backup describe nginx-backup1' or 'velero backup logs nginx-backup1' \
for more details.
$ velero backup describe nginx-backup1
Name:          nginx-backup1
# output partially omitted
Phase: Completed
$ velero backup logs nginx-backup1
# output partially omitted
time="2020-06-03T13:35:51Z" level=info msg="Backed up a total of 24 items" \
backup=velero/nginx-backup1 logSource="pkg/backup/backup.go:436" progress=
```

Listing 5.28: Testing backup operation

The backup result was stored in an S3 bucket and it is presented in figure 5.7.

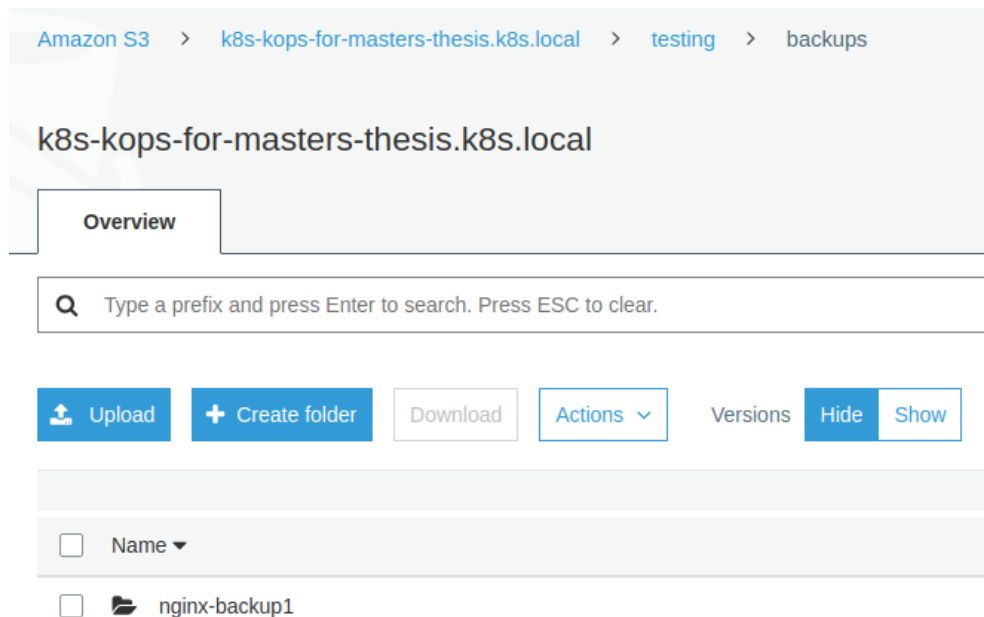


Fig. 5.7: Backup of a Kubernetes namespace created by Velero,
kept in an S3 bucket

Next, a disaster was simulated. The whole *nginx-example* namespace was deleted (listing 5.29).

```
$ kubectl delete namespaces nginx-example
namespace "nginx-example" deleted
$ kubectl get -n nginx-example all
No resources found in nginx-example namespace.
```

Listing 5.29: Simulating a disaster to test backup

The last step was to restore the namespace from *Velero* backup. It was successful (listing 5.30).

```
$ velero restore create --from-backup nginx-backup1
Restore request "nginx-backup1-20200603133811" submitted successfully.
Run 'velero restore describe nginx-backup1-20200603133811' or \
  'velero restore logs nginx-backup1-20200603133811' for more details.
$ kubectl get -n nginx-example all
```

NAME	READY	STATUS	RESTARTS	AGE
pod/nginx-deployment-7cd5ddccc7-9drfn	1/1	Running	0	9s
pod/nginx-deployment-7cd5ddccc7-hr9zg	1/1	Running	0	8s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/my-nginx	LoadBalancer	100.71.21.114	\		

abf6080d1d43a43ba97999bb6b22cabc — 2114878664.eu-west-1.elb.amazonaws.com 80:32539/TCP					
8s					
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/nginx-deployment	2/2	2	2	8s	
NAME	DESIRED		CURRENT	READY	AGE
replicaset.apps/nginx-deployment-7cd5ddccc7	2		2	2	8s

Listing 5.30: Restoring the backup

Then, autoscaling was tested. The autoscaler Kubernetes resources were deployed with command presented in listing 5.31.

```
kops$ ./tasks _enable_as
```

Listing 5.31: *Bash* command to deploy autoscaler

which basically just deployed the resources configured at autoscaler git repository [109]. Some modifications were also applied to the cluster *YAML* configuration, namely: under *additionalPolicies* of node configuration (listing 5.32).

```
{
  "Effect": "Allow",
  "Action": [
    "autoscaling:DescribeAutoScalingGroups",
    "autoscaling:DescribeAutoScalingInstances",
    "autoscaling:DescribeLaunchConfigurations",
    "autoscaling:SetDesiredCapacity",
    "autoscaling:TerminateInstanceInAutoScalingGroup",
    "autoscaling:DescribeTags"
  ],
  "Resource": "*"
}
```

Listing 5.32: IAM Policies needed by autoscaler

Also, additional labels for *InstanceGroup* of worker nodes were set and maximum number of worker nodes were set to 2. This went smoothly thanks to a blog post [161]. The modified configuration is presented in listing 5.33.

```
spec:
  cloudLabels:
    service: k8s_node
    k8s.io/cluster-autoscaler/enabled: ""
    k8s.io/cluster-autoscaler/testing.k8s-kops-for-masters-thesis.k8s.local: ""
```

```
minSize: 1
maxSize: 2
```

Listing 5.33: *Kops* cluster YAML configuration needed for autoscaler

In order to test the autoscaling, a test application was deployed with command in listing 5.34.

```
tests$ ./deploy-test-service.sh
```

Listing 5.34: Deploying a test application

After the above command ended with success, it was verified that "1" pod replica was deployed. Since, the task here was to make the autoscaler decide to scale out – to add another node. The test application deployment was scaled, so that 200 pod replicas were deployed (listing 5.35).

```
kops$ kubectl get -n testing deployment
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
apache-testing      1/1     1            1           9m48s
kops$ kubectl scale -n testing deployment apache-testing --replicas=200
deployment.apps/apache-testing scaled
kops$ kubectl get -n testing deployment
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
apache-testing      12/200   200          12          13m
```

Listing 5.35: Scaling the test application pod replicas

This triggered the creation of another node and in a short while, there were two nodes in the cluster (listing 5.36).

```
kops$ aws ec2 describe-instances --filters "Name=tag-key,Values=deployment" \
--query "Reservations[*].Instances[*].{PublicIP:PublicIpAddress,Name:Tags\
[?Key=='Name']|[0].Value,Status:State.Name}"
[
  [
    {
      "PublicIP": "34.242.4.10",
      "Name": "master-eu-west-1a.masters.testing.k8s-kops-for-masters-thesis.k8s.local",
      "Status": "running"
    }
  ],
  [
    {
      "PublicIP": "34.247.37.124",
      "Name": "nodes.testing.k8s-kops-for-masters-thesis.k8s.local",

```

```

        "Status": "running"
    }
],
[
    {
        "PublicIP": "3.249.123.18",
        "Name": "nodes.testing.k8s-kops-for-masters-thesis.k8s.local",
        "Status": "running"
    }
]
]
]

```

Listing 5.36: Listing the *EC2 instances*, 1 of them was created by autoscaler

Afterwards, the pod replicas were scaled back to "1" and then, autoscaler terminated one of the worker nodes. Soon, one of the *EC2 instances* was in the status: *erminated*. This proved that autoscaler works for a Kubernetes cluster deployed with *kops*.

When it comes to running a cluster in the High Availability mode, there were no problems whatsoever. No matter if the cluster was created with one master node or multiple master nodes – it was working. In *High Availability* mode, the information about the cluster, listing the master and the worker nodes, is demonstrated in listing 5.37. Several master machines were observed.

```

$ kubectl get nodes
NAME                                                    STATUS  ROLES    AGE   VERSION
ip-172-20-108-29.eu-west-1.compute.internal           Ready   master   73s   v1.16.9
ip-172-20-55-72.eu-west-1.compute.internal            Ready   node     49s   v1.16.9
ip-172-20-61-242.eu-west-1.compute.internal           Ready   master   74s   v1.16.9
ip-172-20-89-96.eu-west-1.compute.internal            Ready   master   75s   v1.16.9
$ kops validate
Validating cluster testing.k8s-kops-for-masters-thesis.k8s.local

INSTANCE GROUPS
NAME                ROLE    MACHINETYPE  MIN  MAX  SUBNETS
master-eu-west-1a   Master  t2.micro      1    1    eu-west-1a
master-eu-west-1b   Master  t2.micro      1    1    eu-west-1b
master-eu-west-1c   Master  t2.micro      1    1    eu-west-1c
nodes               Node    t2.micro      1    1    eu-west-1a

NODE STATUS
NAME                                                    ROLE    READY
ip-172-20-108-29.eu-west-1.compute.internal           master   True
ip-172-20-55-72.eu-west-1.compute.internal            node     True
ip-172-20-61-242.eu-west-1.compute.internal           master   True

```

```
ip-172-20-89-96.eu-west-1.compute.internal      master    True

Your cluster testing.k8s-kops-for-masters-thesis.k8s.local is ready
```

Listing 5.37: Getting information about the cluster

The machines can be also viewed on AWS Management Console (Fig. 5.8).

<input type="checkbox"/>	Name	deployment	Instance ID	Instanc	Availabilit	Instance Sta
<input type="checkbox"/>	nodes.testing.k8s-kops-for-master...	kops-testing	i-0de20b2f09a2065c8	t2.micro	eu-west-1a	running
<input type="checkbox"/>	master-eu-west-1c.masters.testin...	kops-testing	i-05f66e586088c0949	t2.micro	eu-west-1c	running
<input type="checkbox"/>	master-eu-west-1b.masters.testin...	kops-testing	i-01f54a8cf76c4e600	t2.micro	eu-west-1b	running
<input type="checkbox"/>	master-eu-west-1a.masters.testin...	kops-testing	i-0cde519ba47185656	t2.micro	eu-west-1a	running

Fig. 5.8: EC2 instance in HA mode for kops cluster

The tests were invoked both in *High Availability* mode and also with one master cluster. The tests succeeded in both cases. In HA mode, all the Kubernetes control plane components were duplicated in this case – three times (listing 5.38).

```
$ kubectl -n kube-system get pod
NAME                READY\
STATUS  RESTARTS  AGE
dns-controller-776cdf4ff4-xw4tg                1/1\
Running o                2m18s
etcd-manager-events-ip-172-20-108-29.eu-west-1.compute.internal 1/1\
Running o                66s
etcd-manager-events-ip-172-20-61-242.eu-west-1.compute.internal 1/1 \
Running o                86s
etcd-manager-events-ip-172-20-89-96.eu-west-1.compute.internal 1/1\
Running o                96s
etcd-manager-main-ip-172-20-108-29.eu-west-1.compute.internal 1/1\
Running o                53s
etcd-manager-main-ip-172-20-61-242.eu-west-1.compute.internal 1/1\
Running o                56s
etcd-manager-main-ip-172-20-89-96.eu-west-1.compute.internal 1/1\
Running o                80s
kops-controller-2tz4h                1/1\
Running o                2m8s
kops-controller-7mr7s                1/1\
Running o                2m14s
kops-controller-psfch                1/1\
Running o                2m8s
kube-apiserver-ip-172-20-108-29.eu-west-1.compute.internal 1/1\
```

Running 3	110 s	
kube-apiserver-ip-172-20-61-242.eu-west-1.compute.internal		1/1\
Running 3	114 s	
kube-apiserver-ip-172-20-89-96.eu-west-1.compute.internal		1/1\
Running 3	92 s	
kube-controller-manager-ip-172-20-108-29.eu-west-1.compute.internal		1/1\
Running 0	2m6s	
kube-controller-manager-ip-172-20-61-242.eu-west-1.compute.internal		1/1\
Running 0	105 s	
kube-controller-manager-ip-172-20-89-96.eu-west-1.compute.internal		1/1\
Running 0	79 s	
kube-dns-autoscaler-594dcb44b5-ccnjf		1/1\
Running 0	2m19s	
kube-dns-b84c667f4-5tb9z		3/3\
Running 0	105 s	
kube-dns-b84c667f4-8vmh4		3/3\
Running 0	2m21s	
kube-proxy-ip-172-20-108-29.eu-west-1.compute.internal		1/1\
Running 0	116 s	
kube-proxy-ip-172-20-55-72.eu-west-1.compute.internal		1/1\
Running 0	110 s	
kube-proxy-ip-172-20-61-242.eu-west-1.compute.internal		1/1\
Running 0	65 s	
kube-proxy-ip-172-20-89-96.eu-west-1.compute.internal		1/1\
Running 0	61 s	
kube-scheduler-ip-172-20-108-29.eu-west-1.compute.internal		1/1\
Running 0	2m	
kube-scheduler-ip-172-20-61-242.eu-west-1.compute.internal		1/1\
Running 0	2m13s	
kube-scheduler-ip-172-20-89-96.eu-west-1.compute.internal		1/1\
Running 0	112 s	

Listing 5.38: Kubernetes control plane components, *High Availability*

5.4.3 Testing a cluster

All the tests can be invoked with a single command (listing 5.39).

```
kops$ ./tasks _test
```

Listing 5.39: Testing a *kops* cluster

The command responsible for testing performs several steps. They are shown in listing 5.40.

```
$ set -x
```



```
$ time kops validate cluster ${K8S_EXP_CLUSTER_NAME} --state \
"s3://${K8S_EXP_KOPS_S3_BUCKET}"
$ cd ../tests
$ time bats tests.bats
$ cd ../kops
```

Listing 5.40: Testing a *kops* cluster - deeper dive

The *kops validate cluster* command is provided by the *kops* CLI. It checks that all k8s master and worker nodes are running and have "Ready" status and that all the control plane components are healthy [88]. The next command runs the *Bats-core* tests. They test the version of Kubernetes, number of worker nodes and also they deploy a test *Helm* chart with Apache Server. All the tests and the deployment (and later deletion) of the Apache *Helm* chart are automated. Thanks to such a test, it is verified that such a Kubernetes cluster is ready for use for the end users.

The test Apache Server serves a very simple website, which all code is just one *index.html* file with contents presented in listing 5.41:

```
<html>
<head>
<title>Hello world</title>
</head>
<body>
<p>Welcome to my Master Thesis website!</p>
</body>
</html>
```

Listing 5.41: Contents of a test application - Apache web server

Thus, there is a test that runs *curl* command and expects that the output contains "Welcome to my Master Thesis website!". Such a test is possible thanks to an ingress resource provided by the Apache *Helm* Chart. When this ingress is deployed on a Kubernetes cluster on AWS, an Elastic Load Balancer (ELB) is created. ELB is an AWS resource. The ELB provides us with the endpoint that can be used with *curl*.

The tests were run with *Bats-core* and their code is presented in listing 5.42.

```
@test "kubernetes machines have correct version" {
  run /bin/bash -c "kubectl version | grep 'Server Version'"
  # this is printed on test failure only
  echo "$# test cmd output: $output"
  [[ "${output}" =~ "v1.16" ]]
```

```

[ "$status" -eq 0 ]
}

@test "1 kubernetes worker nodes have status: Ready" {
    run /bin/bash -c "chmod +x get-workers-count.sh && ./get-workers-count.sh"
    # this is printed on test failure only
    echo "# test cmd output: $output"
    [[ "${output}" =~ "1" ]]
    [ "$status" -eq 0 ]
}

@test "kubectl cluster-info returns expected output" {
    run /bin/bash -c "kubectl cluster-info"
    # this is printed on test failure only
    echo "# test cmd output: $output"
    [[ "${output}" =~ "Kubernetes master" ]]
    [[ "${output}" =~ "https://" ]]
    [ "$status" -eq 0 ]
}

@test "a test application can be deployed on kubernetes" {
    run /bin/bash -c "chmod +x ./deploy-test-service.sh && ./deploy-test-service.sh"
    # this is printed on test failure only
    echo "# test cmd output: $output"
    [[ "${output}" =~ "Success" ]]
    [ "$status" -eq 0 ]

    # k8s resource: pod was created
    run /bin/bash -c "kubectl get --namespace=testing pods"
    # this is printed on test failure only
    echo "# test cmd output: $output"
    [[ "${output}" =~ "apache-testing" ]]
    [[ "${output}" =~ "Running" ]]
    [ "$status" -eq 0 ]

    # k8s resource: svc was created
    run /bin/bash -c "kubectl get --namespace=testing svc"
    # this is printed on test failure only
    echo "# test cmd output: $output"
    [[ "${output}" =~ "apache-testing" ]]
    [ "$status" -eq 0 ]

    # k8s resource: ing was created
    run /bin/bash -c "kubectl get --namespace=testing ing"
    # this is printed on test failure only
    echo "# test cmd output: $output"
    [[ "${output}" =~ "apache-testing" ]]

```

```

[ "$status" -eq 0 ]
}

@test "a test application is available (ingress resource works)" {
    # code omitted for brevity of reading ,
    # the code is available in the Internet
}

@test "a test service (test website) can be deleted from kubernetes" {
    run /bin/bash -c 'helm uninstall --namespace="testing" "apache-testing" &&\
        kubectl delete -f config-map-www-contents.yaml && kubectl delete namespace "testing"'
    echo "# test cmd output: $output"
    [[ "${output}" =~ "uninstalled" ]]
    [ "$status" -eq 0 ]

    # pod, svc and ing resources were deleted
    run /bin/bash -c 'kubectl get --namespace=testing pods,svc,ing'
    echo "# test cmd output: $output"
    [[ "${output}" =~ "No resources found in testing namespace" ]]
    [ "$status" -eq 0 ]
}

```

Listing 5.42: *Bats-core* tests

Several helpful scripts were created and used for the tests. All the needed files are placed in the git repository on *Github.com*.

5.4.4 Deleting a cluster

Deleting a cluster was fully automated and performed with command in listing 5.43.

```
kops$ ./tasks _delete
```

Listing 5.43: Testing a *kops* cluster

5.4.5 Troubleshooting

The first problem of deploying a Kubernetes cluster with *kops* was the command responsible for creating a cluster. The command did not wait until the cluster was ready. It returned as soon as the AWS resources were scheduled for creation. This was a problem, because if one wanted to create, test and then delete a Kubernetes cluster in a CI pipeline,

then the tests would never work. And the pipeline would always fail. The solution for this problem was already described in the section 5.4.2.

The second problem touches configuring sending log messages to *AWS CloudWatch*. The documentation of the *Helm* chart: *kube2iam* [163] provided an example command of how to deploy the chart: *helm install stable/kube2iam --name my-release*. But this resulted in errors, which are presented in listing 5.44, thanks to debugging the *kube2iam* pod.

```
$ kubectl get pod
NAME                                READY   STATUS              RESTARTS   AGE
kube2iam-2c8vv                     0/1     CrashLoopBackOff    9           17m
$ kubectl logs kube2iam-2c8vv
E0601 10:35:53.142224          1 reflector.go:199] github.com/jtblin/kube2iam/\
  vendor/k8s.io/client-go/tools/cache/reflector.go:94: Failed to list *v1.Pod:\
  pods is forbidden: User "system:serviceaccount:default:default" \
  cannot list resource "pods" in API group "" at the cluster scope
E0601 10:35:53.142621          1 reflector.go:199] github.com/jtblin/kube2iam/\
  vendor/k8s.io/client-go/tools/cache/reflector.go:94: Failed to list *v1.Namespace:\
  namespaces is forbidden: User "system:serviceaccount:default:default" \
  cannot list resource "namespaces" in API group "" at the cluster scope
```

Listing 5.44: Debugging kube2iam pod

These errors went away, when the chart was deployed with additional settings set: *helm upgrade "kube2iam" --namespace="default" --wait --atomic --set rbac.create=true stable/kube2iam*.

The third problem concerns the lack of permissions of the *fluentd-cloudwatch* pod. It manifested in the way presented in listing 5.45:

```
$ kubectl logs fluentd-cloudwatch-xkk58
2020-06-01 11:03:14 +0000 [warn]: #0 [out_cloudwatch_logs_host_logs] failed to \
  flush the buffer. retry_time=8 next_retry_seconds=2020-06-01 11:05:22 +0000 \
  chunk="5a703b5e45c9592f24399f9b73acaf43" error_class=Aws::CloudWatchLogs::\
  Errors::AccessDeniedException error="User: arn:aws:sts::976184668068:\
  assumed-role/nodes.testing.k8s-kops-for-masters-thesis.k8s.local/i-04a926040234f36d6\
  is not authorized to perform: logs:DescribeLogGroups on resource: \
  arn:aws:logs:eu-west-1:976184668068:log-group::log-stream:"
```

Listing 5.45: Debugging the fluent-cloudwatch pod

This problem was also already dealt with in the section 5.4.2. The solution was to add *AWS IAM* permissions in the *YAML* template configuration file. The solution was inspired

by a Tobias Sturm's blog post [173] and by the *kops* documentation[79].

The fourth problem was with testing the Apache Server. The essential thing to know is that it takes some time (around 5 minutes) for the ELB to be usable by end users and thus the tests may fail in the meantime. The solution was to implement such a command that verifies the Apache Server endpoint many times, but has a maximum number of trials.

There were also **problems with deploying Velero**. For instance, using the official example deployment command provided on the *Velero Helm Chart* [175] resulted in error. It was caused by omitting specifying the aws plugin. The error is shown in listing 5.46.

```
$ helm repo add vmware-tanzu https://vmware-tanzu.github.io/helm-charts
# output omitted
$ helm install velero --namespace default \
  --set configuration.provider=aws \
  --set configuration.backupStorageLocation.name=aws \
  --set configuration.backupStorageLocation.bucket=${K8S_EXP_KOPS_S3_BUCKET} \
  --set configuration.backupStorageLocation.prefix=${K8S_EXP_ENVIRONMENT} \
  --set configuration.backupStorageLocation.config.region=${K8S_EXP_REGION} \
  --wait --atomic \
  vmware-tanzu/velero
# output omitted
$ kubectl logs -n default velero-7f476777f6-f9qlc
An error occurred: some backup storage locations are invalid: error getting\
backup store for location "aws": unable to locate ObjectStore plugin named velero.io/aws
```

Listing 5.46: Installing Velero server

Tinkering with that command resulted also in different error, presented in listing 5.47.

```
Error: release velero failed , and has been uninstalled due to atomic being set:\
timed out waiting for the condition
```

Listing 5.47: Velero server installation error

It was then decided to try another command, shown in listing 5.48, to install *Velero* server. It pushed the progress further and revealed the underlying problem. The problem was with scheduling the *Velero* pod.

```
$ velero install \
  --provider velero.io/aws \
  --bucket ${K8S_EXP_KOPS_S3_BUCKET} \
  --plugins velero/velero-plugin-for-aws:v1.1.0 \
  --backup-location-config s3Url=${K8S_EXP_KOPS_S3_BUCKET},region=${K8S_EXP_REGION} \
```

```

—use-volume-snapshots=false \
—secret-file=${PWD}/credentials-velero
Velero is installed! Use 'kubectl logs deployment/velero -n velero' to view the status.
$ kubectl get -n velero all
NAME                                READY   STATUS    RESTARTS   AGE
pod/velero-5f8889f694-6xd7t        0/1     Pending   0           2m30s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/velero             0/1     1             0           2m30s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/velero-5f8889f694  1         1         0       2m30s
$ kubectl -n velero describe pods
Name:                velero-5f8889f694-6xd7t
Namespace:           velero
Limits:
  cpu:                1
  memory:             256Mi
Requests:
  cpu:                500m
  memory:             128Mi
Warning FailedScheduling 70s (x5 over 6m46s) default-scheduler 0/2 nodes\
are available: 2 Insufficient cpu.

```

Listing 5.48: Another attempt to deploy *Velero* server

The problem was that the *Velero* pod was in status *Pending*. The Kubernetes documentation on pods troubleshooting suggests, that when a pod is in status *Pending*, then there may be not enough resources in the cluster [123]. That was the problem in this case. The solution was to use bigger AWS instance types for node: *t2.medium* instead of: *t2.micro*.

Furthermore, it may be helpful to know, that whenever an error presented in listing 5.49 occurs, then, it is due to invalid credentials file. It should be possible to list access keys of AWS IAM user created for *Velero*. This can be tested with the command in listing 5.50.

```

$ kubectl get pod -n velero
NAME                                READY   STATUS             RESTARTS   AGE
velero-6fffc8dc85-cm4lb            0/1     CrashLoopBackOff   3           107s
$ kubectl logs -n velero velero-6fffc8dc85-cm4lb
An error occurred: some backup storage locations are invalid: backup store for\
  location "aws" is invalid: rpc error: code = Unknown desc = AccessDenied: \
  Access Denied

```

```
status code: 403, request id: 9513047CEE2BD3, host id:\
FjQ4yAzAl1zhyFXRIS67Ww3rSqt2a1fqoeLzYEAAdKsRLqx3AmpEzWWIoJZXZPx9rKJs9qCH1yHY=
```

Listing 5.49: Debugging *Velero*

Then, it is due to invalid credentials file. It should be possible to list access keys of AWS IAM user created for *Velero*. This can be tested with the following command:

```
$ aws iam list-access-keys --user-name velero-${K8S_EXP_CLUSTER_NAME}
{
  "AccessKeyMetadata": [
    {
      "UserName": "velero-kops-testing",
      "AccessKeyId": "<REPLACE-ME>",
      "Status": "Active",
      "CreateDate": "2020-06-03T10:22:46+00:00"
    }
  ]
}
```

Listing 5.50: Listing AWS IAM keys

A **problem considering autoscaler deployment** was observed. The Kubernetes resources provided by the official git repository on *Github.com* had a minor mistake. The autoscaler pod was in error state and it returned the error presented in listing 5.51:

```
$ kubectl describe -n kube-system pod/cluster-autoscaler-8b46dddf5-8xkns
Warning   Failed          26s      kubelet, ip-172-20-62-245.eu-west-1.compute.internal \
Error: failed to start container "cluster-autoscaler": \
Error response from daemon: OCI runtime create failed: container_linux.go:346:\
starting container process caused "process_linux.go:449: _container_init_caused\
_\\"rootfs_linux.go:58: mounting \\"/etc/ssl/certs/ca-bundle.crt\\" to rootfs \
_\\"/var/lib/docker/overlay2/<long-hash>/merged\\" at \
_\\"/var/lib/docker/overlay2/<long-hash>/merged/etc/ssl/certs/ca-certificates.crt\\" \
caused \\"not_a_directory\\"\"": unknown: \
Are you trying to mount a directory onto a file (or vice-versa)? \
Check if the specified host path exists and is the expected type
Warning   BackOff          12s (x2 over 42s) kubelet, ip-172-20-62-245.eu-west-1.compute.internal \
Back-off restarting failed container
```

Listing 5.51: Debugging autoscaler

Fortunately, this problem was handled by a public issue and the solution was also provided there [111]. The solution was to replace a path: */etc/ssl/certs/ca-certificates.crt* with */etc/ssl/certs/ca-bundle.crt*.

Also, running the *kops create cluster* initially resulted in error demonstrated in listing 5.52.

```
error building tasks: error remapping manifest \
  addons/kops-controller.addons.k8s.io/k8s-1.16.yaml: \
error parsing yaml: error converting YAML to JSON: yaml: line 56: \
  did not find expected alphabetic or numeric character
```

Listing 5.52: Error output of *kops create cluster* command

The reason for the error in listing 5.52 was that the development environment had a non-numeric environment variable set. It was a password and the variable value contained asterisks (****). After the variable was unset (with the *Bash* command: *unset*), the *kops create cluster* succeeded.

5.5 Production deployment using *eksctl*

This subchapter briefly presents all the steps performed that lead to a Kubernetes cluster deployment on the AWS cloud using eksctl. Here an attempt was made to satisfy all the production environment requirements selected in the chapter 4.

5.5.1 Generating the YAML configuration file

When using *eksctl*, it is possible to create a cluster with *eksctl create cluster* command and either with setting command line options or with using a YAML file. The latter solution was chosen. In contrast to *kops*, it was easier to create such YAML file by hand. Due to the fact that it was desired to automate the cluster deployment across multiple environments, it was decided that there will be one configuration file named: *cluster.tmpl.yaml*. This file will contain *Bash* variables as some values. This file will be used as a template. Then, to create a configuration file for production environment, such commands were run (listing 5.53).

```
$ export K8S_EXP_REGION="eu-west-1"
$ export K8S_EXP_ENVIRONMENT="production"
$ export K8S_EXP_CLUSTER_NAME="eks-${K8S_EXP_ENVIRONMENT}"
$ my_ip=$(curl https://ipinfo.io/ip 2>/dev/null)
$ sed "s/\${K8S_EXP_CLUSTER_NAME}/${K8S_EXP_CLUSTER_NAME}/" cluster.tmpl.yaml >\
```



```

cluster1.yaml
$ sed "s/\${K8S_EXP_REGION}/${K8S_EXP_REGION}/" cluster1.yaml > cluster2.yaml
$ sed "s/\${K8S_EXP_ENVIRONMENT}/${K8S_EXP_ENVIRONMENT}/" cluster2.yaml > cluster3.yaml
$ sed "s'\${my_ip}'\${my_ip}/32'" cluster3.yaml > cluster.yaml
$ rm cluster1.yaml cluster2.yaml cluster3.yaml
$ time eksctl create cluster -f cluster.yaml

```

Listing 5.53: Creating *eksctl* configuration

All these commands were scripted and thanks to the automation, they can be invoked just by running a single command, demonstrated in listing 5.54.

```
eks$ ./tasks _create
```

Listing 5.54: Creating a Kubernetes cluster with *eksctl*

This command creates a configuration file for a particular environment and then it deploys a Kubernetes cluster. The whole *cluster.tmpl.yaml* file is presented in listing 5.55.

```

apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: eks-testing
  region: eu-west-1
  version: "1.16"
  tags:
    deployment: eks-testing

cloudWatch:
  clusterLogging:
    enableTypes: ["api", "audit", "authenticator", "controllerManager", "scheduler"]

nodeGroups:
  - name: ng-1
    labels: { role: worker, cluster: eks-testing }
    instanceType: t2.nano
    desiredCapacity: 1
    ssh:
      allow: true

```

Listing 5.55: *Eksctl* configuration file

Thanks to *metadata.tags* set in the YAML file, the AWS resources were labeled with a custom tag: *deployment: eks-testing* or *deployment: eks-production* (depending on the

environment). Figures 5.9 and 5.10 present a CloudFormation and EKS resource with tags, viewed on AWS Management Console.

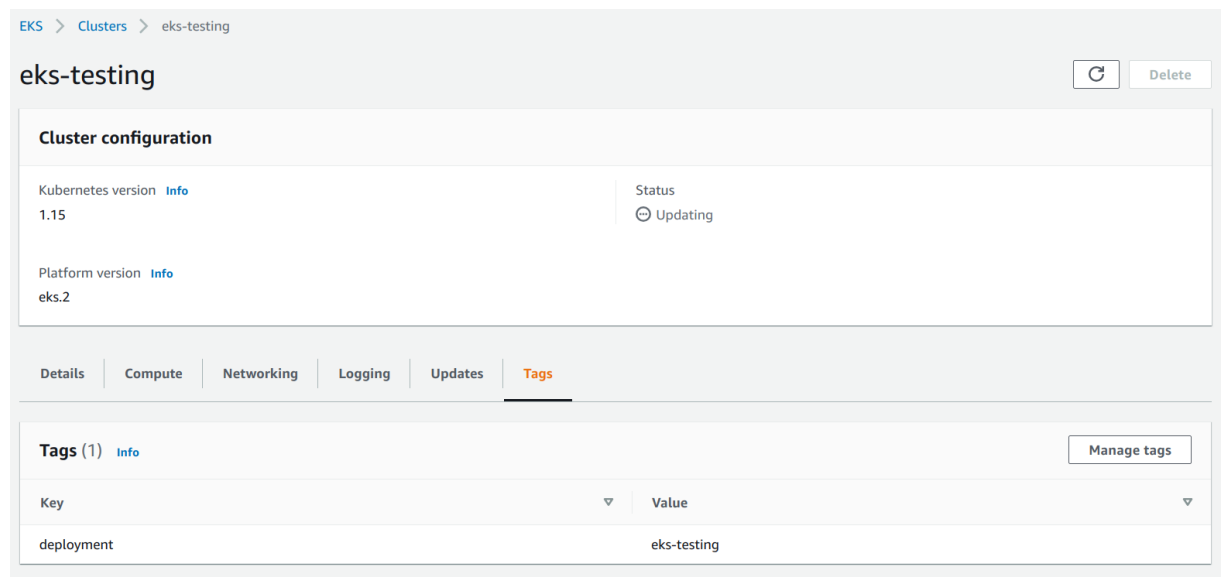


Fig. 5.9: AWS EKS resource with tags set, viewed on AWS Management Console

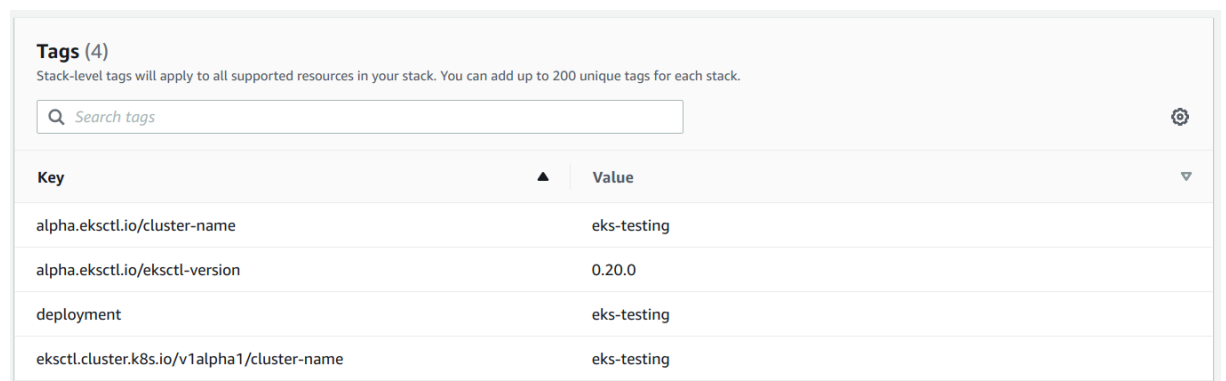
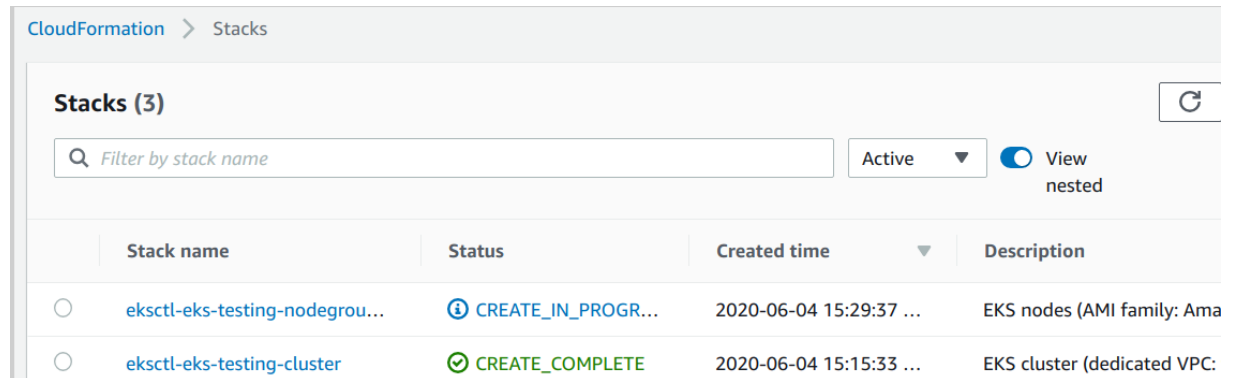


Fig. 5.10: AWS CloudFormation resource with tags set, viewed on AWS Management Console

5.5.2 Creating a cluster

There was one command used to generate cluster configuration (described in the previous subchapter) and to deploy the cluster. The nice thing from the end user point of view was, that this command did not return (meaning: it waited) for the cluster to be ready.

First, the control plane was deployed, then the worker nodes. For each of these two tasks (deploying control plane and deploying worker nodes) a separate CloudFormation stack was used. It was done automatically by AWS EKS. Figure 5.11 presents such two stacks.

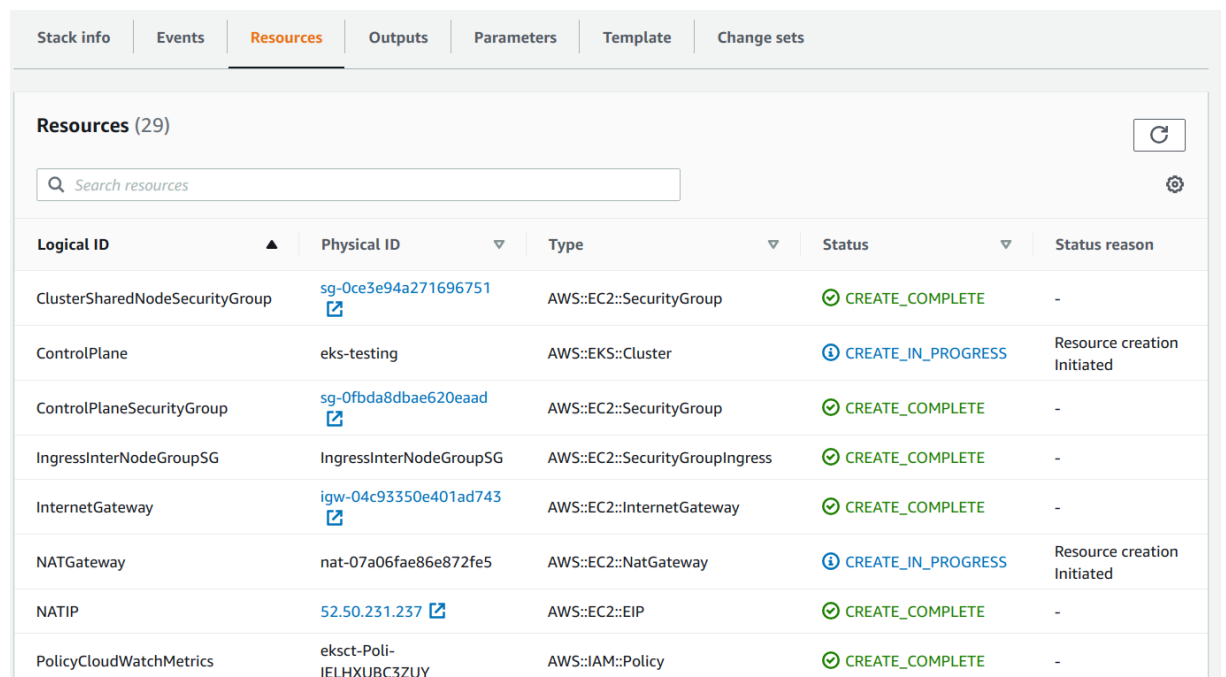


The screenshot shows the AWS CloudFormation 'Stacks' page. There are two stacks listed:

Stack name	Status	Created time	Description
eksctl-eks-testing-nodegroup...	CREATE_IN_PROGR...	2020-06-04 15:29:37 ...	EKS nodes (AMI family: Ama
eksctl-eks-testing-cluster	CREATE_COMPLETE	2020-06-04 15:15:33 ...	EKS cluster (dedicated VPC:

Fig. 5.11: CloudFormation stacks needed by *eksctl*

It was also possible to view the AWS resources which were provided by each CloudFormation stack. It is illustrated in figure 5.12.



The screenshot shows the 'Resources' tab of a CloudFormation stack. It lists 29 resources with the following columns: Logical ID, Physical ID, Type, Status, and Status reason.

Logical ID	Physical ID	Type	Status	Status reason
ClusterSharedNodeSecurityGroup	sg-0ce3e94a271696751	AWS::EC2::SecurityGroup	CREATE_COMPLETE	-
ControlPlane	eks-testing	AWS::EKS::Cluster	CREATE_IN_PROGRESS	Resource creation Initiated
ControlPlaneSecurityGroup	sg-0fbda8dbae620eaad	AWS::EC2::SecurityGroup	CREATE_COMPLETE	-
IngressInterNodeGroupSG	IngressInterNodeGroupSG	AWS::EC2::SecurityGroupIngress	CREATE_COMPLETE	-
InternetGateway	igw-04c93350e401ad743	AWS::EC2::InternetGateway	CREATE_COMPLETE	-
NATGateway	nat-07a06fae86e872fe5	AWS::EC2::NatGateway	CREATE_IN_PROGRESS	Resource creation Initiated
NATIP	52.50.231.237	AWS::EC2::EIP	CREATE_COMPLETE	-
PolicyCloudWatchMetrics	eksct-Poli-IELHXUBC3ZUY	AWS::IAM::Policy	CREATE_COMPLETE	-

Fig. 5.12: CloudFormation resources needed by *eksctl*

The control plane is entirely managed by AWS and it is run across three AWS availability zones in order to ensure high availability. The end user has even no access to the

control plane, meaning: there is no *EC2 instance* with master node visible and when listing all Kubernetes nodes, only the worker nodes are visible (Fig. 5.13).

<input type="checkbox"/>	Name	deployment	Instance ID	Instanc	Availability
<input type="checkbox"/>	eks-testing-ng-1-Node		i-073f6b5ddff4949b2	t2.nano	eu-west-1b

Fig. 5.13: AWS EKS instances, no master nodes

A few sources claimed that the creation of Kubernetes cluster on *AWS EKS* should take 10 - 15 minutes [164, 189], but in this particular case it took about 20 minutes. Also, to the best of this work's author knowledge, there was no command which allows to reconfigure the cluster using all settings from the *YAML* configuration file. This means that when there was a need to change e.g. tags of the *AWS* resources, the cluster must have been deleted and created from scratch. An issue on *Github.com* was created to acknowledge this problem [102]. This cannot be achieved with *eksctl*, but many settings can be changed with *AWS CLI*, for example cluster endpoint [35]. There is, however, a command that enables to upgrade the control plane to the next Kubernetes version (if it available). This command is presented in listing 5.56.

```
$ eksctl update cluster
```

Listing 5.56: Updating *eksctl* cluster

And also, some particular settings may be set using *eksctl utils* command, for example [186] the ones demonstrated in listing 5.57.

```
$ eksctl utils set-public-access-cidrs --cluster=<cluster> 1.1.1.1/32,2.2.2.0/24
$ eksctl utils set-public-access-cidrs -f config.yaml
```

Listing 5.57: Updating *eksctl* cluster configuration

Enabling the Central Logging was very easy. All what was needed was adding the following code to the configuration *YAML* file (listing 5.58).

```
cloudWatch:
  clusterLogging:
    enableTypes: ["api", "audit", "authenticator", "controllerManager", "scheduler"]
```

Listing 5.58: Enabling logging in *eksctl* cluster

In result, an AWS LogGroup was created and all the log messages from Kubernetes components were directed onto the LogGroup (Fig. 5.14 and 5.15).

The screenshot shows the AWS CloudWatch console for the log group `/aws/eks/eks-testing/cluster`. It features a 'Delete' button in the top right. Below the breadcrumb, there's a section for 'Log group details'. A navigation bar includes 'Log streams' (selected), 'Metric filters', and 'Contributor Insights'. The 'Log streams (10)' section contains a search bar and a table of log streams.

<input type="checkbox"/>	Log stream	Last event time
<input type="checkbox"/>	kube-scheduler-e7011cb46cc6cbfea440b167fa7362f6	6/4/2020, 7:50:25 PM
<input type="checkbox"/>	kube-controller-manager-e7011cb46cc6cbfea440b167fa7362f6	6/4/2020, 7:49:01 PM
<input type="checkbox"/>	authenticator-fcccb5324486a39d7d35b045b740316d	6/4/2020, 7:41:53 PM
<input type="checkbox"/>	kube-apiserver-e7011cb46cc6cbfea440b167fa7362f6	6/4/2020, 7:40:51 PM

Fig. 5.14: AWS CloudWatch logs for `eksctl` cluster

The screenshot shows the AWS CloudWatch console for the log stream `kube-apiserver-audit-e7011cb46cc6cbfea440b167fa7362f6` within the log group `/aws/eks/eks-testing/cluster`. It includes a 'Try CloudWatch Logs Insights' banner. The 'Log events' section has a search bar and a table of log events.

	Timestamp	Message
		There are older events to load. Load more.
▶	2020-06-04T20:00:35.463+02:00	<code>{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"095c3f"}</code>
▶	2020-06-04T20:00:35.463+02:00	<code>{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"ac7f2f"}</code>
▶	2020-06-04T20:00:35.463+02:00	<code>{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"5ef986"}</code>

Fig. 5.15: AWS CloudWatch logs for `eksctl` cluster – concerning API server

An example log message in JSON format is contained in listing 5.59.

```
2020-06-04T20:00:36.463+02:00
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Metadata",
  "auditID": "454b8586-2d17-4a53-9eb5-1e7422aabd21",
  "stage": "ResponseComplete",
  "requestURI": "/api?timeout=32s",
  "verb": "get",
  "user": {
    "username": "system:serviceaccount:kube-system:resourcequota-controller",
    "uid": "b7f3c725-196f-4b25-94e4-8f75614d3a04",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:kube-system",
      "system:authenticated"
    ]
  },
  "sourceIPs": [
    "10.0.162.1"
  ],
  "userAgent": "kube-controller-manager/v1.16.8 (linux/amd64) \
    kubernetes/e163110/system:serviceaccount:kube-system:resourcequota-controller",
  "responseStatus": {
    "metadata": {},
    "code": 200
  },
  "requestReceivedTimestamp": "2020-06-04T18:00:36.279592Z",
  "stageTimestamp": "2020-06-04T18:00:36.279722Z",
  "annotations": {
    "authorization.k8s.io/decision": "allow",
    "authorization.k8s.io/reason": "RBAC: allowed by ClusterRoleBinding \
      \"system:discovery\" of ClusterRole \"system:discovery\" to Group \
      \"system:authenticated\""
  }
}
```

Listing 5.59: Example log message

There was nothing needed to be done in order to provide High Availability. The control plane was already Highly Available because AWS EKS automatically provisions and scales the control plane across multiple AWS availability zones for high availability and fault tolerance [37]. Even the output provided when creating the cluster confirms the *High Availability* (listing 5.60).

```

+ eksctl create cluster -f cluster.yaml
[info] eksctl version 0.20.0
[info] using region eu-west-1
[info] setting availability zones to [eu-west-1a eu-west-1b eu-west-1c]
[info] subnets for eu-west-1a - public:192.168.0.0/19 private:192.168.96.0/19
[info] subnets for eu-west-1b - public:192.168.32.0/19 private:192.168.128.0/19
[info] subnets for eu-west-1c - public:192.168.64.0/19 private:192.168.160.0/19

```

Listing 5.60: Output from creating a *eksctl* cluster

Central Monitoring system was also already available. AWS provides several metrics considering *EC2 instances*. The same metrics can be viewed in this subchapter as in the according subchapter for *kops* cluster. All the available metrics are presented in figure 5.16.

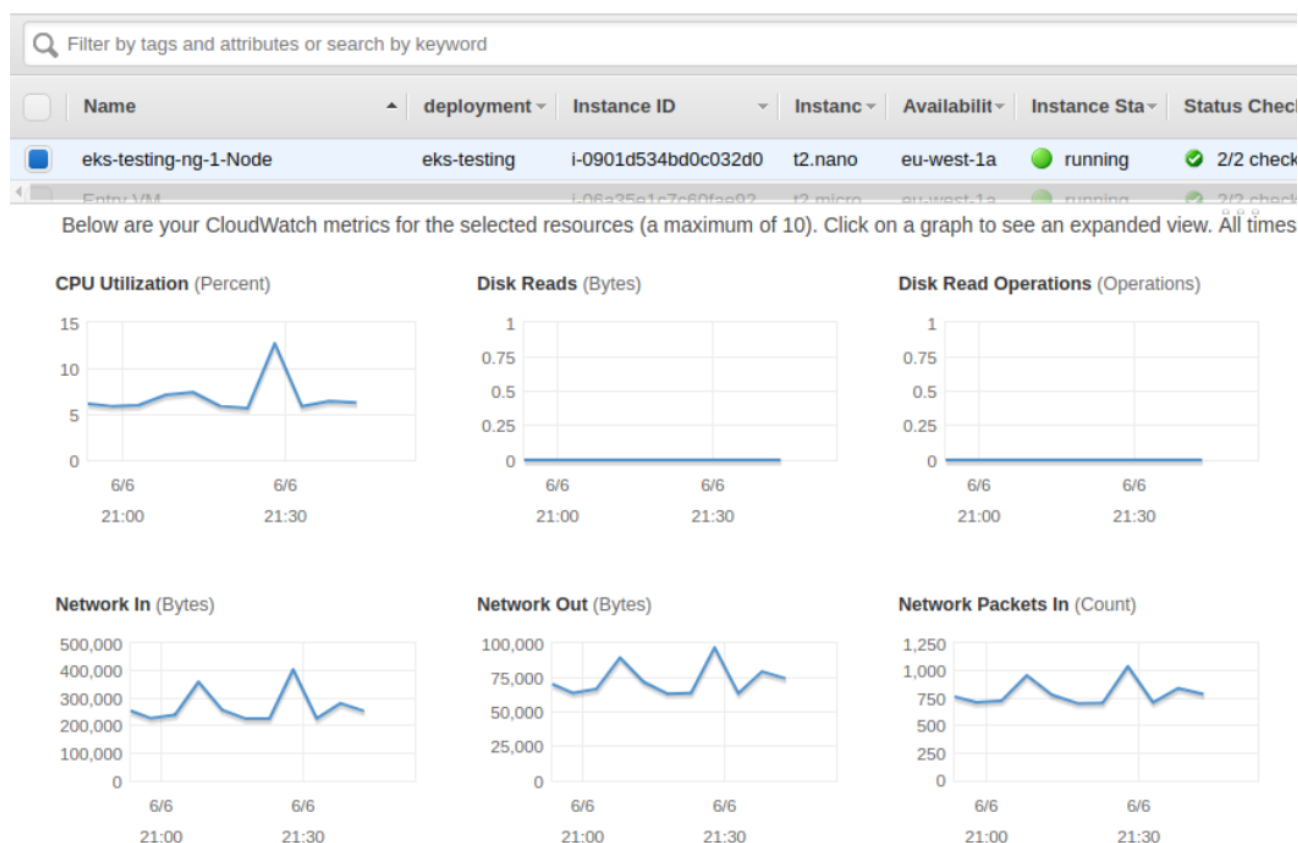


Fig. 5.16: AWS *CloudWatch* monitoring for a worker node

When it comes to Central Audit, also no additional steps were taken. The actions performed on AWS resources are automatically visible on *AWS CloudTrail* dashboard. This is

the same case as with *kops*. Since both methods use AWS, they both utilize *AWS CloudTrail*. An example view showing the *AWS CloudTrail* events, which occurred due to deleting an EKS cluster is shown in figure 5.17.

The screenshot shows the AWS CloudTrail console. At the top, there's a filter bar with 'Read only' selected and a time range selector. Below this is a table of events. The selected event is a 'DeleteSubnet' event from 2020-06-04 at 08:15:17 PM, performed by 'ewa-admin' on an 'EC2 Subnet' resource named 'subnet-07dbe961f15c5fe4f'.

Event time	User name	Event name	Resource type	Resource name
2020-06-04, 08:15:34 PM	ewa-admin	DeleteVpc	EC2 VPC	vpc-0fc46b7e13fe6b77a
2020-06-04, 08:15:17 PM	ewa-admin	DeleteSubnet	EC2 Subnet	subnet-0d2acd1854477379a
2020-06-04, 08:15:17 PM	ewa-admin	DeleteSecurityGroup	EC2 SecurityGroup	sg-0417fe0b9101f902c
2020-06-04, 08:15:17 PM	ewa-admin	DeleteSubnet	EC2 Subnet	subnet-07dbe961f15c5fe4f

AWS access key AWS region eu-west-1 Error code Event ID 927249b0-8bb4-4ae7-b145-b3d73bdae5c7 Event name DeleteSubnet Event source ec2.amazonaws.com	Event time 2020-06-04, 08:15:17 PM Read only false Request ID 4a8a4fdd-88c8-4651-9905-637b570f718d Source IP address cloudformation.amazonaws.com User name ewa-admin
--	--

Resources Referenced (1)

Resource type	Resource name	Config timeline
EC2 Subnet	subnet-07dbe961f15c5fe4f	<⌂

View event

Fig. 5.17: *AWS CloudTrail* events concerning *eksctl* cluster

In order to make the cluster more secure, it was decided to restrict the *SSH* access and *kubectl* access. Searching through *eksctl* configuration, there was no setting which allowed to whitelist the IP addresses which are allowed to ssh login to worker nodes. However, blocking all the *SSH* access whatsoever was available and it was chosen (listing 5.61).

```
nodeGroups :
  - name: ng-1
    privateNetworking: true
    ssh :
      allow: false
```

Listing 5.61: *Eksctl* configuration applying security measures

This means now, that a cluster administrator has only *kubectl* access to cluster and no ssh access at all (because ssh access to worker nodes is blocked and access to master nodes is not given by design of *AWS EKS*). It is acknowledged that sometimes, not being able to ssh to the node may make troubleshooting more difficult, but generally system and *EKS* logs generally contain enough information for diagnosing problems. Usually the solution is to replace the problematic worker node rather than to diagnose and fix it [101]. Since no access to nodes is needed, thus also *privateNetworking* under the *nodeGroups* array

was set to *true*.

The second security requirement was to restrict the access to cluster through *kubect*. By default, *eksctl* exposes the Kubernetes API server publicly but not directly from within the VPC subnets. This means that the public endpoint is by default set to true and the private endpoint to false [186]. In this work it was decided to apply the configuration presented in listing 5.62.

```
vpc:
  publicAccessCIDRs:
    - "<MY_PUBLIC_IP>/32"
  clusterEndpoints:
    publicAccess: true
    privateAccess: true
```

Listing 5.62: *Eksctl* configuration applying security measures

There were problems finding a working configuration, but it was finally solved thanks to a blog post [155] and thanks to *eksctl* official documentation [186]. The problems are described also in the subchapter 5.5.5. The main setting was *publicAccessCIDRs*. It was set to the this-work-author's IP. Thus, only one IP address should be allowed to communicate with the cluster through *kubect*. It was tested that the command *kubect* worked from such an IP (listing 5.63).

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
ip-192-168-126-229.eu-west-1.compute.internal	Ready	<none>	2m31s	v1.16.8-eks-e16311

Listing 5.63: Verifying connection to API server

When the same command was run from a different machine, with a different IP, this command reached a timeout. Whenever there is a timeout when reaching an AWS endpoint, it may be caused by a Security Group not allowing such access. Thus, this proved that the *kubect* access was restricted and, that security measures were working. Creating the EKS cluster with all the security settings took longer – about 25 minutes. But it was successful.

In order to meet the production deployment requirement of backup, *Velero* was used. The identical use case was tested as with the *kops* cluster. The result was identical, thus, there is no sense in describing it for the second time. Both: backup and restore operations went smoothly. But, one thing must be noted. An S3 bucket was needed as *Velero*

backups destination. Thus, the bucket was created with commands presented in listing 5.64.

```
$ export K8S_EXP_VELERO_S3_BUCKET="k8s-eks-for-masters-thesis.k8s.local"
$ aws s3api create-bucket --bucket ${K8S_EXP_VELERO_S3_BUCKET} --region ${K8S_EXP_REGION} \
  --create-bucket-configuration LocationConstraint=${K8S_EXP_REGION}
$ aws s3api put-bucket-versioning --bucket ${K8S_EXP_VELERO_S3_BUCKET} \
  --versioning-configuration Status=Enabled
$ aws s3api put-bucket-encryption --bucket ${K8S_EXP_VELERO_S3_BUCKET} \
  --server-side-encryption-configuration '{"Rules":[{"ApplyServerSideEncryptionByDefault":{"SSEAlgorithm
```

Listing 5.64: Creating an S3 bucket for *Velero*

The backup created by *Velero* was put in the S3 bucket, which is presented in figure 5.18:

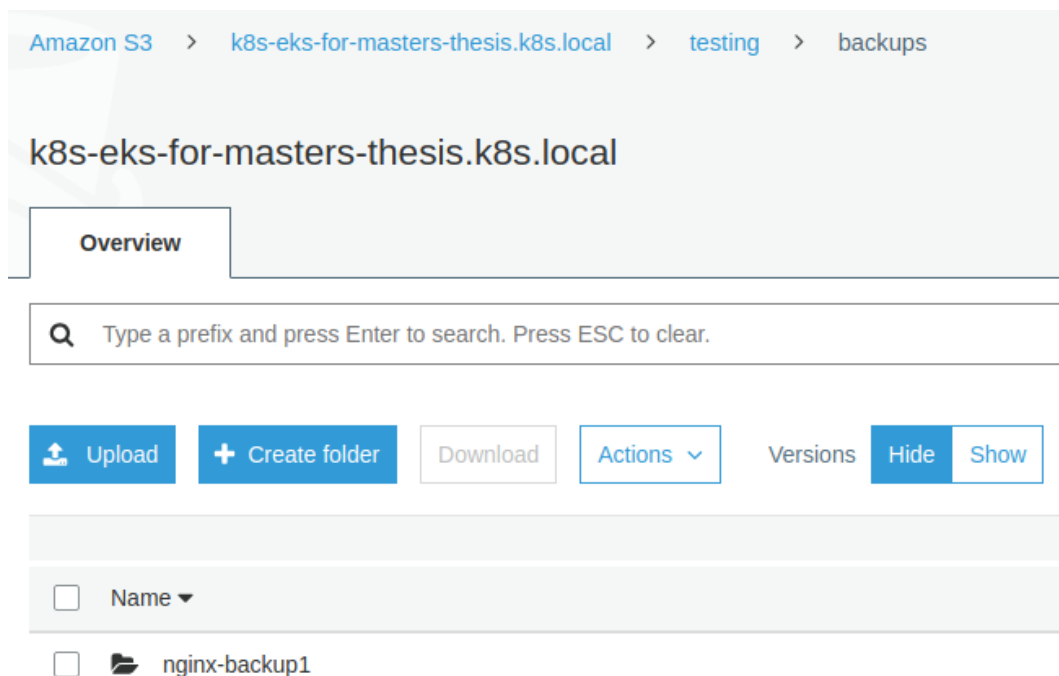


Fig. 5.18: Backup of *eksctl* cluster created by *Velero*

The next step was to deliver the autoscaling requirement. The same Kubernetes resources were used as with the *kops* method. This part of cluster configuration was needed to be put under the *nodeGroups* YAML array (listing 5.65).

```
tags:
  # EC2 tags required for cluster-autoscaler auto-discovery
  k8s.io/cluster-autoscaler/enabled: "true"
  k8s.io/cluster-autoscaler/eks-testing: "owned"
iam:
```

```

withAddonPolicies:
  autoScaler: true
  albIngress: true
  cloudWatch: true

```

Listing 5.65: *Eksctl* configuration needed for autoscaler

Afterwards, the same procedure as described with *kops* was used. In result, deploying a test application and scaling its replicas number to 20 resulted in autoscaler adding another *EC2 instance* - a worker node. All the worker nodes were listed with the commands provided in listing 5.66.

```

eks$ aws ec2 describe-instances --filters "Name=tag-key,Values=deployment" --query\
  "Reservations[*].Instances[*].{PublicIP:PublicIpAddress,Name:Tags\
    .[?Key=='Name']|[0].Value,Status:State.Name}"
[
  [
    {
      "PublicIP": null,
      "Name": "eks-testing-ng-1-Node",
      "Status": "running"
    }
  ],
  [
    {
      "PublicIP": null,
      "Name": "eks-testing-ng-1-Node",
      "Status": "running"
    }
  ]
]
eks$ kubectl get node

```

NAME	STATUS	ROLES	AGE	VERSION
ip-192-168-111-191.eu-west-1.compute.internal	Ready	<none>	67s	v1.16.8-eks-e16311
ip-192-168-162-84.eu-west-1.compute.internal	Ready	<none>	10m	v1.16.8-eks-e16311

Listing 5.66: Listing *EC2 instances*, one was created by autoscaler

Afterwards, the testing application was uninstalled with *Helm* and after more than 5 minutes, the worker node *EC2 instance* was automatically removed.

5.5.3 Testing a cluster

The testing procedure was the same as with *kops*. Different command was invoked (because all the commands to run with *eksctl* were put into one file), but all the *Bats-core* tests were the same. Even the same *Bats-core* files were used. There was obviously no *kops validate cluster* command run, because that command was dedicated to *kops* clusters only. For the details, please refer to section 5.4.3. Running all the tests took slightly less than 7 minutes. It took long to have a working Load Balancer. The command to run tests was presented in listing 5.67.

```
eks$ ./tasks _test
```

Listing 5.67: Testing an *eksctl* cluster

5.5.4 Deleting a cluster

In order to delete the cluster, the command shown in listing 5.68 has to be entered.

```
eks$ ./tasks _delete
```

Listing 5.68: Deleting an *eksctl* cluster

This command usually took about 13 minutes. It deleted almost all AWS resources, but not *CloudWatch LogGroup*. Not deleting the *LogGroup* may be a good thing, because one may be interested to study the logs after the cluster is long gone. But still, it is good to know, that one has to delete it by themselves.

5.5.5 Troubleshooting a cluster

The same issues apply here, considering *Velero*, as with a cluster created with *kops*.

The greatest problem was how to set security measures. Adding to the *YAML* configuration file the lines presented in listing 5.69 resulted in timeout on creating such cluster. It took 43 minutes then. The output with the error returned by the command creating a cluster is presented in listing 5.70 .

```
vpc:
  publicAccessCIDs:
    - "<MY_PUBLIC_IP>/32"
```

Listing 5.69: Attempt to setup security

```
[info] adding identity "arn:aws:iam::976184668068:role/\
_eksctl-eks-testing-nodegroup-ng-1-NodeInstanceRole-8VB5IDO1Z4KQ" to auth ConfigMap
[info] nodegroup "ng-1" has 0 node(s)
[info] waiting for at least 1 node(s) to become ready in "ng-1"
Error: timed out (after 25mos) waiting for at least 1 nodes to join the cluster\
and become ready in "ng-1"
```

Listing 5.70: Error output from creating *eksctl* cluster

This problem was also acknowledged in *eksctl Github.com* issues [145, 152]. Even though the blog post [155], which tremendously helped to find the working solution, suggested setting the following IAM policies to be configured under the *nodeGroups* array, in practice, the IAM policies deemed unneeded. Cluster is working and healthy without them (listing 5.71).

```
nodeGroups:
  iam:
    attachPolicyARNs:
      - arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy
      - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
      - arn:aws:iam::aws:policy/ElasticLoadBalancingFullAccess
```

Listing 5.71: AWS IAM policies recommended for *eksctl* by a blog post

When deploying Cluster Autoscaler, there was a problem. The Cluster Autoscaler pod was in status *Pending*. This is the same case as with the *Velero* Pod, described in section 5.4.5. In this case there was too little memory. The *EC2 instance types* which were used initially, were *t2.nano*. It was decided to try *t2.micro* instances type. The problem was debugged with command presented in listing 5.72.

```
eks$ kubectl get -n kube-system pod
NAME                                READY   STATUS    RESTARTS   AGE
aws-node-9sttv                     1/1     Running   0           61m
cluster-autoscaler-7d7b5564df-r6sqh 0/1     Pending   0           27s
coredns-8568c98b77-d7txg           0/1     Pending   0           74m
coredns-8568c98b77-hpmjk           1/1     Running   0           74m
kube-proxy-4qplq                   1/1     Running   0           61m
$ kubectl describe -n kube-system pod/cluster-autoscaler-7d7b5564df-r6sqh
# some output omitted
Events:
```

Type	Reason	Age	From	Message
Warning	FailedScheduling	66s (x2 over 66s)	default-scheduler	0/1 nodes are available: 1 Insufficient

Listing 5.72: Debugging cluster autoscaler

Using the *t2.micro* instances type, there was a problem with deploying the test application (listing 5.73).

```
$ kubectl -n testing describe pod/apache-testing-77594c54fb-fsc5s
# some output omitted
Events:
  Type      Reason          Age             From           Message
  ----      -
  Warning   FailedScheduling  38s (x10 over 12m)  default-scheduler \
  o/1 nodes are available: 1 Insufficient pods.
```

Listing 5.73: Debugging a test application

The potential solution, found on Stack Overflow, suggested that *t2.small* is the smallest instance type which is acceptable [140]. It turned out, that this problem was caused by an already described fact, that AWS EKS uses hard limits for number of pods allowed for a particular *EC2 instance type* [59]. Thus, even bigger instance was used: *t2.small*. After the bigger instance type was used, all the pods in *kube-system* namespace were in status *Running* (listing 5.74).

```
eks$ kubectl get pod -n kube-system
NAME                                READY   STATUS    RESTARTS   AGE
aws-node-dk457                      1/1     Running   0           3m38s
coredns-8568c98b77-gf5r4           1/1     Running   0           16m
coredns-8568c98b77-xg7jv           1/1     Running   0           16m
kube-proxy-hgjml                    1/1     Running   0           3m38s
eks$ kubectl get pod -n testing
NAME                                READY   STATUS    RESTARTS   AGE
apache-testing-77594c54fb-nl5b2     1/1     Running   0           117s
```

Listing 5.74: Verifying that all the pods are running

Once, there was also a problem with deleting the EKS cluster. This was probably due to an unusual order of Kubernetes cluster operations performed. Usually, EKS was operated in the way presented in listing 5.75.

```
eks$ eksctl create cluster -f cluster.yaml
eks$ eksctl delete cluster -f cluster.yaml --wait
```

Listing 5.75: Deleting *eksctl* cluster

This particular time, which lead to an error, it was decided to delete a *worker nodegroup* with the intension to recreate it with different configuration. Therefore, this particular time, the cluster was operated in a different way and resulted in the output presented in listing 5.76.

```
eks$ eksctl create cluster -f cluster.yaml
# output omitted
eks $ eksctl delete nodegroup -f cluster.yaml --approve
# some output omitted
[info] will delete stack "eksctl-eks-testing-nodegroup-ng-1"
[ok] deleted 1 nodegroup(s) from cluster "eks-testing"
eks$ eksctl delete cluster -f cluster.yaml --wait
# some output omitted
[ok] 2 sequential tasks: { delete nodegroup "ng-1", delete cluster control plane\
    "eks-testing" }
[ok] will delete stack "eksctl-eks-testing-nodegroup-ng-1"
[ok] waiting for stack "eksctl-eks-testing-nodegroup-ng-1" to get deleted
[no] unexpected status "DELETE_FAILED" while waiting for CloudFormation stack\
    "eksctl-eks-testing-nodegroup-ng-1"
[ok] fetching stack events in attempt to troubleshoot the root cause of the failure
[no] AWS::CloudFormation::Stack/eksctl-eks-testing-nodegroup-ng-1: \
    DELETE_FAILED      "The following resource(s) failed to delete: [SG]. "
[no] AWS::EC2::SecurityGroup/SG: DELETE_FAILED      "resource sg-03f9a9928f765ecae\
    has a dependent object (Service: AmazonEC2; Status Code: 400; Error Code:\
    DependencyViolation; Request ID: 682fd971-5ec7-472e-9eee-4d52a44f9455)"
[ok] 1 error(s) occurred while deleting cluster with nodegroup(s)
[no] waiting for CloudFormation stack "eksctl-eks-testing-nodegroup-ng-1"\
    : ResourceNotReady: failed waiting for successful resource state
Error: failed to delete cluster with nodegroup(s)
```

Listing 5.76: *Eksctl* special operations

Fortunately, running the *eksctl delete cluster* again resulted in success.

5.6 Troubleshooting any Kubernetes cluster

This is a subchapter dedicated to present the ideas of how to troubleshoot any Kubernetes cluster. The ideas were gathered during the empirical work of this study.

There are some general guidelines which help to debug the cluster. When a cluster is self-hosted and one is fully responsible for master and worker nodes deployment, then **one should verify whether any node is still functional**. It means that problems like: corrupt filesystem, kernel deadlock, problems with the Docker Daemon or problems with hardware may occur and should be mitigated. A solution to detect such problems may be: a *node problem detector pod* [14]. One should also run the following commands to check if all the worker nodes are running and if the cluster is generally reachable. But, since this work focuses on deployments on the cloud, then the problems described in this paragraph should be already taken care of:

```
$ kubectl get nodes
$ kubectl cluster-info
```

Listing 5.77: Getting information about a cluster

Another level of troubleshooting is **debugging the applications deployed on top of the Kubernetes cluster**. Some ideas are described on the official Kubernetes documentation website [123]. For example, one can get details of a pod with the following commands:

```
$ kubectl describe pods ${POD_NAME}
$ kubectl logs ${POD_NAME}
```

Listing 5.78: Getting information about pods

Even the **pod status** may provide us with enough information, what steps should be taken next. If a pod is in Pending status, then it could mean that there are not enough resources in the cluster (e.g. the worker nodes use too small *EC2 instance types* or there are too few worker nodes). A pod may be also in status *CrashLoopBackOff* which indicates that a container is repeatedly crashing after restarting. There may a problem also that a Docker image for a container cannot be downloaded and a pod may be in state *ImagePullBackOff* or *ErrImagePull* [134].

It is also always a good idea to **read log messages** (so access to log messages should be always supplied). The following Kubernetes components log messages may be helpful [124]:

- logs from *kube-apiserver*, which is responsible for serving the API,

- logs from *kube-scheduler*, which is responsible for making scheduling decisions,
- logs from *kube-controller-manager*, which manages replication controllers,
- logs from *kubelet*, which is responsible for running containers on the node,
- logs from *kube-proxy*, which is responsible for service load balancing.

Besides all the troubleshooting ideas presented earlier, one can also visit solutions dedicated entirely to AWS EKS clusters [42] or to *eksctl* [185]. Still, some problems may be tough to handle. But even then, there are always measures that could be applied to move further into the correct direction. One may **ask a question** on Stack Overflow or talk to a Kubernetes team on Slack or write on a Kubernetes forum or file a bug on Kubernetes *Github.com* project website [125].

To summarize: there are various ways how to debug a Kubernetes cluster or an application deployed on top of it. Some steps may be applied to any Kubernetes cluster, other concern only the clusters deployed using a particular method. Nevertheless, if one is in need, they can ask for help or talk with people who are more experienced with Kubernetes.

5.7 Summary

This was an empirical chapter. Many Kubernetes clusters were deployed. The two selected deployment methods were utilized: *kops* and *eksctl*. Apart from ensuring that the production environment requirements are met, the chapter also described some encountered problems and provided solutions. The chapter ended with information how to troubleshoot a Kubernetes cluster.

6 Comparison of the used methods of Kubernetes cluster deployment

In this chapter, several comparison criteria are used in order to compare the two chosen methods of a Kubernetes cluster deployment. The chapter ends with a summary deciding which method was better in relation to which comparison criterion.

Even though the two methods: using *kops* and using *eksctl*, help to deploy a Kubernetes cluster, they have some differences. For example: *kops* works for many clouds (e.g. AWS, GCP), whereas *eksctl* supports only AWS. Another difference is the default AMI. *kops* chooses Debian Operating System, while *eksctl* uses Amazon Linux 2. These AMIs are, however, only defaults, one can always choose another AMI. The mentioned differences can be known by skimming through the official documentation. On the contrary, this chapter is based on the empirical work.

6.1 Production requirements which could not be satisfied

There were nine production environment requirements which were supposed to be met by both tested methods. **All of them were met.** This means that both of the methods can be applied in production use cases. The table 6.1 provides a brief explanation how each of the requirements was satisfied for each method.

Many requirements were handled in the same way for both methods. Subjectively, the hardest requirement to meet was: security, using *eksctl*. There was a problem with finding a working YAML configuration and also ensuring that cluster was healthy. ***Eksctl provides a more automated approach.*** For example the *AWS CloudWatch LogGroup*, needed for central logging, was automatically created when using *eksctl*, but not when using *kops*. Also, *eksctl* provides the cluster which is already *Highly Available*.

On the other hand, ***kops provides more flexibility.*** The master node can be accessed with *SSH* when using *kops* and not when using *eksctl*. When using *kops*, it is possible to see all the kube-system namespaced pods, which is not the case with *eksctl*. When using *eksctl* no parameters of the control plane components can be set (to the best of

the author's knowledge), with *kops* it is possible.

Table 6.1: A comparison of how each production requirement was satisfied using the two methods: *kops* and *eksctl*

Requirement	Using <i>kops</i> method	Using <i>eksctl</i> method
A healthy cluster	The same approach was used, <i>Bats-core</i> was chosen as a test framework	
Automated operations	The same approach was used, a <i>Bash</i> file <i>tasks</i> was used	
Central Monitoring	It was provided by AWS upfront, thanks to <i>AWS CloudWatch</i>	
Central Logging	Two <i>Helm</i> charts were deployed	It was easy to configure with <i>YAML</i>
Central Audit	It was provided by AWS upfront, thanks to <i>AWS CloudTrail</i>	
Backup	The same approach was chosen, <i>Velero</i> was used with an S3 bucket	
High Availability	It was easy to configure either with <i>YAML</i> or <i>CLI</i>	It was provided by <i>AWS EKS</i> upfront
Autoscaling	The same approach was chosen, <i>ClusterAutoscaler</i> was deployed	
Security	It was easy to set in <i>YAML</i>	Finding a working <i>YAML</i> configuration was more demanding

6.2 Time of chosen Kubernetes cluster operations

The next criterion used to compare the two deployment methods was time. **The time of several operations concerning Kubernetes cluster management was measured.** Each of the operation was performed several times and the mean value was used. Time was measured separately for **a minimal working cluster and for the full cluster.**

The minimal cluster here means such a cluster that is usable for end user and utilizes

the default configuration (central monitoring, central audit, healthy cluster, automated operations are met). The production-grade cluster includes also the following requirements: HA, central logging and security. The about 5 min difference between creating a minimal and full cluster using *eksctl* was caused by setting the security measures. The full cluster here does not include backup and autoscaling, because meeting these requirements is negligible in the terms of time and takes the same time for *kops* and *eksctl*.

Testing a cluster involves running the automated tests – verifying that the cluster is healthy. It does not involve testing each of the production deployment requirements. Both the minimal and full clusters use *t2.micro* for *kops* and *t2.small* for *eksctl* for worker nodes. The time observed for the cluster operations is given in table 6.2 and in figure 6.1.

Table 6.2: Time needed to run Kubernetes management operations using *kops* and *eksctl*

Operation	Using <i>kops</i> method	Using <i>eksctl</i> method
Create a minimal cluster	6 min 12 s	19 min 26 s
Create a production-grade cluster	6 min 35 s	25 min 40 s
Test a cluster	6 min 33 s	6 min 40 s
Delete a cluster	2 min 23 s	13 min 25 s

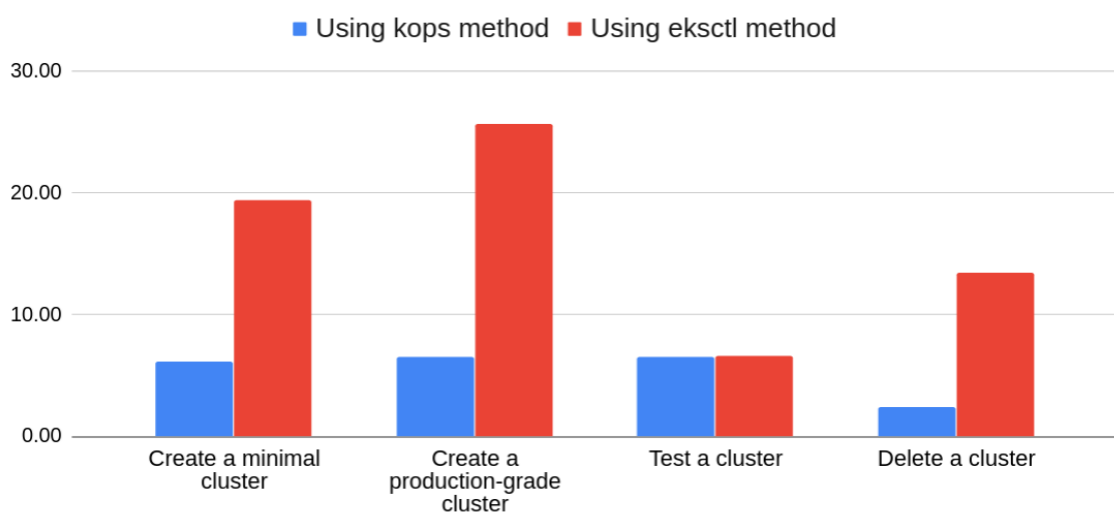


Fig. 6.1: Time needed to run Kubernetes management operations using *kops* and *eksctl*

The files' names used to perform each cluster operation are demonstrated in table

6.3. The files can be found in the public Git repository, presented in section 5.1.2.

Table 6.3: Files used to deploy minimal and full clusters using: *kops* and *eksctl*

Which cluster	Using <i>kops</i> method	Using <i>eksctl</i> method
A minimal cluster	<i>src/kops/cluster-minimal.yaml</i>	<i>src/eks/cluster-minimal.yaml</i>
A production cluster	<i>src/kops/cluster.yaml</i> and central logging deployed	<i>src/eks/cluster-minimal.yaml</i>

To summarize, it is clearly visible that operating a Kubernetes cluster is faster with *kops*. The time of operations is an important criterion, because firstly, the faster it is, the faster one can get the feedback and experiment with the cluster. And secondly, longer creation and deletion times mean longer time of keeping AWS resources created, which in turn means that they cost more.

6.3 Additional steps needed to create a Kubernetes cluster

In order to deploy the clusters, many tools were needed. However, these tools were packaged inside a Docker image. Therefore, the development environment, provided by the Docker image was reliable and easily reproducible. It could be destroyed and recreated any time. But, apart from the development tools, there were also some prerequisites needed to be deployed for the production-grade clusters.

When using *kops*, one has to use and create an S3 bucket. *Kops* needs this bucket to store cluster configuration. Contrary to that, AWS EKS does not need any additional AWS Resources created beforehand. However, this work attempted to deploy Kubernetes clusters in a production environment. In order to satisfy the backup requirement, *Velero* was used. *Velero* needs some location to store backups. In this work, a S3 bucket was used. Therefore **an S3 bucket was needed for both deployments.**

To summarize, both methods require the same steps to work in a production environment specified in this work, but the S3 bucket was used by choice and it could be replaced with some other storage solution.

6.4 Minimal EC2 instance type needed for a Kubernetes worker node

Various EC2 instances types cost various amount of money. Thus, it is preferable to use the cheapest possible instance type. In this subchapter the minimal EC2 instance type needed for a Kubernetes worker node, in a production ready cluster, will be indicated.

In order to meet production environment requirements, this number of pods is needed to be deployed on a worker node (additionally to the pods deployed by default by *eksctl* and *kops*):

- **three pods for a cluster created with *eksctl*** (one pod for testing, one pod for backup and one pod for autoscaler),
- **five pods for a cluster created with *kops*** (one pod for testing, one pod for backup, one pod for autoscaler, two pods for logging).

As far as *eksctl* method is concerned, there are by default four pods deployed on a worker node. This was shown in listing 5.72. Thus, **altogether, there have to be $4 + 3 = 7$ pods deployed on a worker node**. Now, the hard limits described in section 4.4.7 have to be considered. The EC2 instance types *t2.nano* and *t2.micro* allow for maximally 4 pods on one EC2 instance [59]. The next, bigger instance type is *t2.small* and it allows for 11 pods. Conclusion: when it comes to the number of allowed pods on a worker node, the minimal EC2 instance type is *t2.small*.

Another thing to consider is whether *t2.small* is enough when it comes to EC2 CPU and memory resources. This was checked empirically, by deploying a cluster with *eksctl* and then deploying the three pods needed to satisfy the production environment requirements (listing 6.1).

```
eks$ export K8S_EXP_ENVIRONMENT=testing
eks$ ./tasks _create
eks$ ./tasks _enable_velero
eks$ ./tasks _enable_as
eks$ ./tasks _test
```

Listing 6.1: Deploying a production ready *eksctl* cluster

All the seven pods were in status *Running* and the tests were successful. Thus, it may be concluded that when it comes to EC2 instance CPU and memory resources, the EC2 instance type *t2.small* is enough. This may be seen in listing 6.2.

```

eks$ ./tasks _test
+ cd ../tests
+ bats tests.bats
    kubernetes machines have correct version
    1 kubernetes worker nodes have status: Ready
    kubectcl cluster-info returns expected output
    a test application can be deployed on kubernetes
    a test application is available (ingress resource works)
    a test service (test website) can be deleted from kubernetes

6 tests , 0 failures

real    6m46.134s
user    0m13.979s
sys     0m4.544s
+ cd ../eks
eks$ kubectl get --all-namespaces pod

```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	aws-node-p4985	1/1	Running	0	11m
kube-system	cluster-autoscaler-7d7b5564df-2t6jj	1/1	Running	0	3m2s
kube-system	coredns-8568c98b77-hps7d	1/1	Running	0	27m
kube-system	coredns-8568c98b77-l4pw5	1/1	Running	0	27m
kube-system	kube-proxy-md2fd	1/1	Running	0	11m
testing	apache-testing-7448f49855-qdfzx	1/1	Running	0	111s
velero	velero-6fffc8dc85-tflxg	1/1	Running	0	3m54s

Listing 6.2: Verifying a production ready *eksctl* cluster

The third aspect is, that it is possible to deploy some pods on a master node or mark the master node as schedulable only by Kubernetes system components [108]. However, this is not the case for *AWS EKS*. In *AWS EKS*, the control plane is locked down and scheduling any workloads on the control plane nodes is forbidden [137].

Contrary to the *AWS EKS* cluster, there are no hardcoded limits for the number of pods when using *kops*. From the experiments run in the section 5.4.5, it was tested that the *EC2 instance type t2.micro* is too small. The experiment was repeated with the *EC2 instance type t2.small* for a worker node and with deploying all the services needed for a production environment (listing 6.3).

```

kops$ export K8S_EXP_ENVIRONMENT=testing
kops$ ./tasks _create
kops$ ./tasks _enable_velero
kops$ ./tasks _enable_as

```

```
kops$ ./tasks _enable_logging
kops$ ./tasks _test
```

Listing 6.3: Deploying a production ready *kops* cluster

The tests were successful and all the pods were in status *Running*. Thus, it may be concluded that the minimal *EC2 instance type* is *t2.small*.

To summarize, **both methods of deploying Kubernetes clusters *kops* and *eksctl* allow worker nodes minimally of *EC2 instance type: t2.small***, in order to satisfy the production deployment requirements. However, for experimental environments, *kops* allows for smaller types (*t2.micro* can be used).

6.5 Easiness of configuration

With *kops*, it was harder to generate the config file. *Kops*' documentation recommends that one should use *kops create cluster* command instead of trying to generate it by hand. Using this command, one needs to create an S3 bucket first, the configuration file is put there and then, the object hosted on an S3 bucket can be exported to a local file. On the contrary, **with *eksctl*, it was easier to generate the YAML file and it was done by hand.**

On the other hand, ***kops* allows for configuring the parameters of the control plane components, while *eksctl* does not.** Another thing is, that ***kops* allows using *Golang templates*** [83]. Thus, it is possible to use one template file and use it, for example, to deploy into two different environments: testing and production. However, since such templating solution was not possible with *eksctl* (to the best knowledge of this work's author), in this work, template files with *Bash* variables were used. It was easier to manage the template files for *kops* and for *eksctl* in the same way. This solution worked well and accomplished its purpose.

Considering the encountered problems, **there was an unexpected error, when creating the cluster with *eksctl* and applying the security measure** to restrict the IP addresses which can access the API server. It was, subjectively, quite hard to find the working configuration and also it took long, because creating the not working cluster then, resulted in timeout after about 43 minutes. However, this problem was solved.

6.6 Meeting the automation requirement

It is worth to summarize how it went to automate the cluster operations. Firstly, both ***kops* and *eksctl* automatically edited the kubeconfig file**. This was tremendously helpful, because immediately after cluster creation, one can access a cluster using *kubectl* command.

Secondly, there was a difference concerning the moment in which the cluster creation commands return. **The *kops* creation command returned immediately**. This means, that in order to automate the cluster creation to the extent that it can be run on a CI server, **some waiting mechanism was needed**. However, the waiting mechanism was easy to implement, because *kops* provides the *kops validate cluster* command. The command was run in a loop (with sleep in between each trial) and after it succeeded, the cluster was deemed created. On the other hand, **creating the cluster with *eksctl* returned after all the AWS resources were created**.

Besides the aforementioned differences, **automating the operations was done in a very similar manner for both deployment methods**. Creating the configuration was harder with *kops*, but this was a manual step and it does not need to be run on a CI server.

6.7 Cost of chosen Kubernetes cluster operations

In order to be aware of the cost generated by AWS resources needed to run the clusters, **an AWS tag was applied to all the AWS resources**. Thanks to such a solution, it was possible to use the **AWS Cost Explorer** to verify AWS resources cost. The particular tag applied was either *deployment: eks-testing* or *deployment: kops-testing*, depending on the deployment method used. In practice, no cluster was deployed with the label *deployment: kops-production* or *deployment: eks-production*, because in this work there was no difference between the testing and production environment. The intention of having 2 environments was to prove that all the configuration and operations can be automated for multiple environments and, so that in the future, it is possible to choose the environment. Chart taken from the AWS Cost Explorer website is presented in figure 6.2.

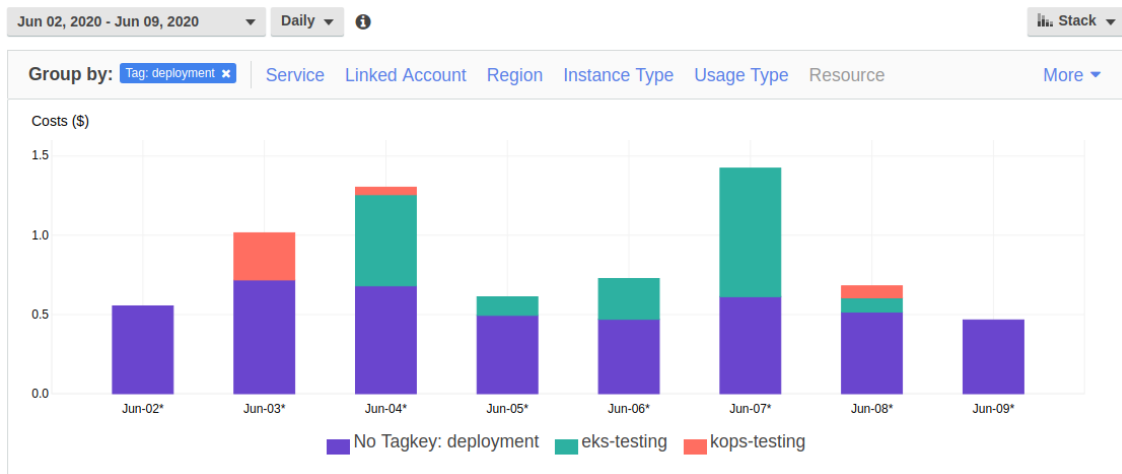


Fig. 6.2: Cost report available on AWS Cost Explorer, grouped by the AWS tag: *deployment*

This chart should not be treated as a single source of truth, because firstly, the AWS tags were enabled later in the empirical work phase and secondly, experimenting with the clusters created with *kops* took different amount of time and was performed different amount of times than experimenting with the clusters created with *eksctl*. **The chart is just a way of showing an example how it is possible to monitor Kubernetes clusters cost.**

In order to be able to compare the cost of a cluster created with *kops* and a cluster created with *eksctl*, the following steps were taken.

1. A cluster was created with *kops*, using the *src/kops/cluster.yaml* file and central logging was deployed.
2. All the AWS resources decorated with the tag: *deployment kops-testing* were listed with the following command: `aws resourcegroupstaggingapi get-resources --tag-filters Key=deployment,Values=kops-testing`.
3. The *kops* cluster was deleted.
4. A cluster was created with *eksctl*, using the *src/eks/cluster.yaml* file.
5. All the AWS resources decorated with the tag: *deployment eks-testing* were listed with the following command: `aws resourcegroupstaggingapi get-resources --tag-filters Key=deployment,Values=eks-testing`.
6. The AWS EKS cluster was deleted.

This procedure resulted in two lists of all AWS resources, used by each Kubernetes deployment method. Next, all the free AWS resources (*VPC Subnets, Security Groups, Internet Gateway, Route Table*) were omitted. Overall, using *eksctl*, the following AWS resources were used: 1 *Nat Gateway*, 2 *CloudFormation stacks*, *AWS EKS cluster*, 2 *EC2 instances* of type: *t2.small* (used as worker nodes), an *S3 bucket*, 1 *Load Balancer (ALB)* (to expose API server). For *kops*, there was a bug and not all the AWS resources were tagged [171]. That bug should be fixed in the future releases of *kops*, because there is already a PR merged on *kops Github.com* website [169]. Despite this, it was possible to itemize the not free AWS resources used by *kops*: 3 *EC2 instances* of type: *t2.micro* (used as master nodes), 2 *EC2 instances* of type: *t2.small* (used as worker nodes), 1 *Load Balancer (ALB)* (to expose API server), 6 *EBS volumes* (3 for *etcd-main* and 3 for *etcd-events*) of type *gp2* and 20GB each, an *S3 bucket*. Whenever *Elastic IPs* were used, they were omitted, because an attached *Elastic IP* does not cost anything.

Then, the cost for running a cluster for one month was computed. It is presented in tables 6.4 and 6.5. Each of them lists the AWS resources used by one deployment method. **The last row of each table contains the total amount, in USD, of the monthly cost of the AWS resources.** The calculated cost does not include everything (e.g. transfer costs), but it should be enough to compare *kops* to *eksctl*. Many pricing information is given by Amazon per hour. 720 hours in a month was assumed. When it comes to *CloudFormation* resources, there were 2 stacks used. The resources managed by the *CloudFormation* stacks were received with the commands presented in listing 6.4.

```
$ aws cloudformation list-stack-resources --stack-name eksctl-eks-testing-nodegroup-ng-1
$ aws cloudformation list-stack-resources --stack-name eksctl-eks-testing-cluster
```

Listing 6.4: AWS CLI commands used to list *CloudFormation* resources used by *eksctl*

The first stack managed 10 resources and the second stack - 34. The *CloudFormation* pricing [51] depends on operations count. $10+34=44$ creation operations and $10+34=44$ deletion operations were assumed based on the information given. 88 operations total. The pricing website also informs that for operation durations above 30 seconds per operation, one will be charged \$0.00008 per second above the threshold. It cannot be assumed that all the resources took the same time to create, this was not measured. Thus, this cost is not included in the below calculations.

When it comes to the S3 cost, the storage cost is very cheap - \$0.023 for 1 GB [48]. The storage used by either *kops* or *eksctl* was for sure less than 1 GB, thus \$0.023 was assumed as monthly usage cost. Transfer costs were omitted.

Table 6.4: Expected cost of running a Kubernetes cluster created with *eksctl* for one month

AWS Resource	Cost using <i>eksctl</i> method
1 Nat Gateway [49]	$720 * \$0.048 = \34.56
Classic Load Balancer [58]	$720 * \$0.028 = \20.16
2 CloudFormation stacks [51]	$88 * \$0.0009 = \$ 0.0792$
AWS EKS cluster [38]	$720 * \$0.10 = \$ 72$
2 EC2 instances of type: <i>t2.small</i> [33]	$2 * 720 * \$0.025 = \36
keeping files on S3 bucket [48]	\$0.023
Total	\$ 162.8222

Table 6.5: Expected cost of running a Kubernetes cluster created with *kops* for one month

AWS Resource	Cost using <i>kops</i> method
3 EC2 instances of type: <i>t2.micro</i> [33]	$3 * 720 * \$0.0126 = \$ 27.216$
2 EC2 instances of type: <i>t2.small</i> [33]	$2 * 720 * \$0.025 = \36
Classic Load Balancer [58]	$720 * \$0.028 = \20.16
6 EBS volumes of type gp2 and 20GB each [32]	$6 * 20 * \$0.11 = \13.2
keeping files on S3 bucket [48]	\$0.023
Total	\$96.599

Based on the calculations provided above it costs \$ 162.83 to run a Kubernetes cluster created by *eksctl* and \$ 96.60 to run a Kubernetes cluster created by *kops*, on AWS, for a 30-day month. Some costs, e.g. data transfer, were not included.

6.8 Summary

This was the main chapter of this work. It utilized the knowledge and experience gained throughout the previous, empirical chapters. The goal of this master's thesis was to com-

pare two methods of a Kubernetes cluster deployment: using *kops* and using *eksctl*. In this chapter, seven comparison criteria were applied. The criteria entailed: whether all the production environment requirements were met, time needed to operate a cluster, number of additional prerequisites, minimal *EC2 instance types* for worker nodes, easiness of configuration, meeting the automation requirement and cost of chosen Kubernetes cluster operations.

Some criteria are subjective, because one may enjoy having a wide choice of possible configuration to set, whereas another (or other use case) would prefer to create a cluster without a redundant hassle. Unquestionably: time and cost are the criteria helping us to decide objectively which method is better. Considering the two criteria, the winner is *kops*. *Kops* also provides more configuration options and can be applied on multiple clouds (not only AWS).

On the contrary, *eksctl* provides an easier way to configure the cluster, because it is convenient to create a *YAML* configuration file by hand. There are also no prerequisites needed before performing the deployment with *eksctl*, unless one needs to satisfy the backup requirement. Then, some storage location (such as an *S3* bucket) is needed. *Kops* needs an *S3* bucket to keep the cluster configuration. Besides, *eksctl* provides a managed service (*AWS EKS*), thus the Kubernetes cluster administrator does not have to worry about managing the control plane components.

The last things that should be noted is that *eksctl* is not the same as *AWS EKS*, even though *eksctl* utilizes *AWS EKS* resource. Furthermore, all the results collected in this study may depend on the versions of *kops*, *eksctl* and maybe Kubernetes itself. This thesis used the particular versions of each tool and is based on the author's own experiments.

7 Conclusions

A summary of the whole study is demonstrated in this chapter.

The aim of this study was to deploy a Kubernetes cluster in a production environment, on AWS cloud, to use multiple deployment methods, and to compare them. Two deployment methods were chosen: using *kops* and using *eksctl*. They were described in the chapter: 3. Both of them facilitate the deployment tasks to a tremendous extent.

Production environment was defined in the 2.6 chapter and it was stated, that several requirements must be met in order to be able to name a deployment production-ready. Some ideas of how to satisfy the requirements were described in the 4 chapter. Obviously, there are more such requirements and this study does not exhaust this topic. Then, a decision was made on which production environment requirements should be satisfied in this study. 9 production environment requirements were selected.

Next, the development environment and essential tools were chosen. The following 5 chapter described in detail how the Kubernetes clusters were deployed, using the two methods. The cluster operations were: create, test and delete. They were automated to the point, so that they can be run using a single *Bash* command. All the commands needed to configure the cluster and to satisfy all the production requirements were presented. Additionally, the study was supposed to have a practical approach, thus several encountered problems and solutions to them were described. This chapter ended with general guidelines on Kubernetes clusters troubleshooting.

Then, the two methods were compared using several comparison criteria. The criteria contained, among others: time, cost and the verdict whether a deployment method can satisfy all the production environment requirements. This comparison chapter, 6, was based on the whole empirical work and experiments, described earlier. The conclusion was that each method can be used for different use cases. For example: *kops* allows more configuration, whereas *eksctl* is easier to configure. The method which was deemed faster (in relation to cluster operations) and cheaper was *kops*.

7.1 Lessons learned

Writing this work allowed to **gather some knowledge about Kubernetes, its components and architecture**. The best documentation sources happen to be the official Kubernetes, *AWS EKS*, *kops* and *eksctl* websites. But, there are also many blog posts available providing tutorials on how to use *kops* or *eksctl*. However, not all the expertise can be achieved just by reading. The many cluster deployments were very helpful to build an intuition of how a Kubernetes cluster works, what are the potential problems and how to deal with them. For instance, there was an unexpected problem, when deploying with *eksctl*. This problem was caused by *AWS EKS* hard limits of number of allowed pods on a particular *EC2 instance type*. Many problems were described in this work and solutions for them were provided too.

Furthermore, it was experienced, that **one should aspire to satisfy the requirements one by one**. This helps to isolate the failure - meaning: when there is some issue, the scope of the issue is smaller. Secondly, working on one requirement at a time, makes it easier to later describe how the requirement was fulfilled.

Next, **the automation requirement turned out to be extremely vital**. Thanks to the cluster operations being automated, one could recreate a cluster any number of times and, each time, expect the same results. One could then rely on the work already performed and build upon its results. Automation enables us to have easily reproducible environments and facilitates the experiments repetition.

Overall, this work was an opportunity to immerse into the Kubernetes world and to get to know its possibilities and problems. The nice aspect of such work was that this kind of technology is still fresh and that the history of the programs described in this work happens now (e.g. through the pull requests on *Github.com*). This also has its downsides, for example a bug may just have been discovered and a solution has not yet been released.

7.2 Future work potential

There is a great deal of improvements which can be applied while deploying a Kubernetes cluster using various methods and trying to compare them. Probably the most obvious way to make such a comparison better, is to **add more deployment methods**. Kubernetes

can be deployed in the cloud, thus other managed services like: Azure Kubernetes Service (AKS) or Google Kubernetes Engine (GKE) could be tried. But there are also many custom solutions available, e.g. using Terraform, described in the chapter: 3.

Secondly, in this work 9 production environment requirements were considered. Yet, many more could be. Ideas for **additional requirements are**: cluster live upgrades, cluster live reconfiguration, dealing with persistent data, obeying some country laws (concerning for example how long some data should be kept) and more.

The third aspect is, that **every administrator or every company has their own views on how to satisfy each of the requirements**. For some people whitelisting one IP address to have access to a Kubernetes API server is enough, while the other could demand to keep all master and worker nodes in a private network and use a bastion host. The same applies to other requirements. Potentially, the automation requirement may have been met by deploying some CI server, creating a pipeline which would create, test and then delete a testing cluster and only then, another pipeline could have been run to deploy to a production environment. Besides, there are many tools listed in the 2.6 chapter, like Grafana, Graylog, Logstash, which could have been applied.

The next thing is that, in order to decide which method is the best for a production deployment, **a production load could be simulated**. This could be accomplished either by running experiments on e.g. 100 EC2 worker nodes or conducting a survey among Kubernetes administrators (or companies using Kubernetes) and asking questions about the long running clusters. Such experiments could lead to different problems, than ones described in this study. There could be some AWS hard limits reached on the allowed number of *EC2 instances* or perhaps there could be a problem with not fast enough data transfer.

This leads to the fifth idea, namely, the **comparison criteria could be performance oriented**. In such a case, a chosen test application could have had needs for special resources, maybe running some Machine Learning tasks, demanding access to GPU, lightning-fast data transfer and tiny latencies. Then, the cluster providing the most efficient resources would have been a winner. Choosing this approach, one could also compare the results with the existing papers like the one paper which evaluates a performance of containers running on Managed Kubernetes Services [11].

To sum up, there is a wide variety of the potential future work. Kubernetes is a very popular platform, thus it can be hoped that more academic papers, regarding this topic, will appear. For now, Kubernetes is still evolving and many helpful tools (like *kops* and *eksctl*) are being worked on.

List of Figures

2.1	The difference between two software architecture styles: monolith and microservices [154]	11
2.2	A simple example CI pipeline, consisting of a few stages, starting with a developer uploading a change of code	13
2.3	Hosted serverless platforms preferred by CNCF community during September and October 2019 [104]	15
2.4	Cloud native storage preferred by CNCF community during September and October 2019 [104]	15
2.5	Results of CNCF Survey from year 2017 showing Kubernetes as number one cloud management platform [103]	20
2.6	Kubernetes cluster diagram depicting a master-slave architecture	21
2.7	Kubernetes components including master and node components [119] . .	22
2.8	Example Grafana dashboard for a Kubernetes cluster, showing among others: Memory, CPU and File system usage [159]	27
2.9	Kubernetes dashboard depicting CPU and Memory usage by Kubernetes pods[127]	28
2.10	Security features which a request sent to API Server goes through [5] . .	33
2.11	RPO and RTO illustrated in relation to time	34
3.1	Top 10 tools used by organizations and companies to manage containers [104]	37
3.2	The logos of three Managed Kubernetes Services, offered by the major cloud providers	38
3.3	Selected opensource tools which deploy a Kubernetes cluster, found on <i>Github.com</i> , state for date: 20.04.2020	39
3.4	A schema presenting the stages of working with AWS EKS	44
3.5	Stages of working with a Kubernetes cluster deployed on AWS with <i>kops</i> .	46
5.1	AWS <i>CloudWatch</i> with logs from a Kubernetes cluster	77
5.2	AWS <i>CloudWatch</i> with logs from a Kubernetes cluster	78

5.3	AWS Management Console showing tags applied on <i>EC2 instance</i> which is a master node	79
5.4	AWS <i>CloudWatch</i> metrics of a worker node, CPU utilization	79
5.5	AWS <i>CloudWatch</i> metrics of a worker node, Network input	80
5.6	AWS <i>CloudTrail</i> events of a <i>kops</i> cluster	80
5.7	Backup of a Kubernetes namespace created by Velero, kept in an S3 bucket	83
5.8	<i>EC2 instance</i> in HA mode for <i>kops</i> cluster	87
5.9	AWS EKS resource with tags set, viewed on AWS Management Console .	98
5.10	AWS CloudFormation resource with tags set, viewed on AWS Management Console	98
5.11	CloudFormation stacks needed by <i>eksctl</i>	99
5.12	CloudFormation resources needed by <i>eksctl</i>	99
5.13	AWS EKS instances, no master nodes	100
5.14	AWS <i>CloudWatch</i> logs for <i>eksctl</i> cluster	101
5.15	AWS <i>CloudWatch</i> logs for <i>eksctl</i> cluster – concerning API server	101
5.16	AWS <i>CloudWatch</i> monitoring for a worker node	103
5.17	AWS <i>CloudTrail</i> events concerning <i>eksctl</i> cluster	104
5.18	Backup of <i>eksctl</i> cluster created by Velero	106
6.1	Time needed to run Kubernetes management operations using <i>kops</i> and <i>eksctl</i>	116
6.2	Cost report available on AWS Cost Explorer, grouped by the AWS tag: <i>deployment</i>	122

List of Tables

4.1	Comparison of the price, noted in USD, of 1 hour running of an <i>EC2 instance</i> on AWS, considering a few selected AWS Regions. Based on the official AWS EC2 pricing website [33]	51
4.2	Comparison advantages and disadvantages of having a few bigger nodes in a Kubernetes cluster instead of having many smaller nodes [188]	60
5.1	Options set to <i>kops</i> when creating a Kubernetes cluster	73
6.1	A comparison of how each production requirement was satisfied using the two methods: <i>kops</i> and <i>eksctl</i>	115
6.2	Time needed to run Kubernetes management operations using <i>kops</i> and <i>eksctl</i>	116
6.3	Files used to deploy minimal and full clusters using: <i>kops</i> and <i>eksctl</i> . . .	117
6.4	Expected cost of running a Kubernetes cluster created with <i>eksctl</i> for one month	124
6.5	Expected cost of running a Kubernetes cluster created with <i>kops</i> for one month	124

List of Listings

2.1	A Dockerfile to build a Docker image with SSH server installed. Based on Docker official documentation [150]	17
3.1	A command of <i>eksctl</i> CLI tool used to create a Kubernetes cluster	43
3.2	An example <i>YAML</i> file used to customize a Kubernetes cluster created with <i>eksctl</i> CLI tool [183]	44
3.3	The commands of <i>kops</i> CLI tool used to create a Kubernetes cluster configuration and then to deploy the cluster	45
3.4	A command of <i>kops</i> CLI tool used to edit a Kubernetes cluster configuration	45
3.5	A command of <i>kops</i> CLI tool used to create a Kubernetes cluster using a <i>YAML</i> configuration file	46
4.1	A command of <i>AWS CLI</i> tool used to list all the available regions (for an <i>AWS</i> account)	50
4.2	A command of <i>AWS CLI</i> tool used to list all the available <i>AZs</i> in the chosen <i>AWS</i> Region)	51
4.3	Commands provided by tasks file – a <i>Bash</i> script which automates a Kubernetes cluster operations	56
5.1	An example code listing explaining the format	65
5.2	Command used to create a cluster with <i>kops</i> , without prerequisite steps performed	66
5.3	Output of the commands used to create a cluster with <i>kops</i> , without prerequisite steps performed	66
5.4	Commands used to set an <i>AWS</i> <i>S3</i> bucket for <i>kops</i>	67
5.5	Commands used to generate <i>kops</i> configuration	67
5.6	Command used to edit a Kubernetes cluster managed by <i>kops</i>	68
5.7	Command used to deploy a Kubernetes cluster with <i>kops</i>	69
5.8	Commands run to connect with a cluster created by <i>kops</i> together with output	69
5.9	Command used to request information about a running Kubernetes cluster	70
5.10	Command used to delete a Kubernetes cluster created with <i>kops</i>	70

5.11	Commands used to create a cluster with <i>eksctl</i> , without prerequisite steps performed	71
5.12	Command used to list Kubernetes worker nodes to verify that one such node was running	71
5.13	Command used to delete Kubernetes cluster with <i>eksctl</i>	72
5.14	Commands used to generate a cluster configuration with <i>kops</i>	73
5.15	Command used export <i>kops</i> configuration from S3 to a local file	74
5.16	AWS IAM permissions added to <i>kops</i> cluster template needed for logging	74
5.17	<i>Bash</i> commands automating cluster configuration generation	75
5.18	<i>Bash</i> command automating cluster creation	75
5.19	A waiting mechanism that waits until a <i>kops</i> cluster is ready	75
5.20	Listing all AWS <i>EC2 instances</i> by AWS tag	76
5.21	Kubernetes components in a <i>kops</i> cluster	76
5.22	Commands providing the logging solution	77
5.23	An example Kubernetes log message, presented on AWS <i>CloudWatch</i> . .	78
5.24	Automated command to install <i>Velero</i> CLI	81
5.25	Contents of AWS credentials file	81
5.26	Deploying a vs <i>Helm</i> chart	81
5.27	Choosing a backup storage location for <i>Velero</i>	81
5.28	Testing backup operation	82
5.29	Simulating a disaster to test backup	83
5.30	Restoring the backup	83
5.31	<i>Bash</i> command to deploy autoscaler	84
5.32	IAM Policies needed by autoscaler	84
5.33	<i>Kops</i> cluster <i>YAML</i> configuration needed for autoscaler	84
5.34	Deploying a test application	85
5.35	Scaling the test application pod replicas	85
5.36	Listing the <i>EC2 instances</i> , 1 of them was created by autoscaler	85
5.37	Getting information about the cluster	86
5.38	Kubernetes control plane components, <i>High Availability</i>	87
5.39	Testing a <i>kops</i> cluster	88

5.40	Testing a <i>kops</i> cluster - deeper dive	88
5.41	Contents of a test application - Apache web server	89
5.42	<i>Bats-core</i> tests	89
5.43	Testing a <i>kops</i> cluster	91
5.44	Debugging <i>kube2iam</i> pod	92
5.45	Debugging the <i>fluent-cloudwatch</i> pod	92
5.46	Installing <i>Velero</i> server	93
5.47	<i>Velero</i> server installation error	93
5.48	Another attempt to deploy <i>Velero</i> server	93
5.49	Debugging <i>Velero</i>	94
5.50	Listing AWS IAM keys	95
5.51	Debugging autoscaler	95
5.52	Error output of <i>kops create cluster</i> command	96
5.53	Creating <i>eksctl</i> configuration	96
5.54	Creating a Kubernetes cluster with <i>eksctl</i>	97
5.55	<i>Eksctl</i> configuration file	97
5.56	Updating <i>eksctl</i> cluster	100
5.57	Updating <i>eksctl</i> cluster configuration	100
5.58	Enabling logging in <i>eksctl</i> cluster	100
5.59	Example log message	102
5.60	Output from creating a <i>eksctl</i> cluster	103
5.61	<i>Eksctl</i> configuration applying security measures	104
5.62	<i>Eksctl</i> configuration applying security measures	105
5.63	Verifying connection to API server	105
5.64	Creating an S3 bucket for <i>Velero</i>	106
5.65	<i>Eksctl</i> configuration needed for autoscaler	106
5.66	Listing <i>EC2 instances</i> , one was created by autoscaler	107
5.67	Testing an <i>eksctl</i> cluster	108
5.68	Deleting an <i>eksctl</i> cluster	108
5.69	Attempt to setup security	108
5.70	Error output from creating <i>eksctl</i> cluster	109

5.71	AWS IAM policies recommended for <i>eksctl</i> by a blog post	109
5.72	Debugging cluster autoscaler	109
5.73	Debugging a test application	110
5.74	Verifying that all the pods are running	110
5.75	Deleting <i>eksctl</i> cluster	110
5.76	<i>Eksctl</i> special operations	111
5.77	Getting information about a cluster	112
5.78	Getting information about pods	112
6.1	Deploying a production ready <i>eksctl</i> cluster	118
6.2	Verifying a production ready <i>eksctl</i> cluster	119
6.3	Deploying a production ready <i>kops</i> cluster	119
6.4	AWS CLI commands used to list CloudFormation resources used by <i>eksctl</i>	123

References

- [1] The Kubernetes Authors. *CASE STUDY: Booking.com*. 2020. URL: <https://kubernetes.io/case-studies/booking-com/>. (accessed: 16.05.2020).
- [2] The Kubernetes Authors. *CASE STUDY: ING*. 2020. URL: <https://kubernetes.io/case-studies/ing/>. (accessed: 16.05.2020).
- [3] The Kubernetes Authors. *CASE STUDY: Zalando*. 2020. URL: <https://kubernetes.io/case-studies/zalando/>. (accessed: 16.05.2020).
- [4] The Kubernetes Authors. *Declarative Management of Kubernetes Objects Using Configuration Files*. 2020. URL: <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config/>. (accessed: 16.05.2020).
- [5] The Kubernetes Authors. *Kubernetes API Server- security*. 2020. URL: <https://kubernetes.io/docs/reference/access-authn-authz/>. (accessed: 16.05.2020).
- [6] The Kubernetes Authors. *Kubernetes official website*. 2020. URL: <https://kubernetes.io/>. (accessed: 16.05.2020).
- [7] Rekha Pitchumani and Yang-Suk Kee. "Hybrid Data Reliability for Emerging Key-Value Storage Devices". In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 309–322. ISBN: 978-1-939133-12-0. URL: <https://www.usenix.org/conference/fast20/presentation/pitchumani>.
- [8] John Arundel and Justin Domingus. *Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud*. 1st ed. O'Reilly Media, 2019. ISBN: 978-1492040767.
- [9] Sergi Blanco-Cuaresma et al. *Fundamentals of effective cloud management for the new NASA Astrophysics Data System*. 2019. arXiv: 1901.05463 [astro-ph. IM].
- [10] Gor Mack Diouf, Halima Elbiaze, and Wael Jaafar. "On Byzantine Fault Tolerance in Multi-Master Kuberntes Clusters". In: *ArXiv abs/1904.06206* (2019).
- [11] A. Pereira Ferreira and R. Sinnott. "A Performance Evaluation of Containers Running on Managed Kubernetes Services". In: *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2019, pp. 199–208.

- [12] Aneta Poniszewska-Marańda et al. "A real-time service system in the cloud". In: *J Ambient Intell Human Comput* 11, 961–977 (2020) (2019).
- [13] Leila Abdollahi Vayghan et al. "Kubernetes as an Availability Manager for Microservice Applications". In: (Jan. 2019).
- [14] Gigi Sayfan. *Mastering Kubernetes*. 2nd ed. Packt Publishing, 2018. ISBN: 978-1788999786.
- [15] Thalita Vergilio and Muthu Ramachandran. "Non-Functional Requirements for Real World Big Data Systems - An Investigation of Big Data Architectures at Facebook, Twitter and Netflix". In: May 2018. DOI: 10.5220/0006825408330840.
- [16] Mohammad M. Alshammari et al. "Disaster Recovery in Single-Cloud and Multi-Cloud Environments: Issues and Challenges". In: *The 4th IEEE International Conference on Engineering Technologies and Applied Sciences (ICETAS 2017)* (2017).
- [17] Khoa Mai. "Bachelor's thesis: Building High Availability Infrastructure in Cloud". Bachelor's Thesis. 2017.
- [18] Hylson V. Netto et al. "State machine replication in containers managed by Kubernetes". In: *Journal of Systems Architecture* Volume 73, February 2017, Pages 53-59 (2017).
- [19] Hideto Saito, Hui-Chuan Chloe Lee, and Cheng-Yang Wu. *DevOps with Kubernetes*. Packt Publishing, 2017. ISBN: 978-1-78839-664-6.
- [20] Thomas Uphill. *DevOps: Puppet, Docker and Kubernetes*. Packt Publishing, 2017. ISBN: 978-1788297615.
- [21] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jashidi. "Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture". In: *IEEE Software* 33(3):1-1 (2016).
- [22] Víctor Medel et al. "Modelling performance & resource management in kubernetes". In: Dec. 2016, pp. 257–262. DOI: 10.1145/2996890.3007869.
- [23] Kief Morris. *Infrastructure as Code*. 1st ed. O'Reilly Media, 2016. ISBN: 978-1491924358.
- [24] Joakim Verona. *Practical DevOps*. Packt Publishing, 2016. ISBN: 978-1-78588-287-6.
- [25] Omar H. Alhazmi and Yashwant K. Malaiya. "Evaluating Disaster Recovery Plans Using the Cloud". In: *Taibah University* (2013).

- [26] Ali Kanso, Maria Toeroe, and Ferhat Khendek. "Comparing redundancy models for high availability middleware". In: *Springer-Verlag* (2013).
- [27] Jinesh Varia. "Architecting for the Cloud: Best Practices". In: *AWS Whitepapers* (2011).
- [28] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st ed. Addison-Wesley Professional, 2010. ISBN: 978-0321601919.
- [29] Flaviu Cristian. "Understanding Fault-Tolerant Distributed Systems". In: *Communications of the ACM*, Vol. 34, Issue 2 (1993).
- [30] Amazon. *Amazon CloudWatch*. URL: <https://aws.amazon.com/cloudwatch/>. (accessed: 15.04.2020).
- [31] Amazon. *Amazon Compute Service Level Agreement*. URL: <https://aws.amazon.com/compute/sla/>. (accessed: 15.05.2020).
- [32] Amazon. *Amazon EBS Pricing*. URL: <https://aws.amazon.com/ebs/pricing/>. (accessed: 8.06.2020).
- [33] Amazon. *Amazon EC2 Pricing*. URL: <https://aws.amazon.com/ec2/pricing/on-demand/>. (accessed: 21.05.2020).
- [34] Amazon. *Amazon EKS*. URL: <https://aws.amazon.com/eks/>. (accessed: 16.05.2020).
- [35] Amazon. *Amazon EKS cluster endpoint access control*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/cluster-endpoint.html>. (accessed: 25.05.2020).
- [36] Amazon. *Amazon EKS clusters*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/clusters.html>. (accessed: 19.05.2020).
- [37] Amazon. *Amazon EKS FAQs*. URL: <https://aws.amazon.com/eks/faqs/>. (accessed: 4.06.2020).
- [38] Amazon. *Amazon EKS Kubernetes Pricing*. URL: <https://aws.amazon.com/eks/pricing/>. (accessed: 18.05.2020).
- [39] Amazon. *Amazon EKS Kubernetes versions*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/kubernetes-versions.html>. (accessed: 01.05.2020).

- [40] Amazon. *Amazon EKS-optimized Linux AMI*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-ami.html>. (accessed: 20.05.2020).
- [41] Amazon. *Amazon EKS Service Level Agreement*. URL: <https://aws.amazon.com/eks/sla/>. (accessed: 18.05.2020).
- [42] Amazon. *Amazon EKS troubleshooting*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/troubleshooting.html>. (accessed: 6.06.2020).
- [43] Amazon. *Amazon Elastic Container Registry*. URL: <https://aws.amazon.com/ecr/>. (accessed: 20.05.2020).
- [44] Amazon. *Amazon Elastic Container Service*. URL: <https://aws.amazon.com/ecs/>. (accessed: 20.05.2020).
- [45] Amazon. *Amazon Linux*. URL: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/amazon-linux-ami-basics.html>. (accessed: 22.05.2020).
- [46] Amazon. *Amazon Linux 2*. URL: <https://aws.amazon.com/amazon-linux-2/>. (accessed: 22.05.2020).
- [47] Amazon. *Amazon Machine Images (AMI)*. URL: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html>. (accessed: 20.05.2020).
- [48] Amazon. *Amazon S3 Pricing*. URL: <https://aws.amazon.com/s3/pricing/>. (accessed: 8.06.2020).
- [49] Amazon. *Amazon VPC pricing*. URL: <https://aws.amazon.com/vpc/pricing/>. (accessed: 8.06.2020).
- [50] Amazon. *AWS adds the ability for customers to enable AWS Local Zones themselves*. URL: <https://aws.amazon.com/about-aws/whats-new/2020/03/aws-customers-enable-aws-local-zones-themselves/>. (accessed: 21.05.2020).
- [51] Amazon. *AWS CloudFormation Pricing*. URL: <https://aws.amazon.com/cloudformation/pricing/>. (accessed: 8.06.2020).
- [52] Amazon. *AWS CloudTrail*. URL: <https://aws.amazon.com/cloudtrail/>. (accessed: 15.04.2020).
- [53] Amazon. *AWS Fargate*. URL: <https://aws.amazon.com/fargate/>. (accessed: 20.05.2020).

- [54] Amazon. *AWS Local Zones*. URL: <https://aws.amazon.com/about-aws/global-infrastructure/localzones/>. (accessed: 21.05.2020).
- [55] Amazon. *BACKUP AND RESTORE*. URL: https://eksworkshop.com/intermediate/280_backup-and-restore/backup-restore/. (accessed: 30.05.2020).
- [56] Amazon. *CI/CD WITH CODEPIPELINE*. URL: https://eksworkshop.com/intermediate/220_codepipeline/. (accessed: 20.05.2020).
- [57] Amazon. *Document history for Amazon EKS*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/doc-history.html>. (accessed: 16.05.2020).
- [58] Amazon. *Elastic Load Balancing pricing*. URL: <https://aws.amazon.com/elasticloadbalancing/pricing/>. (accessed: 8.06.2020).
- [59] Amazon. *File containing hard limits set on EKS, limiting the number of pods allowed for an EC2 instance type*. URL: <https://github.com/aws-labs/amazon-eks-ami/blob/master/files/eni-max-pods.txt>. (accessed: 22.05.2020).
- [60] Amazon. *Launching Amazon EKS Windows worker nodes*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/launch-windows-workers.html>. (accessed: 20.05.2020).
- [61] Amazon. *Linux Bastion Hosts on the AWS Cloud*. URL: <https://docs.aws.amazon.com/quickstart/latest/linux-bastion/overview.html>. (accessed: 25.05.2020).
- [62] Amazon. *Platform versions*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/platform-versions.html>. (accessed: 27.05.2020).
- [63] Amazon. *Regions, Availability Zones, and Local Zones*. URL: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html>. (accessed: 18.05.2020).
- [64] Amazon. *Regions, Availability Zones, and Local Zones*. URL: <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html>. (accessed: 21.05.2020).
- [65] Amazon. *Run Serverless Kubernetes Pods Using Amazon EKS and AWS Fargate*. URL: <https://aws.amazon.com/about-aws/whats-new/2019/12/run-serverless-kubernetes-pods-using-amazon-eks-and-aws-fargate/>. (accessed: 20.05.2020).

- [66] Amazon. *Ubuntu Amazon EKS-optimized AMIs*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/eks-partner-amis.html>. (accessed: 22.05.2020).
- [67] Amazon. *Using Cost Allocation Tags*. URL: <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/cost-alloc-tags.html>. (accessed: 3.06.2020).
- [68] Amazon. *VPCs and subnets*. URL: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html. (accessed: 25.05.2020).
- [69] Amazon. *What is Amazon EKS?* URL: <https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html>. (accessed: 18.05.2020).
- [70] Amazon. *Worker nodes*. URL: <https://docs.aws.amazon.com/eks/latest/userguide/worker.html>. (accessed: 20.05.2020).
- [71] Helm Author. *Official Helm page*. URL: <https://helm.sh/>. (accessed: 30.05.2020).
- [72] Kops Authors. *Addons*. URL: <https://github.com/kubernetes/kops/blob/v1.16.2/docs/operations/addons.md>. (accessed: 30.05.2020).
- [73] Kops Authors. *Cluster Templating*. URL: https://github.com/kubernetes/kops/blob/v1.16.2/docs/operations/cluster_template.md. (accessed: 30.05.2020).
- [74] Kops Authors. *Continuous Integration*. URL: https://github.com/kubernetes/kops/blob/v1.16.2/docs/continuous_integration.md. (accessed: 30.05.2020).
- [75] Kops Authors. *Description of Keys in config and cluster.spec*. URL: https://github.com/kubernetes/kops/blob/v1.16.2/docs/cluster_spec.md. (accessed: 2.06.2020).
- [76] Kops Authors. *Getting Started with kops on AWS*. URL: https://kops.sigs.k8s.io/getting_started/aws/. (accessed: 20.05.2020).
- [77] Kops Authors. *Gossip DNS*. URL: <https://kops.sigs.k8s.io/gossip/>. (accessed: 28.05.2020).
- [78] Kops Authors. *High Availability (HA)*. URL: https://github.com/kubernetes/kops/blob/v1.16.2/docs/operations/high_availability.md. (accessed: 30.05.2020).

- [79] Kops Authors. *IAM Roles*. URL: https://github.com/kubernetes/kops/blob/v1.16.2/docs/iam_roles.md. (accessed: 1.06.2020).
- [80] Kops Authors. *Images*. URL: <https://github.com/kubernetes/kops/blob/v1.16.2/docs/operations/images.md>. (accessed: 30.05.2020).
- [81] Kops Authors. *Installing*. URL: https://kops.sigs.k8s.io/getting_started/install/. (accessed: 18.05.2020).
- [82] Kops Authors. *Kops an Kubernetes versions on Github.com*. URL: <https://github.com/kubernetes/kops/blob/master/channels/stable>. (accessed: 29.05.2020).
- [83] Kops Authors. *Kops Golang package documentation*. URL: <https://pkg.go.dev/k8s.io/kops/pkg/apis/kops?tab=doc#ClusterSpec>. (accessed: 20.05.2020).
- [84] Kops Authors. *Kops official website*. URL: <https://kops.sigs.k8s.io>. (accessed: 20.04.2020).
- [85] Kops Authors. *Kops project page on Github.com*. URL: <https://github.com/kubernetes/kops>. (accessed: 18.05.2020).
- [86] Kops Authors. *Kops Releases & Versioning*. URL: <https://kops.sigs.k8s.io/welcome/releases/>. (accessed: 21.05.2020).
- [87] Kops Authors. *Kops releases on Github.com*. URL: <https://github.com/kubernetes/kops/releases>. (accessed: 27.05.2020).
- [88] Kops Authors. *kops validate cluster*. URL: https://github.com/kubernetes/kops/blob/v1.16.2/docs/cli/kops_validate_cluster.md. (accessed: 1.06.2020).
- [89] Kops Authors. *Kubernetes Addons and Addon Manager*. URL: <https://github.com/kubernetes/kops/blob/master/docs/operations/addons.md>. (accessed: 20.05.2020).
- [90] Kops Authors. *Labels*. URL: <https://github.com/kubernetes/kops/blob/v1.16.2/docs/labels.md>. (accessed: 30.05.2020).
- [91] Kops Authors. *Using A Manifest to Manage kops Clusters*. URL: https://kops.sigs.k8s.io/manifests_and_customizing_via_api/. (accessed: 20.05.2020).

- [92] Kops Authors. *Using A Manifest to Manage kops Clusters*. URL: https://github.com/kubernetes/kops/blob/v1.16.2/docs/manifests_and_customizing_via_api.md. (accessed: 30.05.2020).
- [93] Kubespray Authors. *Comparison*. URL: <https://github.com/kubernetes-sigs/kubespray/blob/master/docs/comparisons.md>. (accessed: 18.05.2020).
- [94] Kubespray Authors. *Kubespray project page on Github.com*. URL: <https://github.com/kubernetes-sigs/kubespray>. (accessed: 18.05.2020).
- [95] Minikube authors. *minikube project page on Github.com*. URL: <https://github.com/kubernetes/minikube>. (accessed: 18.05.2020).
- [96] The Kubernetes Authors. *kubectl*. URL: <https://kubernetes.io/docs/reference/kubectl/kubectl/>. (accessed: 30.05.2020).
- [97] Velero Authors. *Examples*. URL: <https://velero.io/docs/v1.4/examples/>. (accessed: 3.06.2020).
- [98] Paul Bakker. *One year using Kubernetes in production: Lessons learned*. URL: <https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned>. (accessed: 15.04.2020).
- [99] Bitnami. *Apache Helm Chart*. URL: <https://github.com/bitnami/charts/tree/master/bitnami/apache>. (accessed: 30.05.2020).
- [100] Amazon EKS Bob Wise General Manager. *eksctl – the EKS CLI*. URL: <https://aws.amazon.com/blogs/opensource/eksctl-eks-cli/>. (accessed: 19.05.2020).
- [101] Karen Bruner. *Guide to Designing EKS Clusters for Better Security*. URL: <https://www.stackrox.com/post/2020/03/guide-to-eks-cluster-design-for-better-security/>. (accessed: 5.06.2020).
- [102] Github.com user: cdenneen. *Cluster update EndPoint*. URL: <https://github.com/weaveworks/eksctl/issues/1558>. (accessed: 4.06.2020).
- [103] CNCF. *CNCF Survey 2017*. URL: <https://www.cncf.io/blog/2017/06/28/survey-shows-kubernetes-leading-orchestration-platform/>. (accessed: 20.04.2020).
- [104] CNCF. *CNCF Survey 2019*. URL: https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf. (accessed: 20.04.2020).

- [105] CNCF. *Services for CNCF Projects*. URL: <https://www.cncf.io/services-for-projects/>. (accessed: 15.05.2020).
- [106] CNCF. *With Kubernetes, the U.S. Department of Defense Is Enabling DevSecOps on F-16s and Battleships*. URL: <https://www.cncf.io/blog/2020/05/07/with-kubernetes-the-u-s-department-of-defense-is-enabling-devsecops-on-f-16s-and-battleships/>. (accessed: 16.05.2020).
- [107] Kubernetes Community. *Installing Kubernetes with kops*. URL: <https://kubernetes.io/docs/setup/production-environment/tools/kops/>. (accessed: 28.05.2020).
- [108] Kubernetes Community. *Advanced Kubernetes Scheduling*. URL: <https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes/>. (accessed: 20.06.2020).
- [109] Kubernetes Community. *Autoscaler on Github.com*. URL: <https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/autoscaler/cloudprovider/aws/examples/cluster-autoscaler-autodiscover.yaml>. (accessed: 3.06.2020).
- [110] Kubernetes Community. *Building large clusters*. URL: <https://kubernetes.io/docs/setup/best-practices/cluster-large/>. (accessed: 22.05.2020).
- [111] Kubernetes Community. *Cluster Autoscaler AWS example uses wrong host path for SSL certs*. URL: <https://github.com/kubernetes/autoscaler/issues/2139>. (accessed: 3.06.2020).
- [112] Kubernetes Community. *Cluster Networking*. URL: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>. (accessed: 10.04.2020).
- [113] Kubernetes Community. *Configure Liveness, Readiness and Startup Probes*. URL: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>. (accessed: 17.04.2020).
- [114] Kubernetes Community. *DaemonSet*. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>. (accessed: 30.05.2020).
- [115] Kubernetes Community. *Deployments*. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>. (accessed: 30.05.2020).
- [116] Kubernetes Community. *kubeadm page on Github.com*. URL: <https://github.com/kubernetes/kubeadm>. (accessed: 16.05.2020).

- [117] Kubernetes Community. *Kubelet*. URL: <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>. (accessed: 12.06.2020).
- [118] Kubernetes Community. *Kubernetes 1.18: Fit & Finish*. URL: <https://kubernetes.io/blog/2020/03/25/kubernetes-1-18-release-announcement/>. (accessed: 21.05.2020).
- [119] Kubernetes Community. *Kubernetes Components*. URL: <https://kubernetes.io/docs/concepts/overview/components/>. (accessed: 10.04.2020).
- [120] Kubernetes Community. *Kubernetes Concepts*. URL: <https://kubernetes.io/docs/concepts/>. (accessed: 3.06.2020).
- [121] Kubernetes Community. *Kubernetes Metrics Server*. URL: <https://github.com/kubernetes-sigs/metrics-server>. (accessed: 5.06.2020).
- [122] Kubernetes Community. *Node Allocatable*. URL: <https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/#node-allocatable>. (accessed: 22.05.2020).
- [123] Kubernetes Community. *Troubleshoot Applications*. URL: <https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application/>. (accessed: 3.06.2020).
- [124] Kubernetes Community. *Troubleshoot Clusters*. URL: <https://kubernetes.io/docs/tasks/debug-application-cluster/debug-cluster/>. (accessed: 6.06.2020).
- [125] Kubernetes Community. *Troubleshoot Clusters*. URL: <https://kubernetes.io/docs/tasks/debug-application-cluster/troubleshooting/>. (accessed: 6.06.2020).
- [126] Kubernetes Community. *Using Minikube to Create a Cluster*. URL: <https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/>. (accessed: 10.04.2020).
- [127] Kubernetes Community. *Web UI (Dashboard)*. URL: <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>. (accessed: 12.06.2020).
- [128] Ewa Czechowska. *docker-k8s-aws-dojo page on Github.com*. URL: <https://github.com/kudulab/docker-k8s-aws-dojo>. (accessed: 30.05.2020).

- [129] Ewa Czechowska. *Dojo page on Github.com*. URL: <https://github.com/kudulab/dojo>. (accessed: 30.05.2020).
- [130] Mike Danese and Tim Hockin. *kube-aws project page on Github.com*. URL: <https://github.com/kubernetes-incubator/kube-aws>. (accessed: 18.05.2020).
- [131] IBM Cloud Education. *etcd*. URL: <https://www.ibm.com/cloud/learn/etcd>. (accessed: 12.06.2020).
- [132] Justin Ellingwood. *An Introduction to Continuous Integration, Delivery, and Deployment*. URL: <https://www.digitalocean.com/community/tutorials/an-introduction-to-continuous-integration-delivery-and-deployment>. (accessed: 15.05.2020).
- [133] Google. *Google Kubernetes Engine*. URL: <https://cloud.google.com/kubernetes-engine>. (accessed: 16.05.2020).
- [134] Google. *Troubleshooting*. URL: <https://cloud.google.com/kubernetes-engine/docs/troubleshooting>. (accessed: 6.06.2020).
- [135] Graylog. *Graylog official website*. URL: <https://www.graylog.org/>. (accessed: 15.04.2020).
- [136] Graylog. *IMPROVING KUBERNETES CLUSTERS' EFFICIENCY WITH LOG MANAGEMENT*. URL: <https://www.graylog.org/post/improving-kubernetes-clusters-efficiency-with-log-management>. (accessed: 15.04.2020).
- [137] Gruntwork. *Comprehensive Guide to EKS Worker Nodes*. URL: <https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#per-pod-configurable-eviction-behavior-when-there-are-node-problems-alpha-feature>. (accessed: 20.06.2020).
- [138] Gruntwork. *How to deploy a production-grade Kubernetes cluster on AWS*. URL: <https://blog.gruntwork.io/how-to-build-an-end-to-end-production-grade-architecture-on-aws-part-1-eae8eeb41fec>. (accessed: 25.05.2020).
- [139] Arun Gupta. *Manage Kubernetes Clusters on AWS Using Kops*. URL: <https://aws.amazon.com/blogs/compute/kubernetes-clusters-aws-kops/>. (accessed: 28.05.2020).
- [140] Rajesh Gupta. *AWS EKS 0/1 nodes are available. 1 insufficient pods*. URL: <https://stackoverflow.com/a/58988063/4457564>. (accessed: 7.06.2020).

- [141] Hashicorp. *Terraform official website*. URL: <https://www.terraform.io/>. (accessed: 18.05.2020).
- [142] Hashicorp. *Terraform official website*. URL: <https://www.terraform.io/intro/vs/cloudformation.html>. (accessed: 18.05.2020).
- [143] heyicanuserreddit. *EKS Backup & Restore*. URL: https://www.reddit.com/r/aws/comments/bx92v5/eks_backup_restore/. (accessed: 3.06.2020).
- [144] Kelsey Hightower. *Kubernetes The Hard Way page on Github.com*. URL: <https://github.com/kelseyhightower/kubernetes-the-hard-way>. (accessed: 18.05.2020).
- [145] isokie. *create cluster hangs on waiting for nodes to become ready*. URL: <https://github.com/weaveworks/eksctl/issues/1482>. (accessed: 5.06.2020).
- [146] Docker Official Images. *Ubuntu Docker Image*. URL: https://hub.docker.com/_/debian. (accessed: 15.05.2020).
- [147] Docker Official Images. *Ubuntu Docker Image*. URL: https://hub.docker.com/_/ubuntu. (accessed: 15.05.2020).
- [148] Chef Software Inc. *Official Chef website*. URL: <https://www.chef.io/>. (accessed: 16.04.2020).
- [149] Docker Inc. *Docker Documentation*. URL: <https://docs.docker.com/get-started/part2/>. (accessed: 15.05.2020).
- [150] Docker Inc. *Dockerize an SSH service*. URL: https://docs.docker.com/engine/examples/running_ssh_service/. (accessed: 15.05.2020).
- [151] ThoughtWorks Inc. *Official GoCD website*. URL: <https://www.gocd.org/>. (accessed: 15.05.2020).
- [152] Github.com user: janavenkat. *PrivateNetworking for worker nodegroup not working when publicAccessCIDRs enabled*. URL: <https://github.com/weaveworks/eksctl/issues/2054>. (accessed: 5.06.2020).
- [153] Jenkins. *Official Jenkins website*. URL: <https://www.jenkins.io/>. (accessed: 15.05.2020).
- [154] Simon John. *A Transition From Monolith to Microservices*. URL: <https://dzone.com/articles/a-transition-from-monolith-to-microservices>. (accessed: 15.05.2020).

- [155] Vladik Khononov. *'eksctl' Stuck on Waiting for Nodes to Join the Cluster*. URL: <https://blog.doit-intl.com/eksctl-stuck-on-waiting-for-nodes-to-join-the-cluster-c3670aa74487>. (accessed: 5.06.2020).
- [156] Kubernetes maintainer and SIG-Cluster-Lifecycle member. *A Stronger Foundation for Creating and Managing Kubernetes Clusters*. URL: <https://kubernetes.io/blog/2017/01/stronger-foundation-for-creating-and-managing-kubernetes-clusters/>. (accessed: 18.05.2020).
- [157] Microsoft. *Azure Kubernetes Service (AKS)*. URL: <https://azure.microsoft.com/en-us/services/kubernetes-service/>. (accessed: 16.05.2020).
- [158] Microsoft. *Health Endpoint Monitoring pattern*. URL: <https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring>. (accessed: 17.04.2020).
- [159] Mihajlo Milenovi. *How to Monitor Kubernetes Cluster with Prometheus and Grafana*. URL: <https://linuxide.com/linux-how-to/monitor-kubernetes-cluster-prometheus-grafana/>. (accessed: 15.04.2020).
- [160] Krishna Modi. *How to Setup a Perfect Kubernetes Cluster using KOPS on AWS*. URL: <https://medium.com/faun/how-to-setup-a-perfect-kubernetes-cluster-using-kops-in-aws-b616bdfae013>. (accessed: 30.05.2020).
- [161] Diego Najar. *Kops: Kubernetes cluster with Autoscaling on AWS*. URL: <https://varlogdiego.com/kubernetes-cluster-with-autoscaling-on-aws-and-kops>. (accessed: 3.06.2020).
- [162] Mark O'Connor. *Kops Create Cluster fails with gossip in AWS Linux*. URL: <https://stackoverflow.com/a/54858527/4457564>. (accessed: 30.05.2020).
- [163] Helm chart owners. *kube2iam Helm chart*. URL: <https://github.com/helm/charts/tree/master/stable/kube2iam>. (accessed: 1.06.2020).
- [164] Liran Polak. *EKS Done Right – From Control Plane To Worker Nodes*. URL: <https://spot.io/blog/eks-done-right-from-control-plane-to-worker-nodes/>. (accessed: 4.06.2020).
- [165] Daniele Polencic. *Kubernetes Chaos Engineering: Lessons Learned — Part 1*. URL: <https://learnk8s.io/blog/kubernetes-chaos-engineering-lessons-learned>. (accessed: 25.05.2020).

- [166] Prometheus. *Github page with Prometheus project*. URL: <https://github.com/prometheus/prometheus>. (accessed: 15.04.2020).
- [167] Prometheus. *Prometheus official website*. URL: <https://prometheus.io/>. (accessed: 15.04.2020).
- [168] Inc. Red Hat. *Official Ansible website*. URL: <https://www.ansible.com/>. (accessed: 16.04.2020).
- [169] Github user: rifelpet. *Add CloudLabels tags to additional AWS resources*. URL: <https://github.com/kubernetes/kops/pull/8903>. (accessed: 7.06.2020).
- [170] Inc. SaltStack. *Official SaltStack website*. URL: <https://www.saltstack.com/>. (accessed: 16.04.2020).
- [171] Github user: schollii. *Not all resources get tags from cloudLabels in kops 1.13*. URL: <https://github.com/kubernetes/kops/issues/8792>. (accessed: 7.06.2020).
- [172] Sam Stephenson and bats-core organization. *Official Bats-core page on Github.com*. URL: <https://github.com/bats-core/bats-core>. (accessed: 30.05.2020).
- [173] Tobias Sturm. *How To Use Aws Cloudwatch Logs With Kubernetes Kops*. URL: <http://www.tobiassturm.de/2017/04/20/how-to-use-aws-cloudwatch-logs-with-kubernetes-kops>. (accessed: 1.06.2020).
- [174] Alvise Susmel. *Create a High-Availability Kubernetes Cluster on AWS with Kops*. URL: <https://www.poeticoding.com/create-a-high-availability-kubernetes-cluster-on-aws-with-kops/>. (accessed: 29.05.2020).
- [175] VMware Tanzu. *Velero Helm Chart on Github.com*. URL: <https://github.com/vmware-tanzu/helm-charts/tree/master/charts/velero>. (accessed: 3.06.2020).
- [176] VMware Tanzu. *Velero Plugin For AWS on Github.com*. URL: <https://github.com/vmware-tanzu/velero-plugin-for-aws>. (accessed: 3.06.2020).
- [177] Jr. Timothy W. Gravier. *What is RAID and why should you want it?* URL: https://raid.wiki.kernel.org/index.php/What_is_RAID_and_why_should_you_want_it%3F. (accessed: 12.06.2020).
- [178] Sophie Turol, Carlo Gutierrez, and Sergey Matykevich. *A Multitude of Kubernetes Deployment Tools: Kubespray, kops, and kubeadm*. URL: <https://www.altoros.com>.

- com/blog/a-multitude-of-kubernetes-deployment-tools-kubespray-kops-and-kubeadm/?utm_campaign=SMM-Val_K8sTools%20Post-Free-Promo-Medium&utm_source=Medium. (accessed: 20.05.2020).
- [179] WeaveWorks. *CloudWatch logging*. URL: <https://eksctl.io/usage/cloudwatch-cluster-logging/>. (accessed: 27.05.2020).
- [180] WeaveWorks. *Creating and managing clusters*. URL: <https://eksctl.io/usage/creating-and-managing-clusters/>. (accessed: 27.05.2020).
- [181] WeaveWorks. *Eksctl configuration examples on Github.com*. URL: <https://github.com/weaveworks/eksctl/tree/master/examples>. (accessed: 20.05.2020).
- [182] WeaveWorks. *Eksctl releases on Github.com*. URL: <https://github.com/weaveworks/eksctl/releases>. (accessed: 27.05.2020).
- [183] WeaveWorks. *Official eksctl website*. URL: <https://eksctl.io/>. (accessed: 20.05.2020).
- [184] WeaveWorks. *Production Ready Checklists for Kubernetes*. URL: <https://go.weave.works/production-ready-kubernetes-checklist.html?LeadSource=Website%20-%20General&CampaignID=7014M000001cEod>. (accessed: 15.04.2020).
- [185] WeaveWorks. *Troubleshooting*. URL: <https://eksctl.io/usage/troubleshooting/>. (accessed: 6.06.2020).
- [186] WeaveWorks. *VPC Networking*. URL: <https://eksctl.io/usage/vpc-networking/>. (accessed: 24.05.2020).
- [187] WeaveWorks. *Your Guide to a Production Ready Kubernetes Cluster*. URL: <https://go.weave.works/WP-Production-Ready.html?LSD=Homepage&Source=Website%20-%20General>. (accessed: 15.04.2020).
- [188] Daniel Weibel. *Architecting Kubernetes clusters — choosing a worker node size*. URL: <https://learnk8s.io/kubernetes-node-size>. (accessed: 22.05.2020).
- [189] Haimo Zhang. *Learning Kubernetes on EKS by Doing Part 1— Setting Up EKS*. URL: <https://medium.com/faun/learning-kubernetes-by-doing-part-1-setting-up-eks-in-aws-50dcf7a76247>. (accessed: 4.06.2020).