UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Mikko Korhonen

ANALYZING RESOURCE USAGE ON MULTI TENANT CLOUD CLUSTER FOR INVOICING

Master's Thesis
Degree Programme in Computer Science and Engineering
November 2017

Korhonen M. (2017) Analyzing resource usage on multi tenant cloud cluster for invoicing. University of Oulu, Degree Programme in Computer Science and Engineering. Master's thesis, 49 p.

ABSTRACT

Software development is a complex and challenging process. Throughout the history of software development organizations have tried to implement various methodologies to ease this process. Technological solutions like microservice based architecture and virtualization have spun that embrace Agile thinking principles. Virtualization technologies have made running multi environment solutions very appealing and significantly reduced host management. Lightweight and isolated application containers have taken this even one step further. They have become so popular that systems for managing the life cycles of these containers, also known as container orchestration systems, have emerged.

Codemate Ltd, a company specialized in mobile and web development, also researched a possibility of running a cloud based cluster computing system for application containers. The company decided to build this system on top of one of the most prominent container orchestration systems called Kubernetes. A shift into this kind of system proposed various challenges, billing a customer being one of them.

The main focus of this thesis is to describe the process of designing and implementing a software solution that would analyze multi-tenant cloud cluster resource usage metrics on customer level and then use the information to bill customers accordingly. This thesis also presents all the necessary background information to what has gone into building this kind of tool. It also delves into challenges that the process presented. Finally this thesis presents and evaluates the tool that was created as a part of this process.

Keywords: cloud computing, cluster, kubernetes, monitoring

Korhonen M. (2017) Pilvipohjaisen laskentaklusterin resurssien käytön analysoiminen laskuttamista varten. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 49 s.

TIIVISTELMÄ

Ohjelmistojen kehitys on monimutkainen ja haastava prosessi. Sen historian aikana organisaatiot ovat pyrkineet kehittämään erilaisia menetelmiä tämän prosessin helpottamiseksi. On syntynyt teknologisia ratkaisuja kuten microservice arkkitehtuuri ja virtualisointi, jotka toteuttavat agile-ajattelun periaatteita. Virtualisointitekniikat ovat tehneet usean ympäristön ratkaisuista erittäin houkuttelevia ja vähentäneet huomattavasti ylläpitovaatimuksia. Kevyet ja eristetyt kontit ovat vieneet tämän askeleen pidemmälle. Niiden suosion myötä on syntynyt myös niin sanottuja konttien orkestrointi työkaluja, joilla konttien elinkaarta voidaan hallita.

Codemate Ltd, joka on mobiili ja web-kehitykseen erikoistunut yritys, tutki myös pilvipohjaisia konttiteknologiaan perustuvia laskentaklusteri ratkaisuja. Yritys päätti rakentaa tällaisen systeemin hyödyntäen Kubernetesta, joka on yksi lupaavimmista konttien orkestrointi järjestelmistä. Siirtyminen tämänkaltaiseen järjestelmään on luonut erilaisia haasteita, kuten asiakkaiden laskuttamisen hoitaminen.

Tämän diplomityön päätarkoitus on kuvata sitä suunnittelu ja toteutus prosessia, jonka tuotoksena on työkalu pilvipohjaisen laskentaklusterin resurssien analysoimiseen ja tämän tiedon hyödyntämiseen asiakkaiden laskuttamisessa. Tässä työssä esitellään myös tarvittavat taustatiedot tällaisen työkalun rakentamista varten, prosessin aikana kohdatut haasteet, sekä lopuksi esitellään ja arvioidaan luotu työkalu.

Avainsanat: pilvilaskenta, klusteri, kubernetes, monitorointi

TABLE OF CONTENTS

FOREWORD

First and foremost I would like to thank my employer Codemate Ltd for providing me this opportunity to write my thesis on state-of-the-art technology. I would like to thank my work colleges, especially software architects Juhamatti Niemelä for suggesting this topic and Tuomas Mäkinen for guiding me into miraculous world of Kubernetes.

I would like to thank Denzil Ferreira for supervising and guiding me through this thesis. I would like to thank my school mates, especially the ones that I had the pleasure to spent most time with, I learned so much from you.

Last but definitely not least, I would like to thank my parents Pirjo and Matti for all the support I have gotten during the years which finally lead to this moment.

ABBREVIATIONS

API	Application programming interface
AWS	Amazon Web Services
CD	Continuous delivery
CDN	Content delivery network
CI	Continuous integration
CPU	Central processing unit
CSS	Cascading Style Sheets
DevOps	Development and operations
EBS	Amazon Elastic Block Store
EC2	Amazon Elastic Compute Cloud
EFS	Amazon Elastic File System
HTTP	Hypertext Transfer Protocol
I/O	Input/output
JSON	JavaScript Object Notation
OS	Operating system
RAM	Random-access memory
REST	Representational State Transfer
S3	Amazon Simple Storage Service
SLA	Service level agreement
UI	User interface
UX	User experience
URL	Uniform Resource Identifier
VPS	Virtual private server
YAML	YAML Ain't Markup Language

1. INTRODUCTION

1.1. Codemate Ltd

Codemate Ltd[1], henceforth referred to as the company, is an international software development company founded in 2007. The company is specialized in developing web systems and mobile applications as a service. At the time of writing, the company employs approximately 70 people in five distinct locations: three offices in Finland including Helsinki, Vuokatti and its headquarters in Oulu, one office in Bangkok, Thailand and one in Dhaka, Bangladesh.

The company offers a full range of services for crafting digital solutions including design, development, testing and maintenance. The company has a wide variety of customers from different business sectors ranging in size from small to large.

1.2. My role at the company

I am working at the company as a Software Engineer specialized in full-stack web development. Being a full-stack developer requires being comfortable working with both backend and frontend web technologies. My work requires extensive knowledge of various technologies, understanding how each part of modern software development process works and how software is deployed from start to finish. During my time at the company I have been fortunate to work alongside bright minds on interesting customer projects from different business sectors. These projects have generated value for the customers either by solving a real world problem or by enhancing existing work flows and methods. I have learned a lot about both development and business side of software development and what kind of challenges are often faced.

1.3. Current situation

The company is specialized in developing mobile and web applications. A typical web application consists of a frontend, a backend service and a database. The nature of this kind of solution requires it be deployed somewhere to make it accessible. Traditionally this meant acquiring a physical server which would handle running of entire application. This kind of approach is both costly and time consuming.

With the advancement of virtualization technologies so called cloud based solutions have emerged. From the customers (of cloud providers) perspective this brings multiple upsides. Mainly in the form of easily scaling resources based on the need and only paying for the resources necessary. The company is also a firm believer in cloud base technology and most of the projects are deployed to AWS which is a cloud service platform by Amazon.

Typically each project is setup under its own account with a separate infrastructure. This approach has a couple of drawbacks from both customer's and the company's point of view. First, server deployment in general is rather inefficient even with projects leveraging cloud based solutions. This is often caused by the fact that it is good practice to have extra resources in store to handle unexpected spikes in user traffic. From

customer's point of view this means paying for resources that they use only for a fraction of time. Customers also often have a support contract with the company which takes care of not only scaling the infrastructure on demand, but also updates and new features. From the company's point of view, having to setup and maintain multiple different infrastructures adds unnecessary complexity especially as the customer base grows. Due to these facts the company started to research into alternative solutions and a possibility of having company controlled multi-tenant cloud cluster to which all of the future customer projects would be deployed to.

1.4. Future plans

Because of the reasons mentioned in the previous section the company conducted a research on current cloud cluster solutions. Based on the knowledge at the given time the company identified that this multi-tenant cloud cluster was to support following features:

- Provide isolation to support multiple customers
- Easily scale up and down resources and deployments, e.g., storage, bandwidth, or processing power
- Ability to use variety of software and configurations
- Handle secrets and other sensitive information, like passwords and API keys

As a result of the research company decided to built the cluster on top of Kubernetes which is an open source platform for automating deployment, scaling and operations of applications containers. This was the beginning of a company-wide transition that introduced multiple challenges. The biggest one of them being the billing of customers. This was previously handled by the cloud provider in question, but in the future this would be on the company's responsibility. A billing model that would be both fair for the customers but also profitable for the company had to be created. This was a major hurdle that needed to be solved before the cluster could be prepared for production use. The main purpose of this thesis is to present a solution to that problem.

1.5. Contribution

The company assigned me the task of designing and creating a tool that would be used to bill customers of the new cluster. The tool would need to collect and analyze resource usage on a customer level. Also a pricing model would need to be constructed that is both fair and reflects reality as close as possible. Finally these would be combined into a single, easy-to-use tool. This thesis represent the work that has gone into building this very tool including background information, design and development process, assessment and limitations.

This is thesis is structured as follows: section 2 describes the underling problem in

more detailed manner, section 3 gives context by explaining how modern software development and deployment works, section 4 presents a solution to the problem and explains reasons behind the choice, section 5 is a literature review, section 6 presents the primary work of the thesis, section 7 presents evaluation of the work and finally section 8 presents limitations of the work conducted.

2. PROBLEM STATEMENT

This section describes the motives for which the company decided to build its own cloud computing cluster. It starts by giving a brief introduction about the current state of modern software development. This is followed by a description of the underlying problems from both business and development perspective.

2.1. Motivation

Web application is a client-server software in which the client runs in a web browser. These kind of applications have gained a lot of popularity during the recent years over the traditional desktop applications. Web applications offer seamless upgrades and great availability thanks to advancement of hand-held devices and web technologies. Also the developing costs for these type of applications are often lower compared to more traditional desktops applications especially if one needs to target multiple platforms.

Typical web application consists of two parts: frontend which provides the user interface and backend where the business logic resides. These high level concepts can be split into smaller more specific parts, e.g backend usually utilizes a database where data is stored. Depending on the project backend might also consist of other services like cache storage, authentication provider or mail server.

The company also handles deployments of the finished products. History has proved Amazon Web Services to be very efficient and robust platform and thus it has become the company's number one choice for application deployments. However, application deployments will not come without challenges. It is common practice to use multiple release environments, typically development to which new software builds are rapidly deployed, staging which acts as a pre-production testing environment and production environment which used by the end users. To allow so called rolling updates, that are software updates without downtime, two or more instances are required. Optimizing resources utilization also poses challenges even when using virtualization technologies that allow easier matching for required resources. This is not always an easy task because one should have enough spare resources in case of unexpected traffic spikes. However most of the time server utilization rate and load might be low but the resources still cost money. Each customer typically have their own instances on which the applications are ran. More instances mean more configuration to manage and more maintenance to do. At a large scale this task can become very tedious to manage. These problems can be generalized under two categories, business and development problems.

2.2. Business problems

On average server utilization rates are considerably low, even low as 20% [2][3]. This means businesses end up paying money for resources that they never use.

Scaling instances up and down is also difficult and needs to be planned in advance. If a business expects a large spike in user traffic during the holiday season they usually

need to set up the extra capacity in advance. Once the spike is over the extra resources remain underutilized and costing money.

Maintaining own datacenter or a fleet of hardware requires dedicated IT support team. They need to deal with various configuration problems in the system running different versions of software on different operating systems, each with possibly a different level of patches installed on each of them. These servers often end up being treated pets instead of cattle, meaning you nurse them back to health instead of letting of of them and replacing it with another one[4].

2.3. Development problems

Also developers face many challenges when creating these systems. As the number of dependencies and complexity of configuration grows it gets harder and harder to keep up and have all the systems in the same state. Software might run in a different way or with a different configuration when it's run locally on a developer's machine versus a testing or production environment. This causes infamous "works on my machine" situations which are very difficult to track down and fix.

A lot of business software is often complex and monolithic. Code is highly dependent on other parts of the system or external libraries. This kind of software is not only hard to maintain but also slow and hard to deploy. Change to code in one part of the system might break another completely unrelated part of the system.

All of this adds up and setting up the system gets very time consuming. This impacts both testing and deploying of the software. Fixing the system might take days instead of hours which in the worst case ends up costing a lot of money.

Microservice architecture and containerization are two methods that aim to solve the problems mentioned above. Utilizing these two methods we can now develop small, independent and highly scalable services that are easy to deploy and run in a consistent way across all different environments.

3. CONCEPTS INTRODUCTION

This section provides necessary background information to understand the underlying concepts and ideologies behind modern software development. Kubernetes relies heavily on these concepts which is why they are important to internalize them.

3.1. Organization

Software development is a challenging process. According to a recent study[5] 55% of the 126 IT professionals surveyed have experienced a failed project in the last 12 months. Throughout the history of software development, dating back to as early as the 60's[6], organizations have tried to implement various methodologies to ease this process. This decades-long journey to find a process that improves productivity and quality has brought us many methodologies of which developed during the last two decades are all based on the agile ideology.

3.1.1. Agile

Agile manifest was introduced in 2001 and since it has become increasingly popular among software development projects. Compared to previously widely used waterfall model (that is still largely used by many organizations, especially governments) it introduced the principle that frequent alteration in requirements and solutions should be allowed through efficient communication inside the team. It embraces short development cycles and the "fail fast" ideology which allows more frequent software deliveries to the customer. To ease this process development practices like continuous integration and delivery were developed. These processes are described in more detail later in the thesis.

3.1.2. Conway's Law

Conway's Law[7] states *"Organizations that design systems are constrained to produce designs which are copies of the communication structures of these organizations"*. One expression of this is that a company having separate frontend, backend and database teams have a tendency to come up with solutions using three-tier architecture. Despite organization using Agile methodologies, this kind of role based categorization does not promote Agile principles. The authors of *Exploring the Duality between Product and Organizational Architectures*[8] found that loosely coupled organizations create more modular and less coupled system than tightly coupled organizations. DevOps, a concept that blends development and deployment organization and tools, can be seen as natural extension to Agile development. DevOps is described in more detailed in section 3.3.1.

3.2. Development

To embrace the previously introduced principles and methodologies various technological solutions have been developed. Those solutions are discussed in more detail in this section.

3.2.1. Microservice architecture

As the codebase grows it starts to get harder to introduce new features as it is difficult to know where the change needs to be made and which other parts of the system it possibly affects. All too often these changes tend to break completely different and unrelated part of the system. This is one of the problems that microservice architecture aims to tackle. Microservices take the *Single Responsibility Principle*[9] to service level by putting service boundaries on business boundaries. By creating single purpose services and *doing one thing and doing it well*[10] these kind of difficulties can be avoided.

Microservices are small, autonomous services that work together [11]. They communicate between each other using APIs each of them expose. They need to be able to change independently of each other and to be deployed without interfering with one another. Microservices are technology agnostic solution so they also help organizations to embrace new technologies, use different programming languages and to deliver software faster.

Sam Newman identified seven key benefits of using microservices in his book *Building Microservices*[11]. Next, a brief summary of each one of them is given.

Technology heterogeneity

As previously mentioned, microservice architecture is technology agnostic solution. Services communicate between each other using well defined APIs which enables technology decisions to be made on service level. This way we can use the best pick for each service rather than using one-size-fits-all approach.

Resilience

With a monolithic system the whole system is affected in case of a failure. With a system using microservices the failing component can be easily identified and isolated with rest of the system working as usual.

Scaling

With a large monolithic system you need scale everything together. When system functionality has been split into a smaller services we only have to scale specific services and rest of the system can run on less powerful and cheaper hardware.

Easy of deployment

When one part of the monolithic application is changed the whole application needs to be deployed. With microservices, only change to a single service is required and can be deployed independently from rest of the system.

Organizational alignment

It is easier for organization to handle resource allocation when the application is split into small logical pieces. Small teams can work on a single component and circulated more easily.

Composability

Microservices allow easy reuse of functionalities. Monolithic systems tends to be very fine grained and changing the behavior of certain functionality can be very cumbersome.

Optimizing for replaceability

Replacing or re-writing a part of the system is much easier on microservice system than on a large monolithic application. This also avoids getting into a situation where the system has some old and fragility parts that no one dares to touch.

3.2.2. Virtualization

Handling multiple environments efficiently has always been challenging in software development. As a software developer you want to set up your local development environment as quickly as possible. The application you are working with most likely requires a certain version of programming language used, has multiple external dependencies and might require some additional configuration like environmental variables used by the application. In practice the development environment should always match the production environment to avoid any unnecessary problems caused by environmental differences. However, in reality you might end up running a bit different configuration on your development environment due to time and money constrains. Despite having to occasionally make these compromises you want to keep the environmental differences to minimum. Setting up multiple environments is a challenge in itself. Previously this meant ordering new hardware and making the deployment on bare metal. The hardware itself costs money but someone also has to make the setup and then handle maintenance of it. Luckily today the advancement of virtualization technologies has significantly reduced the overhead of host management. Virtualization has made running multiple environments very appealing to the masses and greatly decreased the level of commitment that was previously required.

There are different levels of virtualization. Here we are focusing on the *Type 2 virtualization* where the hypervisor runs on the host operating system. Hypervisor is essential a program that enables running of multiple virtual machines on a single piece of hardware. Firstly, it handles the allocation of resources like CPU and memory

for each virtual machine and secondly, it acts as a control layer allowing interaction between virtual machine and the host OS. The problem with this kind of virtualization is the hypervisor creating extra overhead. Hypervisor needs to set aside some resources for it to be able to work. The more hosts the hypervisors manages the greater the overhead grows. Luckily there is a popular alternative choice available called Linux containers.

Containers

Containers are isolated and lightweight application containers that offer easy and fast deployment. Container technology itself is not a new invention, but it has only become mature enough during the last few years thanks to development progress made to some kernel features. Most objects in Unix systems are globally visible to all users. This causes the lack of configuration isolation meaning that multiple applications can have conflicting requirements for system-wide configuration settings. This can be problematic as many modern applications often require different versions of the same library. System administrators have tried to enforce resource isolation by installing each application to a separate copy of operating system, like virtual machine. Virtual machines excel at isolation as the communication with the outside world is very limited. However, virtual machines always add extra overhead as sharing data with other hosts or hypervisor requires fairly expensive marshaling and hypercalls. Linux containers however modify existing operating system to provide isolation instead of running another copy of full-blown operating system. They are built on the kernel namespaces feature that allows creating separate instances of previously global namespaces. It is used to create an isolated container that has no visibility or access to objects outside the container. Processes running inside the container appear to be running on a normal Linux system although they are sharing the underlying kernel with processes located in other namespaces. This way the application does not waste RAM on redundant management processes and generally consumes less RAM than equivalent system inside virtual machine. A visualized comparison of standard Type 2 virtualization and Linux containers can be seen in Figure 1.

The memory and CPU consumption of containers is limited by using the Linux control groups (*cgroups*) subsystem. Because a containerized Linux system only has one kernel and the kernel has full visibility into the containers there is only one level of resource allocation and scheduling. Container cannot access what it cannot see which greatly increases security. Root inside the container is not treated as root outside the container. The primary type of security vulnerability in containers is system calls that are not namespace aware and thus can introduce accidental leakage between containers. There are several management tools for containers of which Docker is the most well-known. A key feature of Docker is layered filesystem images that allows reuse of these layers between containers. This reduces the disk image size and generally Docker disk images are much smaller than equivalent virtual machine images. Containers also boot much faster than virtual machines as they do not need to boot another copy of the operating system. Research [12] has shown that Docker equals or exceeds virtual machine performance in most of the cases.

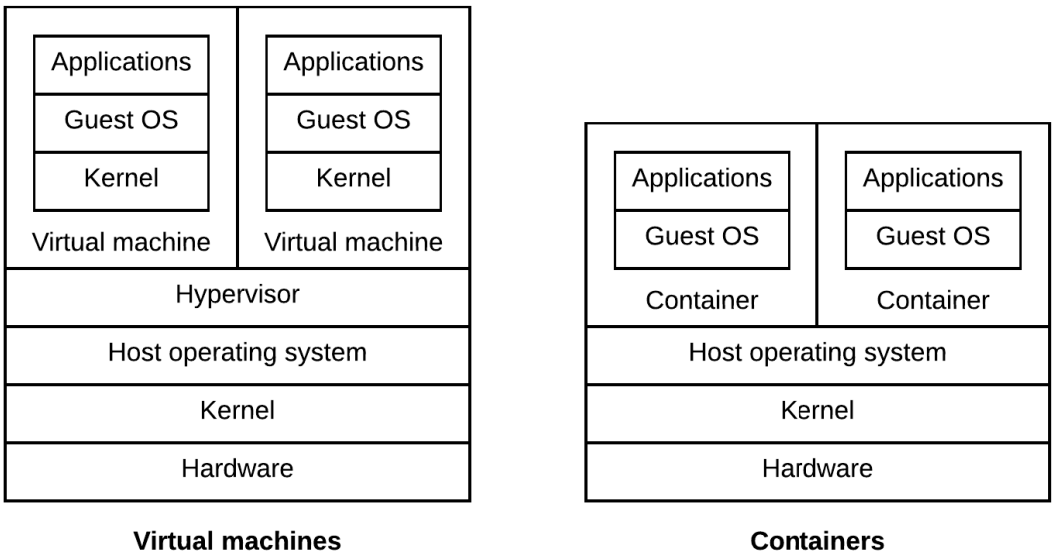

Figure 1. A comparison of virtual machines and containers

3.3. Deployment

Software deployment is a set of processes where software is build, tested and deployed to a chosen environment. Since the adoption of Agile methodologies there has been a focus to automate these tasks. Next, these tasks are discussed in more details and how they benefit software development

3.3.1. DevOps

Operations have been a vital part of software development since the dawn of the computer age. At the beginning, doing operations was a task of developers. In the 60's these two started to separate into mainframe operators and users, and during the 80's and 90's personal computers gave a birth to modern IT operations in form of system and network administrators [13]. Since then these two started to separate even more as developers laid focus on their tasks but at the same time someone had to keep the operations running.

The fast-paced advancement of Internet has created the need of scaling infrastructure at will. Managing a high number of servers manually is not sensible thus automation has become a key tool for people working at operations. Virtualization platforms have played a big role in helping us to achieve this. We are now able to fire up and provision our machines in a reproducible and programmable manner. Physical machines are not disappearing anywhere but as we are shifting towards more and more virtualized environments they are no longer the only thing system operators have to worry. Because of this, code has become the only reasonable way to manage this kind of mixed infrastructure. Operations have again become a part development. In a sense we are seeing a transition back to the roots; rather than siloing the tasks of developers and operators we should embrace collaboration between these two. This is what *DevOps* is all about.

Continuous Integration (CI) and Continuous Delivery (CD) are key concepts of successful DevOps. In the next subsections both of these concepts are explained in more details.

3.3.2. Continuous integration

Continuous Integration is a practice where developers frequently push their code changes to a centralized version control which then builds and runs tests automatically. CI encourages developers to merge their individual changes with each others as early as possible. This way everyone in the team can be kept in sync and made sure that the new changes properly integrate with existing codebase.

It is common that in the beginning of this process artifacts (e.g. application containers) are created that are used later in the pipeline for further validation[11]. For example, first an application container is build, which is then scanned for vulnerable packages, unit tested and finally tested for integration. This saves both time and resources of not having to build the same artifacts over and over again. Other benefits of CI include getting fast feedback, avoiding large merge conflicts and catching bugs earlier.

3.3.3. Continuous delivery

Continuous Delivery is a process of frequently deploying new versions of application to a selected environment (e.g. development, staging or production). It is an extension to CI pipeline as seen in Figure 2. To achieve this each step from check-in to production needs to be modeled and automated properly[11]. Once this has been implemented CD allows to get constant feedback on the production readiness as each check-in is treated as a release candidate. The main goal of CD is to offer consistent stream for pushing new features out to customer as soon as possible.

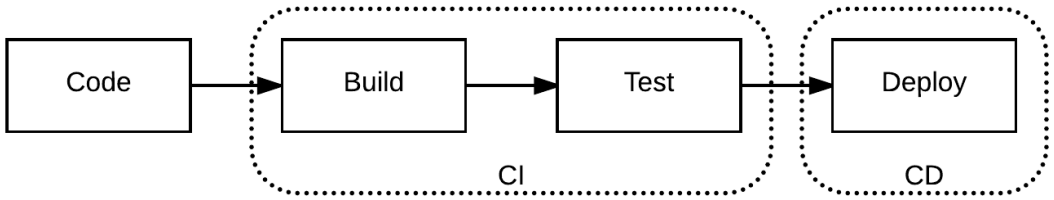

Figure 2. Simplified CI/CD pipeline

3.3.4. Logging and monitoring

Splitting the system into smaller, more fine-grained components introduces a number of advantages as it has been previously discussed. However, when it comes to moni-

toring and logging this kind of system we are faced with a challenge. When problems arise with monolithic system it is often easy pinpoint the misbehaving part. With our system functionality split into microservices, we suddenly have multiple possible suspects. We need to monitor behavior of multiple components each of producing log entries to separate locations. Inspecting each of these components separately and trying to make sense of it all becomes infeasible task in no time. To deal with this kind of situations we need to aggregate log entries into a single location where we can monitor the system as a whole.

4. KUBERNETES

Kubernetes is an open-source orchestration system for containerized applications. It handles deployment, scaling and management of applications by grouping them into logical units called pods. Kubernetes abstracts many of the previously described concepts and lets developer focus on the big picture. In this section the reasons for choosing Kuberenetes are explained, alternative solutions presented and basic architecture and building blocks of Kubernetes system explained.

4.1. Reasons for choosing Kubernetes

Container technology provides good means for setting up minimal running environment and isolating it from rest of the world. However, containers alone are not good as they need to be deployed, scaled and connected to external resources. All of this requires work and deviates developers from their initial task; working on the application.

Kubernetes builds upon Google's over decade long experience of running container management systems. It is an abstraction that let's user to focus on the big picture. Kubernetes offers horizontal scaling and automated deployment rollout and rollback. It has build in service discovery, load balancing and secret management. It provides support for stateful applications and persistent volumes. It can limit access using namespaces and resource usage using CPU and memory quotas.

4.2. Alternatives

In this section various alternative container orchestration solutions are presented and evaluated. Finally they are compared against Kubernetes.

4.2.1. Amazon EC2 Container Service

Amazon EC2 Container Service (ECS) is a highly scalable, high performance container management service for Docker containers[14]. ECS groups a number of EC2 instances into a single cluster and then allows container orchestration onto it using simple API calls. Containers are grouped using task definitions. These task definitions determine how many resources will be allocated, which commands are ran and how containers are linked to together[15]. Services are then used to keep desired amount of task definitions running[16]. ECS is only available inside the AWS ecosystem and is not a turnkey solution, but leverages integration to other AWS services like Elastic Load Balancing, EBS volumes and IAM roles.

4.2.2. Docker Swarm mode

Docker Swarm mode is a cluster management and orchestration tool build into Docker Engine[17]. It allows managing a cluster of Docker containers utilizing the same API

that Docker users are already familiar with. Cluster consists of nodes that can be either manager nodes that performs orchestration and cluster management or worker nodes that receive and execute tasks[18]. It shares a similar feature set with Kubernetes including declarative approach for defining the desired state, build-in service discovery and rolling updates, but unlike Kubernetes it also supports creation of overlay networks out of the box[19]. Docker claims that a single Swarm manager can handle 1000 nodes running 30 000 containers[20].

4.2.3. HashiCorp Nomad

Nomad is a tool for managing a cluster of machines and running applications on them[21]. It ships as single binary with build-in schedulers and resource managers that do not require external services for storing the state. It is distributed and highly available, and it automatically handles fail overs. It does not provide secret management or service discovery but instead embraces integration with Consul and Vault which are tools also build by HashiCorp. This introduces additional challenges in the form of configuring and monitoring these separate systems. However this is how Nomad was designed to be, instead of providing full orchestration system for containers it only focuses on cluster management and scheduling[22].

4.2.4. Mesosphere Data Center Operating System

Mesosphere's Data Center Operating System (DC/OS) is a distributed operating system build upon the Apache Mesos[23]. It pools and abstracts compute resources and allows them to be managed like a single computer. It targets many different types of workloads including, but not limited to, containerized applications. DC/OS comes with a container orchestration platform called Marathon pre-installed and configured. A variety of other different frameworks including Kubernetes is supported that can be used to replace Marathon or even both of them can be used together to manage different workloads[24]. DC/OS offers both community and (paid) enterprise editions of it's platforms. Community version is somewhat limited with missing features like multi-user identity, authentication and secrets management[25].

4.2.5. Resolution

A comparison of the above container orchestration systems can be seen in Table 1. After carefully evaluation all the alternatives, Kubernetes was seen to best serve the company's needs for multi-tenant cluster environment. Kubernetes provides a feature rich and very actively developed[26] framework that is backed by major companies like Google, Microsoft and Cisco[27]. Two alternatives came close, ECS and DC/OS. The biggest shortage of ECS is that it is very coupled to the AWS ecosystem and can not be used outside of it. DC/OS also offers most of the same features as Kubernetes but some of them require paid enterprise subscription like secret management. DC/OS is also more focused on big data related computing with its one click integrations for

tools like Cassandra, Kafka and Spark. Kubernetes was seen to provide just the right amount of features yet still retaining the possibility to configure it as seen fit.

Table 1. Comparison of container orchestration systems (x is a supported feature)

	Kubernetes	ECS	Docker swarm	Nomad	DC/OS
High-availability	x	x	x	x	x
Load balancing	x	x	x	-	x
Auto scaling	x	x	-	-	x
Self-healing	x	x	x	x	x
Rolling updates	x	x	x	x	x
Volume mounts	x	x	x	-	x
Service discovery	x	x	x	-	x
Secret management	x	x	x	-	x/-
Open source	x	-	x	x	x

4.3. Architecture

A running Kubernetes system has two key components; master node that is responsible for managing the cluster and one or multiple worker nodes that run the application containers. Both of these nodes run several essential components that are explained in more detail below [28][29]. A high level diagram of Kubernetes architecture can be seen in Figure 3.

4.3.1. Master node

Master node is in charge of orchestrating the worker nodes where the actual workload is run. It provides an entrypoint for cluster administrative tasks. *Master node* usually runs in a single machine but can be distributed across multiple machines.

API server

Kubernetes exposes a REST API to control the various cluster resources. *API server* processes these REST operations, validates them and executes the bound business logic.

etcd

etcd is a distributed, fault tolerant key-value store that can be distributed across multiple nodes. Kubernetes uses *etcd* to store shared configuration data which is representation of the current state of the cluster and can be used for service discovery.

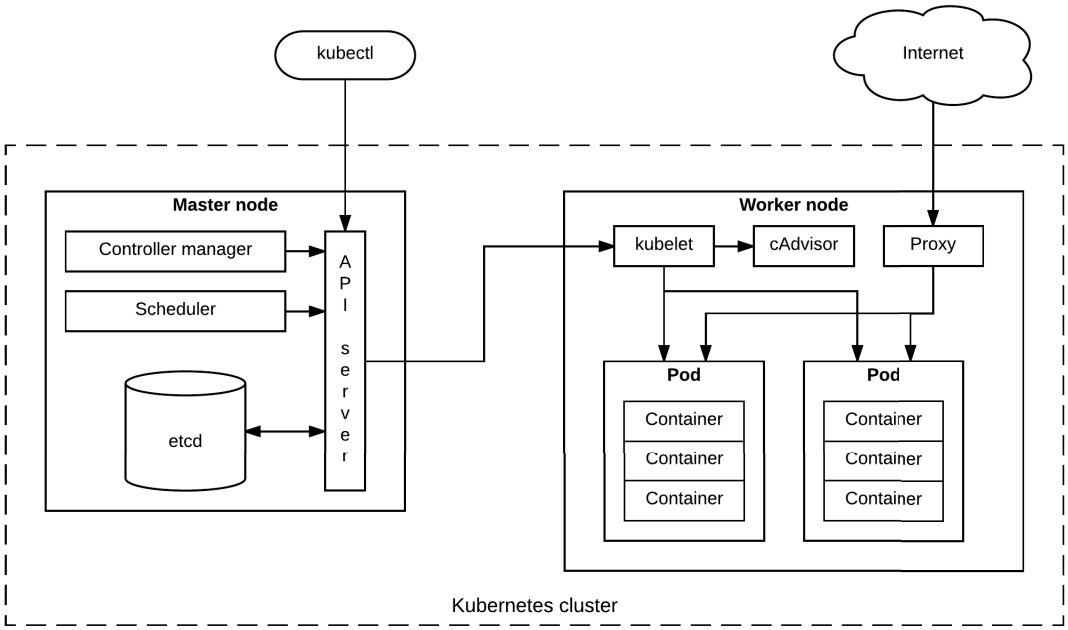

Figure 3. Kubernetes architecture

Controller Manager

Controller Manager uses the *API server* to watch the shared state of the cluster stored in *etcd*. When a change of state is detected the controller performs necessary actions to fulfill the desired state. For example, *Controller Manager* makes sure that desired number of *Pods* defined in *Replica Set* is running.

Scheduler

Scheduler handles workload assignment in the cluster. It monitors the cluster resource utilization and schedules *Pods* onto *Nodes* that have available computing resources.

4.3.2. Worker node

Worker nodes run *Pods* that perform the actual work. They also have the required services to communicate with the *Master node*.

Container runtime

Kubernetes support multiple container runtimes of which Docker is the most popular one. Container runtime takes care of downloading the images and running the containers.

kubelet

Kubelet communicates with *master node* to receive work and makes sure that the described containers in the work manifest are up and running. *Kubelet* also communicates with the *etcd* to read and write information about running services. There is also a built-in agent called *cAdvisor* which collects resource usage and performance data of running containers [30].

kube-proxy

Each Kubernetes *node* has it's own dedicated subnet which is not accessible externally by default. *Kube-proxy* is a network proxy with load-balancing capabilities that forwards network requests to the right containers and allows them to be exposed externally. Each node in the cluster is running a version of *kube-proxy*.

4.3.3. Kubernetes DNS

Kubernetes has a built-in *DNS* service which provides naming and discovery for the services running in the cluster. By default *Pod's* DNS search list includes *Pod's* own namespace and the cluster's default domain for easy discovery.

4.3.4. Kubernetes Dashboard

Kubernetes Dashboard is web-based user interface for managing and troubleshooting applications in the cluster. It provides an overview of the cluster and an easy to way to deploy and scale applications.

4.3.5. kubectl

Kubectl is a powerful command-line tool for interacting with the Kubernetes cluster via *API server*. It is the primary way to create and manage objects in the cluster.

4.4. Building blocks

In Kubernetes, resources are described in declarative manner; user declares the desired state and Kubernetes tries to reach that. Kubernetes system consists of multiple building blocks of which the most important ones are described next.

4.4.1. Cluster

Cluster is a set of virtual or physical machines running a Kubernetes master and one or more worker nodes.

4.4.2. Master

Master consists of multiple components that are used to administrate and control the cluster[31]. These components were presented in subsection 4.3.1. Master exposes Kubernetes API which is primary channel to control the cluster.

4.4.3. Node (minion)

A (worker) *Node* is a either virtual or physical machine that has the services necessary (Docker, kubelet and kube-proxy) to run *Pods*[32]. Worker nodes are managed by the *Master* components.

4.4.4. Pods

Pods are the smallest deployable unit[33]. A single *Pod* consists of one or more tighly coupled containers that are connected to the rest of the environment via an overlay network. Containers inside the *Pod* share same resources like data volumes. *Pods* are treated as ephemeral entities; in case of a failure Kubernetes terminates the *Pod* and re-schedules it to somewhere on the cluster. Scheduling *Pods* is like playing a game of Tetris, based on the required resources Kubernetes places *Pod* on to node it fits best.

4.4.5. Replica Sets

Replica Sets are used to run desired number of *Pods* at certain time. *Replica Set* performs health checks to *Pods* and restarts any malfunctioning ones. *Deployment* is a higher-level concept that manages *Replica Sets* and are recommended to be used instead of directly using *Replica Sets*[34].

4.4.6. Daemon Sets

Daemon Set works in a similar fashion to as *Replica Sets* does. Instead of ensuring a number of replicas are up and running, *Daemon Set* makes sure a set of *Pods* is running on every node within the cluster. Possible use cases include different sorts of agents and daemons[35].

4.4.7. Deployments

Deployment is high-level concept that manages *Replica Sets* and provides declarative updates to *Pods*. *Deployments* are typically used to perform rolling updates to application. *Deployment* creates a new *Replica Set* for a new version of application allowing seamless update process and a option to rollback to an earlier version in case the current version is not stable[36].

4.4.8. Services

Each *Pod* gets an unique IP address, but as they come and go those IP addresses cannot be really relied upon. *Service* provides a persistent endpoint for a logical set of *Pods* that can be used to access the service provided by those *Pods*[33].

4.4.9. Namespaces

Namespace groups and isolates similar resources under the same roof. They can be used to differentiate business clients or deployment environments, and they provide a mean to restrict access[37].

4.4.10. Labels

Labels are key-value pairs that are attached to Kubernetes objects, such as *Pods* and *Replica Sets*[38]. They are used to organize resources and to attach identifying or other meaningful information. Unlike *Namespaces* they do not provide isolation but are used as metadata[38].

4.4.11. Jobs

Jobs are used to run short-lived, one-off tasks that are terminated upon successful execution. They can be used to run things like database migrations or batch jobs which only need to run once[39].

4.4.12. ConfigMaps and Secrets

ConfigMaps are used to make container images as reusable as possible. This way the same image can be used in all development environments. *ConfigMaps* are used to provide information in the form of short string or a file. *Secrets* work in a similar manner but focusing on sensitive information like API keys and passwords[39].

4.4.13. Volumes, Persistent Volumes and Persistent Volume Claims

Files inside containers are ephemeral, meaning they are lost if container crashes or is restarted. To store files in persistent manner and to allow them to be shared between other containers, Kubernetes introduces abstraction called *Volume*. *Volume* allows usage of various different types of storage options[40]. Kubernetes also introduces abstraction called *Persistent Volume* which are storage plugins like *Volume*, but have a lifecycle independent of any individual *Pod* that uses *Persistent Volume*[41]. *Persistent Volume Claim*, for one, is a request to mount *Persistent Volume* into a *Pod*.

5. RELATED WORK

This chapter presents a literature review of related work and discusses their limitations on the matter. Chapter is divided into five sections. In the first section Kubernetes related research is discussed, second section discusses metrics monitoring, third section presents billing systems, section four discusses different pricing models and the last section concludes and presents the contribution of this thesis.

5.1. Kubernetes specific research

Kubernetes was only open sourced in 2014 thus very limited amount of Kubernetes specific research has been conducted. This research can be mainly categorizied into three sections: applications or other software solutions that are utilizating Kubernetes, performance analysis on Kubernetes and general discussion about container technology, container orchestrations systems and Kubernetes.

Bila et al.[42] presents functionality to Kubernetes that enables the system to react to threats in an automated manner according to user-defined policies. Biran et al.[43] presents cloud coordinator prototype that is built on Kubernetes. Pizzolli et al.[43] introduces Cloud4IoT, a platform offering automatic deployment and orchestration of IoT support applications based on Kubernetes. Trnkoczy et al.[44] presents a cross-layer management architecture leveraging Kubernetes. Bekas et al.[45] created a prototype of event-driven videoconferencing application on top of Docker containers and Kubernetes. All of these solutions highly utilize the container orchestration capabilities of Kubernetes. They present a solution to their specific problem, but none of them focus on resource usage analysis.

The performance analysis research done in the past has primary focused on comparing performance differences between different virtualization techniques. More recent studies have also evaluated performance of container based solutions. Medel et al.[46] proposed a Reference net-based model that could be used for capacity planning and resource management inside Kubernetes. Joy[47] focused on comparing performance differences between virtual machines and containers. The study used Kubernetes for cluster management. The containers are seen to perform faster than traditional virtualization solutions. Performance was not however the issue of this thesis.

The swift from traditional virtualization to container based solutions have also been discussed in various studies and articles. Pahl[48] discusses the requirements that arise from having to facilitate applications through distributed multicloud platforms. The article uses Kubernetes as an example of a cluster management platform. Heidari et al.[49] surveyed a couple of different container orchestration solutions including Kubernetes. Singh et al.[50] discussed past and present of virtualization and the current trend of using container based technologies. The studies show that the current trend is heavily leaned towards containerization and that they require orchestration systems for managing them. This supports the company's decision to embrace the container technology.

5.2. Metrics monitoring

Lin et al.[51] proposed a self-adaptive push model for delivering resource monitoring metrics. The model uses transportation window with adaptive window size to store and batch collected metrics before they are delivered to monitoring servers. This was found to improve the data coherency. The nature of the billing tool to be built does not require rapid acquisition of metrics, instead of they are aggregated and fetched over longer time period. If this was a subject to change, using a similar kind of model could be researched.

5.3. Billing systems

Various billing systems have also been built on top of different kinds of cloud computing platforms. Next, a couple of these are presented.

In publication authored by Elmroth et al.[52] an accounting and billing architecture to be used within RESERVOIR project was proposed. Resources and Services Virtualization without Barriers (RESERVOIR) is a research project funded by the European Union. It develops an infrastructure capable of delivering elastic capacity that can automatically be increased or decreased in order to cost-efficiently fulfill established Service Level Agreements.

In paper by Park et al.[53] a secure and non obstructive billing system called THEMIS is proposed. The system takes care of all of the three security and system issues that are often raised by cloud billing systems. These are integrity and non-repudiation capabilities of billing transactions, computational efficiency of a billing transaction and trusted SLA monitoring.

Both of the above solutions are highly technical and either very domain specific or complicated to implement. Zhu et al.[54] presented more pragmatic study by implementing monitoring and billing modules on top of their cloud system called Docklet. Shortly after building the system the authors encountered problems with having a shortage of free computing resources. This was caused by users not releasing their resources after they had finished their tasks. To improve the utilization and fix this problem they developed custom monitoring and billing modules. The cluster in question was mainly used for short lived big data related batch jobs. Most popular cloud service providers like AWS, Google and Microsoft mainly charge users by service time despite how many resources they have used. Zhu et al. found that this traditional billing model does not take the rapid variation of short lived jobs into consideration and presented a billing mode that would charge users by amount resources they use rather than by how long they use them. Each hour the service would figure out how much user should pay for the used CPU time, disk capacity and memory usage. Experiments and actual results show that their billing model is useful for providing them better service and improving the utilization of limited physical resources. This is probably the most relevant piece of research for the sake of the thesis. The proposed model is still however very dependent on the use case presented in the paper.

5.4. Pricing

A number of studies also focused on how spot prices affect the markets and how they can be exploited. Exo-Sphere[55] is platform for portfolio-driven resource management on transient servers. It supports a large class of data-parallel applications such as Spark and Hadoop. Exo-Sphere automatically selects preferable servers from the available set of spot server using the MPT formulation. SpotOn[56] is a batch scheduler that acquires its resource from EC2's spot market. SpotCheck[57] allows applications to create as many virtualized instances as they wish, while the derivative cloud decides how to acquire cloud resources. SpotCheck migrates away from revoked spot servers, but it must maintain an additional backup server somewhere. These studies present an interesting insight into spot pricing, but as the company's cluster is deployed on to fixed price instances these methods can not be utilized.

5.5. Contribution

The Kubernetes related research is still very limited and mainly focuses on presenting software solutions that have been built on top of it. Also the previously implemented billing systems are all very domain specific and cumbersome to put to use. The main purpose of this thesis is to present a more universal solution for billing with as few dependencies as possible.

6. THE TOOL

The company needed a billing tool for a Kubernetes cluster they were building. Based on the previous positive experience the company had with AWS it was decided that this cluster would be deployed there as well. Even though Kubernetes is platform agnostic, possible limitations or peculiarities introduced by AWS services had to be taken into consideration. The cluster would then host multiple customers and each of them would billed individually based on their resource consumption. The tool would expose a web frontend that could be used by a modern web browser on desktop client. A tool with project name *Kubor - Kubernetes Overseer*, later on known as the tool, was created to fulfill this purpose.

6.1. Requirements

A typical customer projects has three different environments: development, staging and production. Normally development environment is setup under the company's account whereas staging and production environments are often under customer's account. After these have been setup AWS will then automatically handle the billing of account owner each month. In the future with the new cluster in place, the billing would be instead under the company's responsibility. This meant that the company had to come up with some kind of billing procedure that could be applied to each customer individually. To achieve this, the company would need to identify how many resources each customer were using and then bill them accordingly. There are multiple different resources that could monitored but it was agreed the tool would focus on four core metrics: CPU, memory, storage and network.

6.2. Process

Once the requirements had been laid out it was time to think about the process that could achieve all of this. The company would be billed each month for the resources they have consumed so it was only natural that the customers of the company would also follow this same billing model. Just as natural was the decision that each customer would be treated equally and that they would be billed only for the resources they consume. To identify which resources each customer were consuming, Kubernetes feature called namespaces was decided to be used. Namespaces provides a layer of isolation and a way to divide cluster resources between multiples users, customers in this case.

Four core metrics were decided to be monitored: CPU, memory, storage and network usage. As some of these metrics will behave differently than the others, following was decided. The amount of processing computing power and allocated memory depends on the instance type. Amazon EC2 instances are billed by hourly rate so customer usage on these resources would be also examined on hourly level. Storage and network on the other hand are billed by used gigabytes and that is how the tool was decided to handle it as well.

To price the customer, it was agreed upon that the algorithm would perform calculations on each resource using the following model: (customer resource usage / total resources available) * price. To achieve this, three main action steps were recognized: collecting metrics, calculating resource usage per customer and pricing customers accordingly. Next, each step is discussed in more detail.

6.2.1. Collecting metrics

First problem of the puzzle was how and where the required metrics would be collected from. Resource monitoring is nothing new and there are multiple different tools for doing that but the short lived nature of application containers and the fact that they are managed by Kubernetes introduced some new challenges. In Kubernetes application containers are treated as ephemeral, meaning they can come and go as long as desired amount of replicas are online at given time. Also the number of nodes (either physical or virtual hosts) might vary over time meaning the whole resource collecting process needed to be very dynamic. This meant that the monitoring tool would need to perform some kind of service discovery. Luckily ready made monitoring tools also existed for Kubernetes domain, one of them being Heapster.

Heapster is a monitoring solutions that aggregates usage information from all parts of Kubernetes. It discovers all nodes in the cluster and then queries nodes' Kubelets for the usage information. The Kubelet itself fetches data from cAdvisor which is a resource usage and performance analysis agent. cAdvisor for one's part discovers all the containers in the machine and then collects core resource metrics like CPU and memory usage.

Even though Heapster exports various metrics to its backends, more thorough inspection revealed that some of the metrics required for the calculations were missing. For example, Heapster did not provide time series data for total available CPU and memory. These values could be queried at any given time but there was no historical data available. Resources on which cluster is running on, like amount of instances, can and will be scaled up or down during the month depending on the workload, meaning that the amount of available CPU and memory is not static. This information is required for the calculations the tool performs so another monitoring solution was needed. After some research a tool named Prometheus was discovered.

Prometheus is a monitoring and alerting system with a built-in time series database and it's own query language called PromQL[58]. Prometheus uses pull based model meaning it scrapes metrics from targets at a given interval. Targets can be either configured or discovered via service discovery.

Core component of Prometheus is the server component which handles scraping data and storing it to its' own time series database. Other main components include push gateway to which short lived jobs can push metrics, various exporters that help exporting metrics from existing third party systems and alert manager component for triggering and sending out alerts. Prometheus also provides Web UI and API endpoint for querying the collected metrics. A high level architecture of Prometheus and its components can be seen in Figure 4. For the purposes of this thesis we are focusing on the server component that handles scraping the metrics.

Prometheus collects metrics from either HTTP endpoints exposed by the monitored targets or from the push gateway component. Targets can implement metrics endpoints on their own allowing all kinds of custom metrics to be collected. Prometheus stores scraped data as a time series. A time series is a stream of samples belonging to the same metric and the same set of labels. Metric name specifies the name of measured metric and labels enable multi-dimensional data model. Samples form the actual data and consists of a float64 value and a millisecond precision timestamp. For example, if we were sampling total number of HTTP request we could label our data with method type and target URL. Using this example we could end up with similar time series as seen on Table 2.

Table 2. Example of Prometheus time series data

Metrics	Timestamp	Value
api_http_requests_total{method="GET", target="/posts"}	1508773222556	1467
api_http_requests_total{method="GET", target="/comments"}	1508773222556	3870
api_http_requests_total{method="GET", target="/news"}	1508773222556	564

Scrape interval is configurable and was set to 15 seconds for purposes of this thesis. Unfortunately Prometheus was not designed for long term monitoring. Due to vast amount of data it is scraping the data retention was set to two weeks. For the purposes of the tool resource usage data for at least the last calendar month was needed. Due to this a workaround had to be implemented. A cron job was created to fetch and store metrics collected by Prometheus into external database. Inner workings of this job is explained in subsection 6.5.4.

Prometheus collects metrics from Kubernetes cluster using two exporters node exporter and kube-state-metrics. Node exporter collects hardware and OS metrics exposed by *NIX kernels like CPU and memory statistics that we are interested in[59]. Kube-state-metrics listens to the Kubernetes API server and generates metrics about state of the objects such as deployments, nodes ands pods[60]. For the purposes of this thesis we are focusing on the former.

Data for four core metrics (CPU, memory, storage and network) were queried from Prometheus. As previously mentioned in this thesis, it was agreed that each customer would get a namespace of their own. It was also agreed that resource usage would be examined on hourly level so samples were aggregated by hour and then averaged. Data collecting process was started by first querying Prometheus for all the namespaces in the Kubernetes cluster. After that the core metrics usage would be queried for each namespace individually. Finally the total available resources would be queried for each core metric. In the end we would end up having CPU, memory, storage and network usage data for each namespace and the total available resources for each metric.

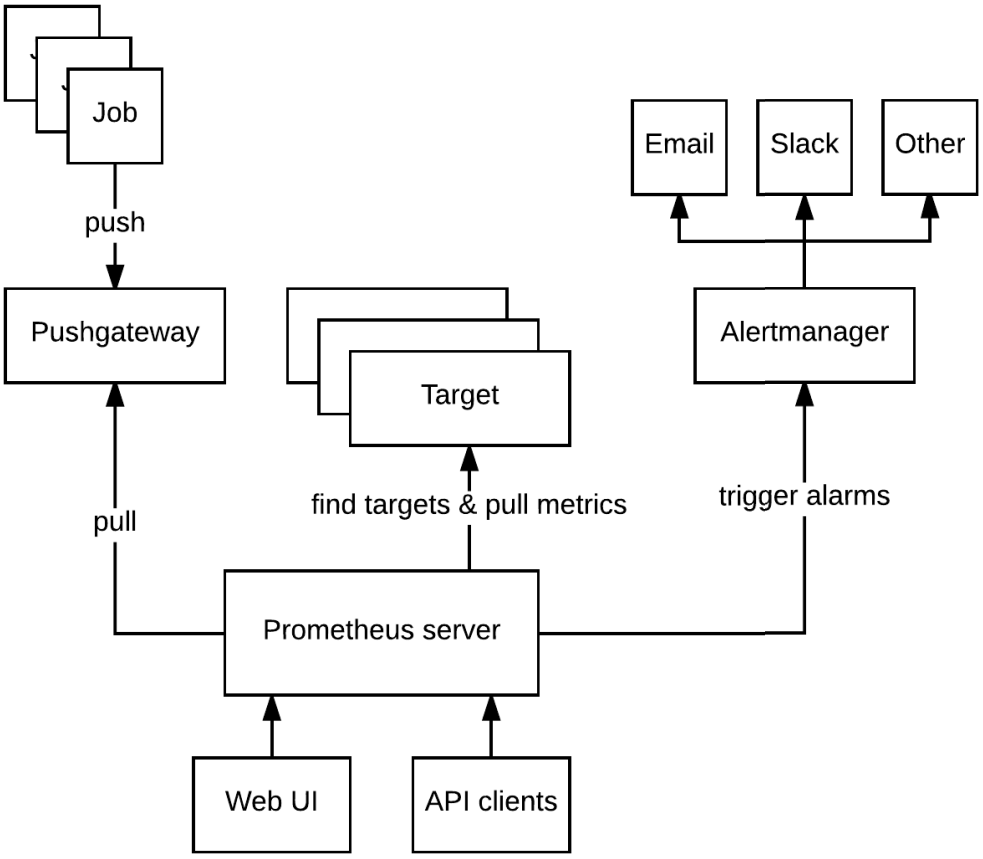

Figure 4. Prometheus architecture

6.2.2. Calculating resource usage

Once the appropriate metrics had been collected it was time to calculate customer specific resource usage. This would follow the previously introduced model: customer resource usage / total resources available. The core idea was to calculate percentage each customer were using certain resource from the total available resources by hourly basis. This meant that each customer would end up with 24 calculations for each record for one day. The value from this calculation could vary between zero and one, zero being no usage at all and one meaning all the resources were being used.

There are some differences between the collected metrics however. The amount of CPU and memory are tied to the type of the instance and the maximum values are capped. For storage and network there are no maximum values however. In this case the value of total available resources are actually the total resources consumed by the whole cluster but the percentage would still be calculated same. The only exception was network usage as Prometheus would only return cumulative usage values so difference to previous measurement had to be calculated.

6.2.3. Pricing usage

Pricing resource usage also introduced couple of challenges. One cannot actually buy certain amount of CPU or memory separately but instead one needs to purchase a whole instance (that can be either physical or virtual host) which comes with certain amount of allocated CPU power and memory. As you pay for the whole instance there is no clear way to dictate how much CPU is worth compared to RAM or vice versa. For the sake of simplicity it was decided that both CPU and RAM were equally valuable.

For storage on the other hand, one has multiple options. AWS offers for example Elastic Block Storage (EBS) which is persistent storage that can be mounted for single EC2 instance or database unit, Elastic File System (EFS) that allows same filesystem to be mounted on multiples EC2 instances and Simple Storage Service (S3) which is scalable storage mainly for archives and backups. What is common for all these three different types of storage is that you pay only for the storage you use (by gigabytes).

From the chosen core metrics network has probably the most complicated pricing scheme. Various things factor to total price like CDN cache misses, geographical nature of the traffic, used services and so on. For the sake of this thesis it was agreed to simplify calculations and treat all the traffic the same but to differentiate incoming and outgoing traffic from one another. In addition, AWS does not charge for incoming network traffic so only outgoing traffic is included in the calculations.

6.3. Dependencies

The tool is highly dependent on the resource metrics which are currently provided by Prometheus. The system was however designed so that the source of these metrics can be easily changed in the future. Kubernetes is still very young and under constant development but one potential alternative source of metrics could be the proposed Kubernetes Infrastore[61].

6.4. Design

Once the core processes of the tool had been outlined the next step was to start working on the user interface. The tool would be running inside the Kubernetes cluster allowing it to access the source of metrics. The user interface of the tool would be a web application meant to be used by desktop browsers. Theoretically the tool could also be accessed by mobile clients but this use case was not considered to be important for the purposes of this thesis. The design of the tool got much of its inspiration from Kubernetes Dashboard[62] that is a general purpose dashboard for managing Kubernetes clusters.

During the first design phase it was planned that the tool would consists of three main views: overview, reports and forecasts. Overview was planned to give quick a glance of the overall status of the cluster resource usage and help the end user to spot bottlenecks. Forecasts would present estimations of future resource usage for each customer based on historical data. It would also include option to set price limits and

trigger alarms if budget overruns were detected. Finally the reports view would contain the main functionality of the tool. Sketch of this view can be seen in Figure 5.

The reports views would consists of two main components. The configurator panel which would be used to set desired filters and the costs summary panel that would then visualize the resource usage and their costs. The user would start the flow by selecting namespace (customer) of which the usage data would displayed. Next the user could choose which resources to take into account when calculations are performed. The idea behind this was that price calculation might be challenging for some resources like network usage due to its vast amount of different pricing factors so these could be optionally omitted from the calculations. Then the user would choose desired time interval from which the calculations would be performed by selecting start and end dates. Last option the user would have is to choose aggregation period for the metrics which could be either hourly and daily. By pressing *Calculate* button the tool would fetch the appropriate metrics, perform calculations and then display cost summary underneath the configurator panel. Cost summary would present the data in table format, each resource being on its own row followed by usage metric that could be either hours for CPU and memory or bytes for network and storage. Followed by these per hour or per day usage depending on which option was chosen and the total costs for the resource. On the bottom of the table one would find summary with the total amounts.

The first version of the design was showed to and evaluated by software architects at the company. The overall feedback received was positive. Couple of improvements and changes were proposed that were mainly related how the costs are filtered and displayed with few feature requests. The first feature request was possibility to add optional fixed costs derived from for example administrative tasks. The date range selector was found flexible but since customers would billed by calendar month a more simpler month selector would be enough. Since we are paying for the resource by the hour, choosing metrics to be aggregated by day would not make sense and most likely only skew the results. Last proposal was to add an ability to view monthly cost summary of all namespaces at once.

Previous feedback was kept in mind when second of version the design was being worked on. Also expert evaluation was done to improve the overall user experience. The revamped user interface with improvements can be seen in Figure 6.

The main navigation was moved to up top from taking space on the left. *Reports* view was renamed to *Cost explorer* to signal the main use case better. For the first version of the tool only this view would be implemented. A hero banner was added below the navigation for visual purposes and to give user a brief description what the view in question was all about. Configurator was then moved the left hand side where it would be accessible at all times and the user would not need to scroll back to the top to just change some filter options. Configurator was also simplified and unnecessary features were eliminated. On the top of the configurator there is a simple month selector which defaults to current month and allows the user to go back six months in time. Resource selection was kept mostly the same but it would now additionally display the costs of the resource among with total costs on the bottom. Cluster costs were lacking from the initial design but where now brought to the second design. Due to time constraints possibility to modify these values using the interface were omitted. The option to choose aggregation period for the metrics was also omitted as discussed on the previous sections. The user flow changed so that it would not be necessary to

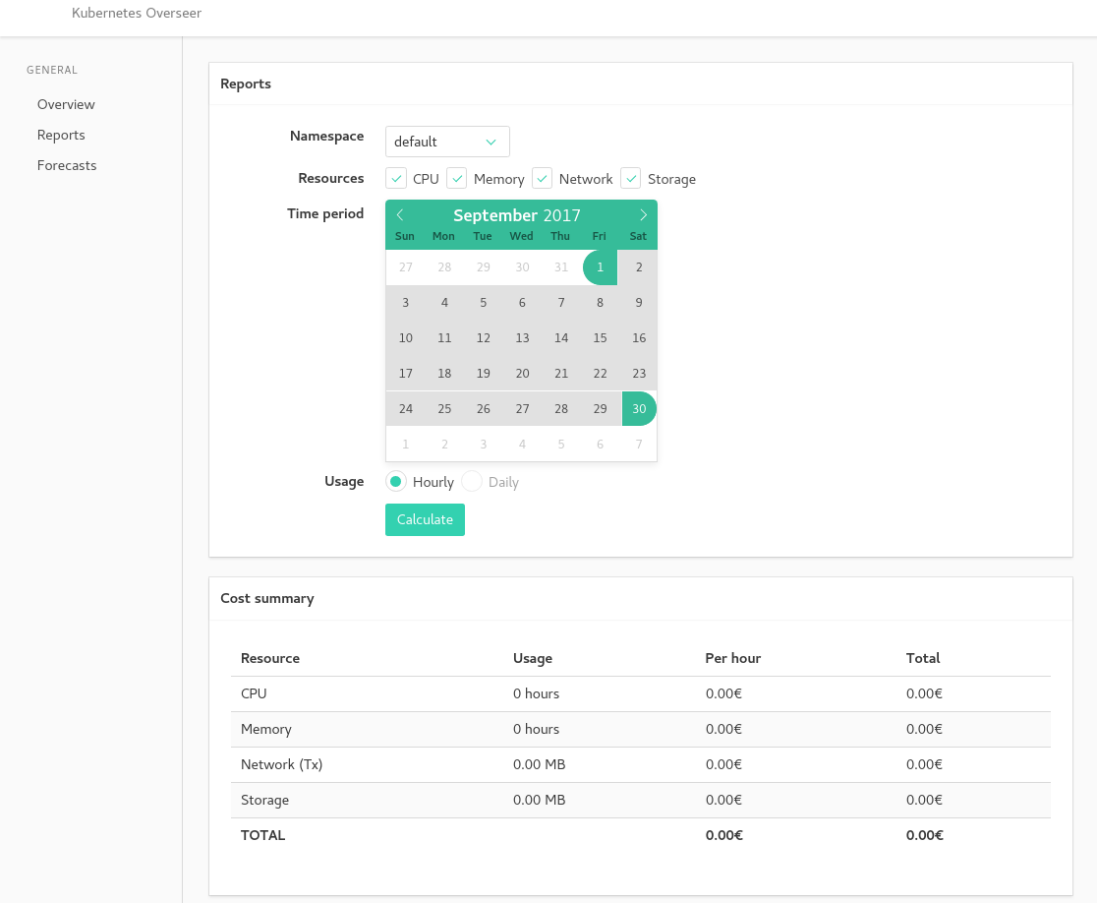

Figure 5. Design version 1

confirm the configuration options by pressing calculate button but instead the UI would now automatically refresh according to the selected options. The cost summary would now display all the namespaces at once cutting down unnecessary user actions. This allows important information to be displayed immediately. This second version of the design was the base for the tool to implemented. In the next section architecture of the tool will be evaluated.

6.5. Architecture

The finished tool will be running in Kubernetes cluster inside a single Docker application container. The tool itself consists of three main components: frontend that is the user interface, backend were the business logic resides and database were the metrics and cluster data is stored. Next each of these components are described in more detail.

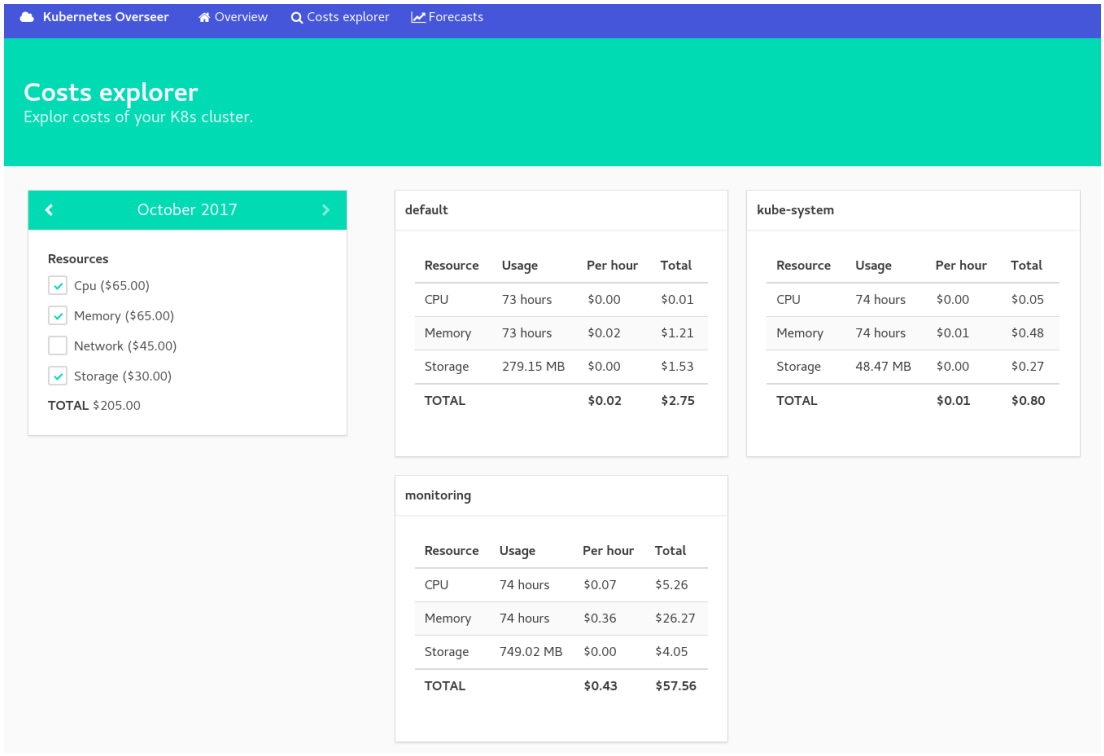

Figure 6. Design version 2

6.5.1. Frontend

Frontend provides the visual user interface and is the primary way for end user to interact with the tool. Frontend communicates with the backend using REST API calls. These calls are used to fetch resource usage metrics, cluster costs and namespace information. Design and the reasons behind the UX decisions were discussed in the previous section.

The tool is written in modern JavaScript. It utilizes React[63] for creating component based user interface and Redux[64] for predictable state management. For visual elements, a modern CSS framework with pre-build UI components called Bulma[65] is used. The tool is then compiled to browser compatible JavaScript by Babel[66] and finally bundled into static assets using webpack[67]. These static assets are served by the backend which is described in the next subsection.

6.5.2. Backend

Backend handles the main business logic of the tool and communication to external database where resource usage metrics and cluster cost data is stored. It is also written in JavaScript allowing re-use of some common logic blocks with the frontend. Backend has two primary use cases, it serves the static frontend assets and exposes REST API for accessing data stored in the database.

The core of the backend is Node.js[68], de facto server side runtime environment for JavaScript. Running of top of that is Express[69], a popular web framework that is used to expose REST API for frontend to interact with. API was versioned to make it as future-proof as possible. Routes exposed by Express can be seen in Table 3. Backend interacts with external SQLite database using Knex.js[70] which is so-called query builder supporting multiple different databases. Knex.js also takes care of initializing the database, handling the migrations and providing seed data. More information about the database and its internal structure in subsection 6.5.3.

Table 3. REST API endpoints

Method	Route	Description
GET	/api/v1/proxy/prometheus<query>	Proxy Prometheus queries
GET	/api/v1/metrics?start=<unix timestamp> &end=<unix timestamp> &type=<cpu/memory/storage/network> &namespace=<namespace name>	Get metrics for selected resource from selected namespace within chosen time period
GET	/api/v1/costs?month=<1-12>	Get cluster costs for selected month
POST	/api/v1/costs	Update or add cluster costs

6.5.3. Database

External database was used to store resource usage metrics and cluster cost data. SQLite[71] was chosen for this task because of its easy of setup, the database is only a single file. The database schema was also very simply with few relations so no advanced features were required. However it was structured so that if more advanced features are needed in the future data can be easily migrated into more sophisticated database solution like PostgreSQL[72].

Database is only accessed by the backend service. A high level diagram of database entity relationships can be seen in Figure 7.

6.5.4. Cronjob

Due to the fact that Prometheus was not designed for long term monitoring and that data retention rate was set to two weeks, a solution for long-term data storage had to thought of. This is why metrics data used by the the tool is stored in the external database. A simple cronjob was created to query Prometheus every hour for namespace and metrics information and then store that data to the database for the tool to use. This job is triggered by the backend service but in the future this logic should be decoupled.

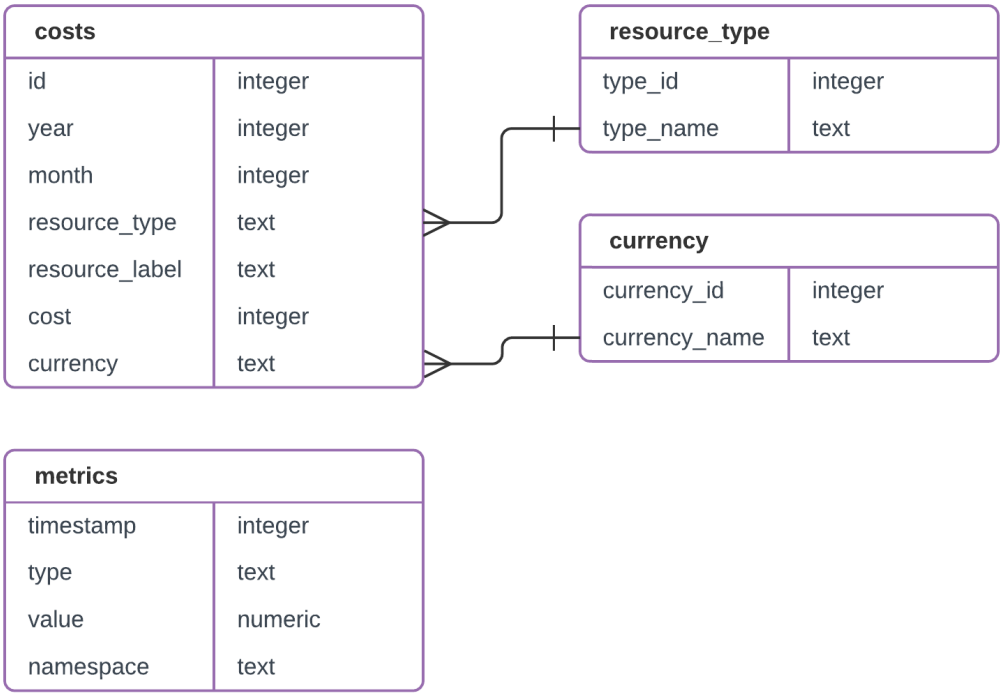

Figure 7. Database entity relationship

6.6. Deployment

Even thought the tool could be run on local environment (if traffic to Prometheus was proxied) it is meant to be run in the Kubernetes cluster among other cluster tools. The tool is deployed to Kubernetes using a Kubernetes configuration file. Kubernetes configuration files are written either in YAML or JSON format and they contain information how resources should be configured. Applications deployed to Kubernetes are application containers, so before that the tool needs to be containerized. This happens using a custom build script written for the tool. This script works in two phases. During the first phase it runs the frontend build pipeline creating static assets for the backend to be served. On the second phase it builds the backend and bundles it into a Docker container with the static frontend asssets. Before this container can be used in the Kubernetes cluster it needs to be pushed to some container registry. Container registry is a centralized repository for application containers. Registry can be either private or public, and it can be running either inside or outside of the Kubernetes cluster. Once the container image has been pushed to the registry the Kubernetes configuration file needs to be modified to use the appropriate container image. On the configuration file also the Prometheus URL has to be set to match the cluster configuration. The URL depends on how the Prometheus has been deployed to the Kubernetes cluster.

The tool utilizes a database and because containers in Kubernetes are treated as epheremal, a persistent volume has to be configured, otherwise the collected data would be lost if the container crashed or was shut down. Configuration for this persis-

tent volume is also dependent on the environment on which the cluster was deployed to. In the company's case, the cluster is deployed to AWS thus Amazon EFS was chosen to be used. Amazon EFS is scalable file storage that can be attached to multiple instances and was good fit for this purpose. Finally once the configuration file has been modified to match the cluster configuration the actual deployment can be performed using a single command: *kubectl apply -f <filename.yaml>*. The configuration file takes care of setting up the persistent volume and running the application. The configuration file created for this tool is just one way to deploy the tool, a custom configuration file can be used as well. The only requirement is to setup a persistent volume that the database utilizes.

6.7. Usage

Once the tool has been deployed to the Kubernetes cluster it can be accessed in multiple ways. One solution is to expose the tool to the Internet which is not recommended as the tool might expose sensitive information like customer names. More preferable solution to access the tool is to use the port forwarding functionality build-in to kubectl. If this method can not be used for some reason, a simple SSH tunnel to one of the worker nodes in the Kubernetes cluster can be used to proxy traffic to the tool itself. This works because the frontend will not directly access Prometheus or any other service in the cluster, but all the communication happens through its own backend.

The tool runs automatically in the background and does not require any maintenance work. Once access to the tool has been achieved, one can start to explore the cost summaries. The tool will automatically fetch the resource usage for the current month and perform the calculations. Filters and other options are explained in the section 6.4.

7. ASSESSMENT

Assessment of the tool proved out to be one of the major challenges. At the time of writing this thesis the company's cluster was still undergoing some configuration and fine tuning. Because of this projects would not have been able to take full advantage of the benefits provided by Kubernetes workflow. This was one of the main reasons why new projects had not been deployed to the new cluster environment. The tool bases heavily on resource usage metrics which need to be accumulated over time. For the metrics to be as accurate as possible this process can not be really accelerated. Due to this it was decided that the assessment would focus on evaluating how well the predefined requirements were met. The tool implemented was considered as a proof of concept implementation that would be iterated and improved in the near future.

The tool was to collect metrics and calculate average resource usage of each customer and to provide an easy-to-use web interface to display that information. Two design iterations of this web interface were made which are more thoroughly discussed in the section 6.4. To conclude this process, the design of the first version was based on the initial requirements set for the tool. This design was then showed and evaluated by software architects of the company. This feedback was then used for the second iteration of the design. Key changes included simpler time period selection, including fixed costs and displaying each customer information at once. This second design was then used to implement the first version of the tool. This is the version that was also deployed to the cluster. Feedback for this tool was then asked from same software architects. The changes made were found to be adequate and the tool to fill it's main purpose. Especially the fixed costs were found to be a good addition as they could help to balance the billing in the early phases when the costs model is still being iterated. Customer or even environment specific costs were proposed to be added in the future. These costs could include things like external databases or data monitoring tools that are setup specifically for certain use cases. This was agreed to be defined in more details in the near future.

It was noted that the introduction of the cluster is still in the early stages due to the ongoing adjustments. Customers projects were yet to be deployed due to which only a very limited amount of metrics were collected. In order to analyze the cost model better a lot more data would be needed especially to make any kind of final conclusions. Balancing the cost model for CPU and memory was found to be a one of major challenges for the future. CPU and memory cannot be bought separately, instead they are dependent on the chosen instance type. Currently these are threated equally, both CPU and memory are considered to be half of the total instance costs. This does not reflect the reality very well because memory is often the limiting factor. This is mainly due to the nature of web applications, they are rarely I/O heavy but instead use a lot of memory for things like caching.

The cluster would be hosting many paying customers in the future drastically increasing the cluster's utilization rate. This will benefit both the company and the customers. As the cluster resources utilization can be better optimized, the customers will have pay less for resource they use. From the company's perspective the positive income from the customers will likely cover the development costs. As a result of this the cost model can be refined and optimized even further. This is one of the main action points for the future.

In conclusion, the work done was seen purposeful. It proved that a billing tool for this kind of cluster environment can made. This is enough for the company to able to proceed with preparing the cluster for production stage. The tool was seen to provide several benefits for company. It can be used as an invoicing criterion and as an internal tool for understanding cost structures. It can be used to justify decisions and to help making estimations. Ultimately, decisions on the payment model are made by the company's management but the implemented tool presents a lot of potential.

8. LIMITATIONS

This thesis presents a software-based solution for analyzing and pricing resource usage on a modern multi-tenant cloud cluster. The tool was field tested for short period of time (one week) during which it performed as expected. Nature of the tool would require resource usage data to be collected for at least one calendar month which was not achieved due to time constraints and because cluster configuration phase the was still undergoing. This is the first and the most major limitation of the tool.

Second limitation of the tool is its eminent dependency to resource usage metrics which are currently provided by Prometheus. Prometheus is however very stable, actively developed tool backed up major companies which makes it seem very future proof choice.

Third limitation is the cluster pricing model. The model was designed to be as flexible as possible but diverse pricing schemes introduced by cloud providers make it hard to come up with one fit solution. A good example of these complex pricing schemes are pricing of network data. Some providers include network traffic as a part of the instance price while others have very complicated pricing scheme based on the nature of the traffic.

Despite the limitations mentioned above the tool provides a solid ground for future development. No major issue were found and the future work mainly consists of tweaking and adding new features.

9. FUTURE WORK

After the initial pilot period was held the company decided that the next step would be to improve storing of the collected resource metrics. It was seen that a best place for these metrics would be an external database outside of the cluster. This way the metrics data would stay intact even in a case of cluster failure. The metrics would be then used in improving the billing tool's cost model. Besides that they could provide other meaningful use like a source of visualization for various graphs.

The tool was designed in such a way that new features can be easily implemented on top of it. Development will be continued and various features additions have been already proposed. Next these future ideas are discussed in more details.

Cluster overview and forecast were left out from the first version as they were not seen relevant for the purposes of cost calculation. Cluster overview would display a live summary of the cluster resource usage including a total of free and used resources, and resource usage by pod and namespace. This information could be used to spot bottlenecks and possible mis-configured deployments. Forecast view would be used to display estimations based on historical data. One could also set warnings and trigger alarms if budget overruns or spikes in the resource usage were be detected.

The cluster costs model will continue to be iterated in the future. The current model allows somewhat flexible usage for different price data but it is missing a user interface to manage this information. A new view could be added to be dashboard that would be used for configuring cluster resources and their costs. This could be taken even more further and some kind integration could made to AWS's billing API.

The costs explorer could be improved further, for example it does not provide any kind of exportable reports. Even automated report generation could be considered.

Currently all the customers are treated equally. In real world, bigger customers might be rewarded with lower prices so custom price modifiers could be introduced. Also the cluster will not ever be running at maximum capacity because of automatic resource scaling. Due to this a price modifier based on desired cluster usage rate could also be introduced.

Packaging the tool to make it more accessible should be considered. Currently it is not publicly available and a private container registry is required. Once the tool has been battle tested it could be released publicly to one of the popular container registries. Kubernetes has official package manager named Helm which could be one of the distribution channels.

10. REFERENCES

[1] Codemate ltd (accessed 31.10.2017). Codemate Homepage. URL: https://www.codemate.com/.

[2] Gartner identifies 10 key actions to reduce it infrastructure and operations costs by as much as 25 percent (accessed 9.4.2017). Gartner. URL: http://www.gartner.com/newsroom/id/1807615.

[3] Barr J., Cloud computing, server utilization, & the environment (accessed 9.4.2017). AWS Blog. URL: https://aws.amazon.com/blogs/aws/cloud-computing-server-utilization-the-environment.

[4] Bias R. (2012), Architectures for open and scalable clouds. Cloud Connect Conference.

[5] Innoitas (2016) The project and portfolio management landscape. Tech. rep.

[6] Elliott G. (2004) Global Business Information Technology: an integrated systems approach. Pearson Education.

[7] Conway M. (1968) How do committees invent. Datamation 14, pp. 28–31.

[8] MacCormack A., Baldwin C. & Rusnak J. (2012) Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis. Research Policy 41, pp. 1309–1324.

[9] Martin R.C., The single responsibility principle (accessed 14.4.2017). 97 Things Every Programmer Should Know. URL: http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle.

[10] Doug McIlroy E. N. Pinson B.A.T. (1978) Unix Time-Sharing System: Foreword. The Bell System Technical Journal 57, pp. 1092–1093.

[11] Newman S. (2015) Building Microservices. O'Reilly, first ed.

[12] Felter W., Ferreira A., Rajamony R. & Rubio J. (2015) An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172.

[13] Loukides M. (2012) What is DevOps? O'Reilly.

[14] Amazon ec2 container service (accessed 8.5.2017). URL: https://aws.amazon.com/ecs/.

[15] Amazon ecs task definitions (accessed 26.10.2017). AWS Documentation. URL: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html.

[16] Amazon ecs services (accessed 26.10.2017). AWS Documentation. URL: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs_services.html.

[17] Docker swarm mode (accessed 8.5.2017). URL: https://docs.docker.com/engine/swarm/.

[18] How nodes work (accessed 26.10.2017). Docker Docs. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes.

[19] Docker swarm mode overlay network security model (accessed 26.10.2017). Docker Docs. URL: https://docs.docker.com/engine/userguide/networking/overlay-security-model.

[20] Scale testing docker swarm to 30,000 containers (accessed 26.10.2017). Docker Blog. URL: https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers.

[21] Introduction to nomad (accessed 26.10.2017). Nomad Homepage. URL: https://www.nomadproject.io/intro/index.html.

[22] Nomad vs. kubernetes (accessed 26.10.2017). Nomad Homepage. URL: https://www.nomadproject.io/intro/vs/kubernetes.html.

[23] Dc/os (accessed 8.5.2017). URL: https://dcos.io/.

[24] Announcing: Kubernetes on dc/os (accessed 26.10.2017). Mesosphere Blog. URL: https://mesosphere.com/blog/kubernetes-dcos.

[25] Mesosphere dc/os subscriptions (accessed 26.10.2017). Mesospherere DC/OS Homepage. URL: https://mesosphere.com/pricing.

[26] Contributors to kubernetes (accessed 26.10.2017). Github. URL: https://github.com/kubernetes/kubernetes/graphs/contributors.

[27] Container party: Vmware, microsoft, cisco and red hat all get in on app hoopla (accessed 26.10.2017). Network World. URL: https://www.networkworld.com/article/2601925/cloud-computing/container-party-vmware-microsoft-cisco-and-red-hat-all-get-in-on-app-hoopla.html.

[28] Kubernetes design and architecture (accessed 20.5.2017). URL: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture.md.

[29] Introduction to kubernetes architecture (accessed 20.5.2017). X-Team Blog. URL: https://x-team.com/blog/introduction-kubernetes-architecture/.

[30] cadvisor (accessed 21.5.2017). Github. URL: https://github.com/google/cadvisor.

[31] Master components (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/overview/components/#master-components.

[32] Nodes (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/architecture/nodes/.

[33] Pod (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod/.

[34] Replica set (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[35] Daemon set (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

[36] Deployment (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[37] Kubernetes namespaces (accessed 23.10.2017). Kubernetes Documentation. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[38] Labels and selectors (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels.

[39] Hightower K. (2017) Kubernetes : up and running: dive into the future of infrastructure. O'Reilly Media, Sebastopol, CA.

[40] Volumes (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/storage/volumes/.

[41] Persistent volumes (accessed 27.10.2017). Kubernetes Docs. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[42] Bila N., Dettori P., Kanso A., Watanabe Y. & Youssef A. (2017) Leveraging the serverless architecture for securing linux containers. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), pp. 401–404.

[43] Pizzolli D., Cossu G., Santoro D., Capra L., Dupont C., Charalampos D., Pellegrini F.D., Antonelli F. & Cretti S. (2016) Cloud4iot: A heterogeneous, distributed and autonomic cloud platform for the iot. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 476–479.

[44] Trnkoczy J., Pašcinski U., Gec S. & Stankovski V. (2017) Switch-ing from multi-tenant to event-driven videoconferencing services. In: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 219–226.

[45] Bekas E. & Magoutis K. (2017) Cross-layer management of a containerized nosql data store. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 1213–1221.

[46] Medel V., Rana O., Banares J.A. & Arronategui U. (2016) Modelling performance resource management in kubernetes. In: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), pp. 257–262.

[47] Joy A.M. (2015) Performance comparison between linux containers and virtual machines. In: 2015 International Conference on Advances in Computer Engineering and Applications, pp. 342–346.

[48] Pahl C. (2015) Containerization and the paas cloud. IEEE Cloud Computing 2, pp. 24–31.

[49] Heidari P., Lemieux Y. & Shami A. (2016) Qos assurance with light virtualization - a survey. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 558–563.

[50] Singh S. & Singh N. (2016) Containers docker: Emerging roles future of cloud technology. In: 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), pp. 804–807.

[51] Lin K., Tong W., Liu X. & Zhang L. (2013) A self-adaptive mechanism for resource monitoring in cloud computing. In: IET International Conference on Smart and Sustainable City 2013 (ICSSC 2013), pp. 243–247.

[52] Elmroth E., Marquez F.G., Henriksson D. & Ferrera D.P. (2009) Accounting and billing for federated cloud infrastructures. In: 2009 Eighth International Conference on Grid and Cooperative Computing, pp. 268–275.

[53] Park K.W., Han J., Chung J. & Park K.H. (2013) Themis: A mutually verifiable billing system for the cloud computing environment. IEEE Transactions on Services Computing 6, pp. 300–313.

[54] Zhu Y., Ma J., An B. & Cao D. (2017) Monitoring and billing of a lightweight cloud system based on linux container. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), pp. 325–329.

[55] Sharma P., Irwin D.E. & Shenoy P.J. (2017) Portfolio-driven resource management for transient cloud servers. CoRR abs/1704.08738. URL: http://arxiv.org/abs/1704.08738.

[56] Subramanya S., Guo T., Sharma P., Irwin D. & Shenoy P. (2015) Spoton: A batch computing service for the spot market. In: Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC '15, ACM, New York, NY, USA, pp. 329–341. URL: http://doi.acm.org/10.1145/2806777.2806851.

[57] Sharma P., Lee S., Guo T., Irwin D. & Shenoy P. (2015) Spotcheck: Designing a derivative iaas cloud on the spot market. In: Proceedings of the Tenth European Conference on Computer Systems, EuroSys '15, ACM, New York, NY, USA, pp. 16:1–16:15. URL: http://doi.acm.org/10.1145/2741948.2741953.

[58] Prometheus (accessed 21.10.2017). Prometheus Homepage. URL: https://prometheus.io/.

[59] Node exporter (accessed 23.10.2017). Github. URL: https://github.com/prometheus/node_exporter.

[60] kube-state-metrics (accessed 23.10.2017). Github. URL: https://github.com/kubernetes/kube-state-metrics.

[61] Kubernetes monitoring architecture proposal (accessed 23.10.2017). Github. URL: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/monitoring_architecture.md.

[62] Kubernetes dashboard (accessed 24.10.2017). Github. URL: https://github.com/kubernetes/dashboard.

[63] React (accessed 21.10.2017). React Homepage. URL: https://reactjs.org.

[64] Redux (accessed 21.10.2017). Redux Homepage. URL: http://redux.js.org.

[65] Bulma (accessed 24.10.2017). Bulma Homepage. URL: https://bulma.io.

[66] Babel (accessed 21.10.2017). Babel Homepage. URL: https://babeljs.io/.

[67] Webpack (accessed 21.10.2017). Babel Homepage. URL: https://webpack.github.io/.

[68] Node.js (accessed 21.10.2017). Node.js Homepage. URL: https://nodejs.org/en/.

[69] Express (accessed 21.10.2017). Express Homepage. URL: https://expressjs.com/.

[70] Knex.js (accessed 21.10.2017). Knex.js Homepage. URL: http://knexjs.org/.

[71] Sqlite (accessed 21.10.2017). SQLite Homepage. URL: https://www.sqlite.org/.

[72] Postgresql (accessed 21.10.2017). PostgreSQL Homepage. URL: https://www.postgresql.org/.