Data processing pipeline automation on cloud platform

Sampsa Hyvämäki

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 18.2.2019

Supervisor

Prof. Simo Särkkä

Advisor

M.Sc. (Tech.) Mika Vuorio

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Sampsa Hyvämäki

Title Data processing pipeline automation on cloud platform

Degree programme Life Science Technologies

Major Biosensing and Bioelectronics **Code of major** ELEC3045

Supervisor Prof. Simo Särkkä

Advisor M.Sc. (Tech.) Mika Vuorio

Date 18.2.2019 **Number of pages** 36 **Language** English

Abstract

A network connection provides us with a global access to a seemingly endless information storage. However, the ways of utilizing the Internet are changing. An increasing amount of functionality is being added to browsers, applications that are run on personal devices to transfer and interpret the online data. Applications that were previously run independently on personal machines are transformed into web applications, and the processing load is distributed away from individual computers. This development has increased competition in the web hosting industry. While the leading multinational technology companies have established large datacenters around the world to inexpensive locations, it has become profitable for smaller companies to outsource their hosting services. As datacenter technology has evolved, the enormous stocks of machines from which the large companies rent out their computing services has been started to be called *the cloud*. The transition to cloud services provides increased reliability, power, scalability, and cost-effectiveness for software businesses.

This thesis describes the process of transferring a public transportation journey planner service from company's own datacenter to a cloud platform. The consequential changes in different abstraction levels of the software architecture are documented, and the methods that were used in planning and executing the transfer are evaluated. A container virtualization method was combined with Amazon's container platform service and it increased the software reliability with automatic health monitoring and scaling. These new features of the system were verified by load tests, and their results show that the cloud platform is able to restore the normal response times of the service in five minutes after the number of users has grown drastically, by provisioning more resources for the routing algorithm.

Keywords cloud computing, container virtualization, docker, infrastructure as a service, data processing, automatic scaling, site reliability engineering

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Sampsa Hyvämäki

Työn nimi Datankäsittelyn automatisointi pilvialustalla

Koulutusohjelma Life Science Technologies

Pääaine Biosensing and Bioelectronics **Pääaineen koodi** ELEC3045

Työn valvoja Prof. Simo Särkkä

Työn ohjaaja DI Mika Vuorio

Päivämäärä 18.2.2019 **Sivumäärä** 36 **Kieli** Englanti

Tiivistelmä

Verkkoyhteys tarjoaa meille globaalin pääsyn näennäisesti loputtomaan tietomäärään. Tavat hyödyntää Internetiä ovat kuitenkin muuttumasssa. Selaimiin, verkossa olevaa dataa hakeviin ja tulkitseviin ohjelmiin lisätään jatkuvasti uusia toiminnallisuuksia. Aikaisemmin henkilökohtaisilla laitteilla ajettuja sovelluksia muutetaan verkkopohjaisiksi ja samalla henkilökohtaisten laitteiden laskentakuormaa siirtyy palvelimille. Tämä kehitys on lisännyt kilpailua verkkoisännöintipalveluissa. Kun johtavat teknologiayritykset ovat rakentaneet suuria palvelinkeskuksia edullisiin sijainteihin, on isännöintipalvelujen ulkoistaminen muuttunut kannattavaksi pienemmille yrityksille. Kun palvelinkeskusteknologia on kehittynyt, on suuresta palvelinmäärästä vuokrattavia laskentapalveluja alettu kutsua *pilveksi*. Pilvilaskentaan siirtyminen tuo ohjelmistoalan liiketoimintaan luotettavuutta, tehoa, skaalattavuutta ja kustannustehokkuutta.

Tässä diplomityössä kuvataan julkisen liikenteen reittiopas-palvelun siirtäminen yrityksen omasta palvelinsalista pilvialustalle. Työssä dokumentoidaan siirtoprosessista seuranneet muutokset ohjelmiston arkkitehtuurin eri abstraktiotasoilla, ja arvioidaan menetelmät siirron suunnittelussa ja toteutuksessa. Siirrossa yhdistettiin konttivirtualisointi ja Amazonin konttialustapalvelu, mikä paransi ohjelmiston luotettavuutta automaattisella valvonnalla ja skaalauksella. Järjestelmän uudet ominaisuudet verifioitiin kuormatesteillä, joiden tulosten mukaan alustalla on kyky kasvattaa automaattisesti reititysmoottorin laskentaresursseja ja palauttaa palvelun vasteaika normaaliksi noin viidessä minuutissa käyttäjämäärän äkillisestä kasvusta.

Avainsanat pilvilaskenta, konttivirtualisointi, docker, infrastruktuuri palveluna, datan käsittely, automaattinen skaalaus, verkkosivustojen luotettavuustekniikka

Preface

First of all, I would like to thank my advisor Mika Vuorio for giving me the opportunity for this thesis project, and his guidance throughout the project. Thanks to my supervisor Simo Särkkä, especially for his help on the academic style of writing.

Special thanks to my colleague Jyrki Posti for sharing his knowledge about Docker containers and giving suggestions for the thesis content. Credit also needs to be given to other fellow workers for their help.

Last but not least, I would like to thank my girlfriend Laura for her patience and comprehensive support for achieving results during the project, and my friends and family for their presence and support.

Espoo, 18.2.2019

Sampsa Hyvämäki

Contents

Abbreviations

API	Application programming interface
CLI	Command line interface
CI	Continuous integration
CPU	Central processing unit
DNS	Domain name system
EC2	Elastic computing cloud
ECR	Elastic container registry
ECS	Elastic container service
GUI	Graphical user interface
HTTP	Hypertext transfer protocol
IaaS	Infrastructure as a service
IP	Internet protocol
OS	Operating system
PaaS	Platform as a service
RDS	Relational database service
SaaS	Software as a service
SES	Simple email service
SMTP	Simple mail transfer protocol
SQL	Structured query language
TCP	Transport control protocol
VM	Virtual machine
VPC	Virtual private cloud
XML	Extensible markup language
YAML	Yet another markup language

1 Introduction

As stated by Sommerville (2013), the development of the Internet and the web has affected our lives profoundly. A network connection provides us with a global access to a seemingly endless information storage, that resides on other computers connected to the Internet. However, the ways of utilizing the Internet are changing. An increasing amount of functionality is being added to browsers, applications that are run on personal devices to transfer and interpret the online data. As browsers are being developed to run small programs and do enhanced local processing, applications that were previously run independently on personal machines are transformed into web applications that are run with browsers. The evolution of the web changes software engineering, as it is profitable to share, change and upgrade software through the web, and distribute the processing load away from individual computers. Therefore, the web is beginning to resemble more and more a globally shared supercomputer, for which our personal devices work as a user interface.

In the beginning of the Internet era of software engineering, it was common for technology companies to have their own datacenters, where they placed their remotely accessible machines, *servers*, that were connected to the Internet. According to Armbrust et al. (2010), as the leading multinational technology companies had established large datacenters around the world in inexpensive locations, it became profitable for smaller companies to outsource their hosting services. As datacenter technology evolved, the enormous stocks of machines from which the large companies rent out their computing services was started to be called *the cloud*. The name of the concept describes its order of magnitude and widespread nature, as while the individual servers are as hard to distinct from each other like water molecules in a meteorological cloud, they are also geographically scattered as the locations of the significant datacenters vary from Europe to Asia.

According to Armbrust et al. (2010), the use of cloud services provides increased reliability, power, scalability, and cost-effectiveness for software businesses that were once facilitated by the rise of the Internet. Currently, computation is being outsourced to cloud in many fields of technology from enterprise resource planning to scientific calculation in health-technology. For example, as reported by Bort (2014), a German-based European multinational software corporation has begun to develop and sell cloud-based data processing and resource planning software (SAP, 2019). Experimental cloud solutions are already being developed for health care: in their study about cloud-assisted health monitoring, Hossain and Muhammad (2016) presented an electrocardiography-based health monitoring service in the cloud. The prototype was running on Windows Server 2010 operating system in Amazon's server infrastructure. In his masters' thesis Zheng (2017) developed a cloud-based solution also for cardio graphic data collection and processing. The software was deployed in the servers of Amazon with Debian 9 operating system based on Linux.

This thesis describes the process of transferring a public transportation journey planner service from a company's own datacenter to a cloud platform (Figure 1). The consequential changes in different abstraction levels of the software architecture are documented, and the methods that were used in planning and executing the

transfer are evaluated. In conclusion, this thesis is a documentation of the realization and outcome of a cloud migration project and the applicability of technologies that were taken in to use to make the most out of a cloud platform. The suitability of the transfer process for other software setups is discussed.

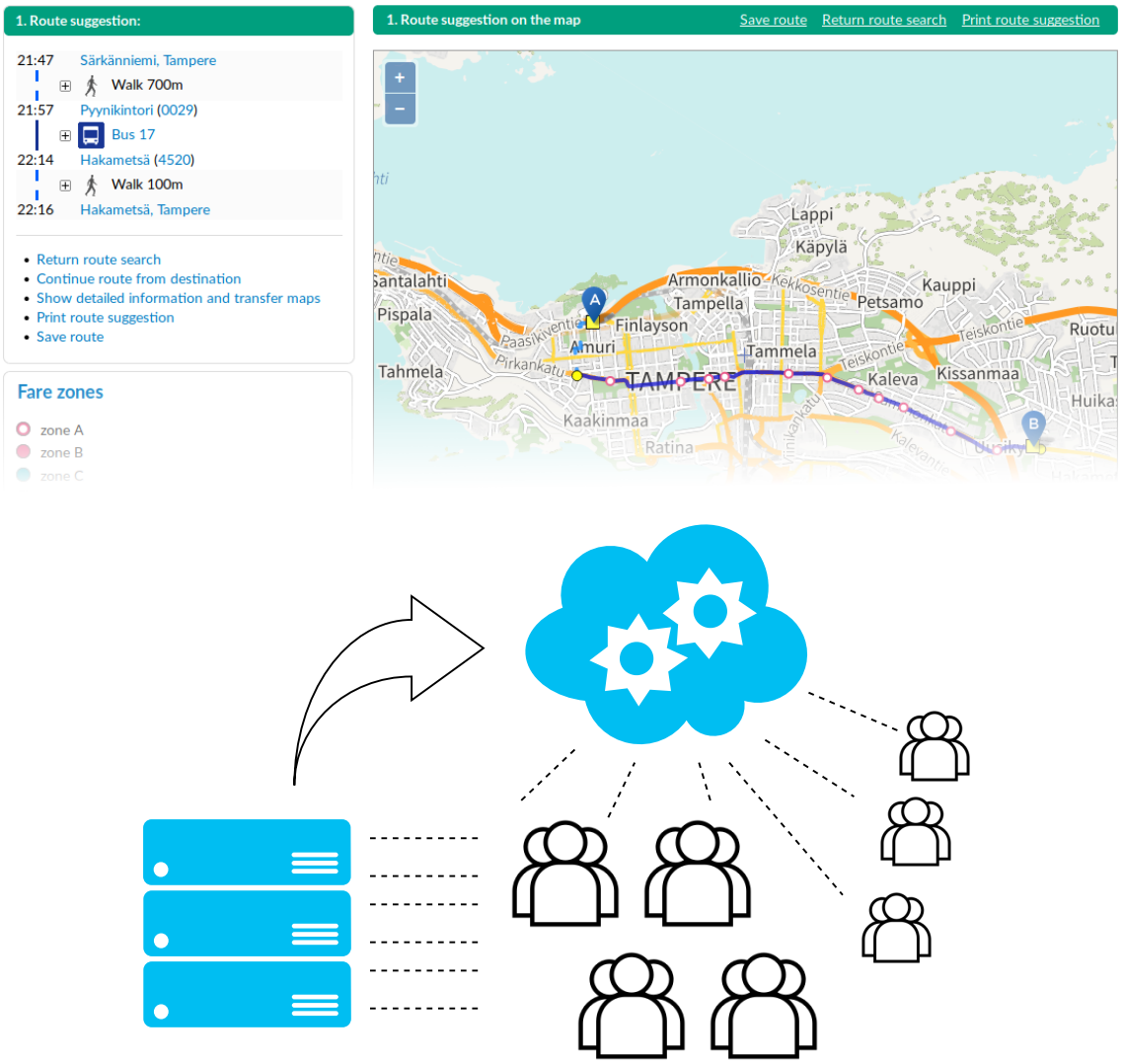

Figure 1: Conceptual illustration of a cloud migration of a web service software. Modified from The City of Tampere, CGI, National Land Survey of Finland (2019a).

2 Background

In this chapter, cloud computing, microservices, and serverless architecture are defined. The public transportation journey planner web service to be moved to cloud platform is introduced. The requirements for its computing environment are presented, as well as current methods of replicating the environment using cloud services.

2.1 Cloud computing

The NIST (National Institute of Standards and Technology) defines cloud computing as a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (Mell et al., 2011). These include networks, servers, storage, applications and services that can be rapidly provisioned and released with minimal management effort or service provider interaction. In their definition, Mell et al. introduce three cloud service models, which are Cloud Software as a Service (SaaS), Cloud Platform as a Service (PaaS), and Cloud Infrastructure as a Service (IaaS). They are defined by the amount of control that a consumer has over the provisioned computing resources. In SaaS, the consumer is able to use applications running on the cloud infrastructure, but does not manage or control the computing resources. In PaaS, the consumer is able to deploy his own applications to the cloud infrastructure, but has limited control over the provisioned computing resources. IaaS enables consumers to provision and control the computing resources themselves.

There is typically at least one software layer that exists between the resources that are visible to the client and the real hardware of the cloud servers. The cloud platform providers give tools for performing computation in a *virtualized* environment that includes virtual network, memory and central processing unit components similar to their physical counterparts in a server room. A provisioned central processing unit (CPU) from a cloud service is usually a part of a bigger CPU that has multiple cores (logical execution units). Furthermore, the memory for a provisioned hard disk drive is often within a larger physical drive. Therefore, the combination of cloud resources that form a complete computing unit resembling a traditional server with an operating system is called *a cloud instance* or *a virtual machine (VM)*.

The same basic functionalities are offered by every major cloud service provider, including Amazon (Amazon Web Services), Google (Google Cloud) and Microsoft (Microsoft Azure), but their naming conventions for the resource components are different as well as the ways to provision them. According to Panettieri (2017), Amazon Web Services (AWS) had a 40% share of the IaaS and PaaS sector in all global cloud service market in 2017. All of its biggest competitors had more than three times smaller shares, but their percentages have been increasing. Amazon has also established a publicly recognized certificate program to license professionals users of its cloud services, which indicates that while being the biggest provider, AWS aims to become an industry standard (Amazon, 2019).

In their view on cloud computing, Armbrust et al. (2010) explain how the cloud service providers have an advantage over others in the field of computing services. In

their opinion, establishing extremely large data centers at inexpensive locations has remarkably lowered their running costs of electricity, network bandwidth, operations, software and hardware. Armbrust et al. also state that in addition to the decreased direct expenses per resource unit, the methods to provision smaller shares from the large amount of resources have improved. Therefore, the cloud service providers are able to offer services under the hourly price of medium-size data centers and still keep their business profitable. Khajeh-Hosseini et al. (2010) studied a process of migrating an information technology system for oil and gas industry with multiple companies as stakeholders from a small in-house data center to AWS, and found the cost decrease to be 37% for the system infrastructure for over five years. Furthermore, having an own datacenter becomes less appealing for companies as the cloud service providers choose the locations of their datacenters to bring the servers across nations' borderlines, such as an American technology company Google did by constructing a datacenter in 2018 to Hamina, Finland (Google, 2019). In addition, the interest in cloud computing rises from the appearance of unlimited computing resources that are available for companies starting with a need for small capacity, but preparing for a rise in demand, such as start-up companies that utilize web technologies.

When designing a datacenter for a web service, it has been safe to estimate the sufficient size of the server infrastructure by the amount needed to be able to serve every client during the peak hours of traffic load. However, during periods of low traffic load, there remains an excess amount of computing resources (Figure 2 (a)). According to Armbrust et al., using scalable cloud computing resources with occasionally higher hourly rates than the relatively constant rate of one's own datacenter, it is profitable to have a scalable infrastructure and keep the quality of service stable. Armbrust et al. demonstrate two scenarios where underprovisioning negatively affects end-user experience (Figure 2 (b) and (c)) and thus, the generated revenue. In the scenario of Figure 2 (b), there is a constant loss generated by the underprovisioning. In the scenario of Figure 2 (c), the underprovisioning leads to permanently diminished amount of users, which causes more loss of revenue than scenario one.

Additionally, a change in the development of computing technologies has made the use of cloud services appealing. As the size of a transistor has kept decreasing, their number in the cores of CPUs has grown. However, as Sutter (2005) mentions in his article, physical limitations, such as the generation of heat, has stalled the rise of clock rate, that corresponds to the speed by which a single transistor can perform consecutive computation tasks. Therefore, according to Sutter, current approach in software development is to utilize parallelism, which means dividing the computing tasks between multiple CPUs. Also, according to Walter (2005), the density of data storage rises and the handling of large amounts of data benefits from parallel processing. Instead of investing to a single machine with multiple CPUs and large amount of storage space, it is more cost-efficient to use cloud computing resources. As Armbrust et al. point out, a company doing analytics by batch processing large datasets could benefit from using 1000 VMs for one hour during a batch run, instead of using one VM for 1000 hours, which is equally expensive on a cloud platform. Nevertheless, as Dean (2009) stated, the parallel cloud computing systems are still

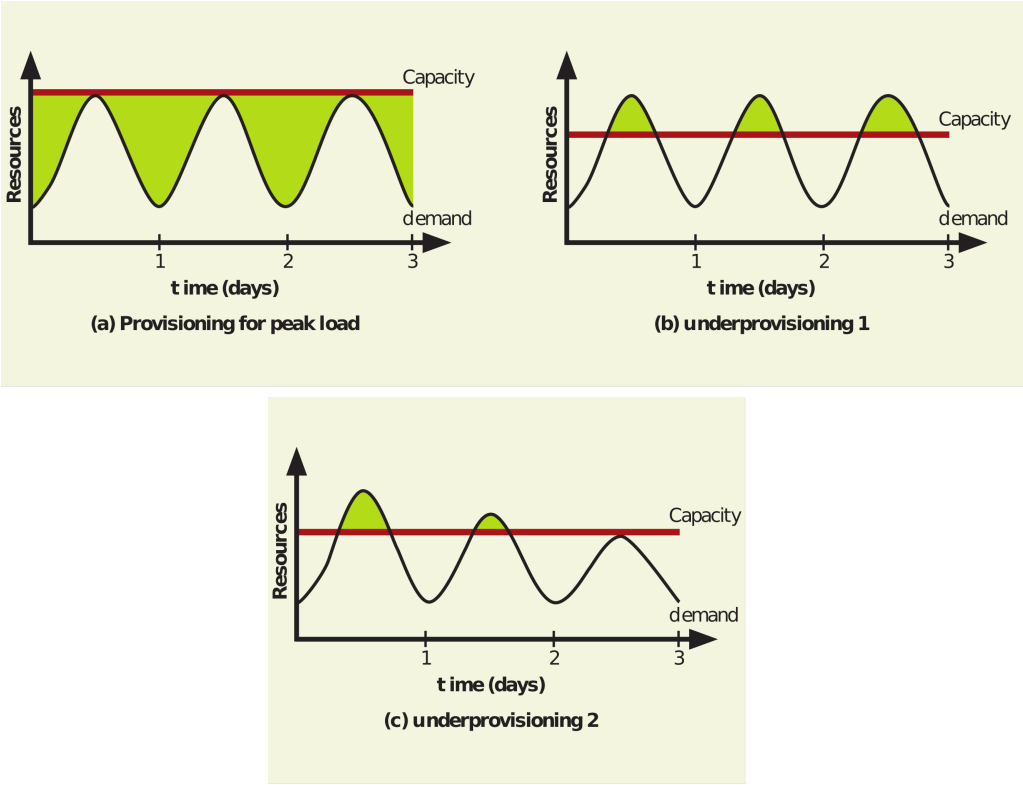

Figure 2: Effects of inaccurate provisioning of computing resources. In (a), green area represents the wasted resources in provisioning for peak load. In (b) and (c) the green area depicts sacrificed potential revenue. Modified from Armbrust et al. (2010).

vulnerable to hardware faults, for which the probability is proportional to the total number of provisioned hardware components. Thus, the software for business-critical batch processing using large amount of computing resources needs to tolerate the faults and is harder to design. However, the use of cloud resources takes away the effort from customers to monitor the hardware health themselves, as e-mail notifications are sent from instances running on degrading hardware before they are automatically moved to functioning hardware during reboot.

2.2 Microservices architecture

The traditional approach for designing a software environment to serve multiple clients over the Internet has been to dedicate computing resources persistently for each part of the service software. For example, a logical resource division for a public transportation journey planner would have separate, customized computer clusters for webserver software, timetable database, and routing algorithm. As Balalaie et al. (2016) describe in their study, microservices aim to realize software systems as a packages of small services that are independently deployable on different application platforms and technological stacks. Regarding the journey planner example, the

mentioned parts of the service could be developed further to microservices that have their own separate virtual environments. Furthermore, instead of running the whole service in a shared physical server rack, separating the computation of each microservice to its own cloud instance would have benefits regarding service reliability.

According to Balalaie et al. using microservices architecture would not only increase the hardware fault tolerance of the whole service, but also provide adaptability to technological changes, reduce time-to-market and improve development team structuring around the services. First of all, distributing the service to run on different cloud instances, it is possible to isolate the physical computing environments of different microservices. While it is rarely possible to define the actual physical machine where the cloud instance is located, one can choose to run the cloud instances on areas that are physically isolated from each other. For example, the data centers of AWS form regions, that each contain multiple availability zones (Figure 3). There is no information provided on how the availability zones are physically isolated from each other, but according to their documentation, Amazon (2018) claims that by launching instances in different availability zones, it is possible to protect applications from a breakdown of the other location.

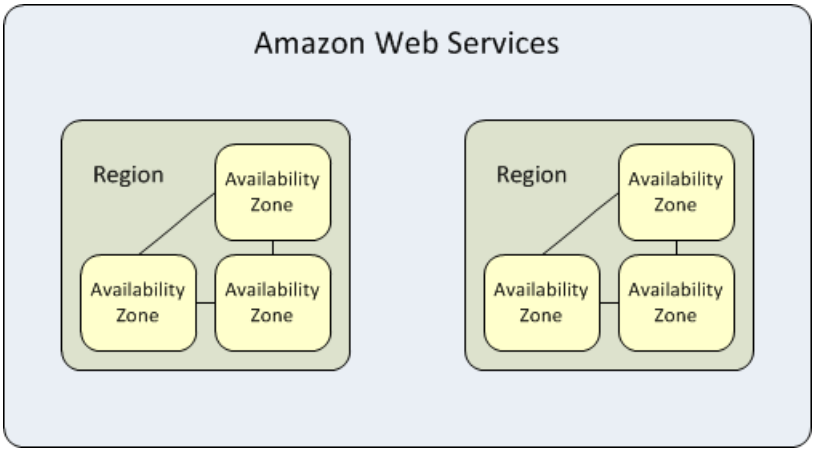

Figure 3: Division of the physical computing areas in AWS (Amazon, 2018).

2.3 Serverless cloud architecture

The concept of serverless computing has gained interest during recent years, though according to Fox et al. (2017), the definition for *serverless* is not clear. Usually, it relates to making short-running computation on a cloud platform, which is billed with millisecond granularity by the cloud service provider. However, a physical server running the computations always exists, which makes the term to slightly contradict itself. In serverless computing, an arbitrary server machine is provisioned to perform computation for the customers of the serverless computation service. Serving a large number of users with multiple machines has been the de facto method to run a web service for a long time, but when a CPU core from a large pool of machines is

provisioned for few milliseconds to a small piece of software after a client request, it is clear that the methods to serve the clients have changed. Along with the growing popularity of microservices model and parallelism, the granularity of computational tasks is becoming finer, and the precise physical location of the code execution is becoming less important.

A use-case for the serverless computation is, for example, to process journey planner routing requests by on-demand basis with a code stored in cloud. From a traditional system administrator's point of view, the benefits of using serverless cloud computing are that the amount of server environment maintenance work diminishes and, for example, the software developers are able to transfer their code to the cloud environment by themselves with a simple user-interface of the serverless computing service. Other advantages for software development processes are that the development of demanded software components can easily be outsourced by using serverless computing services. For example, considering a micro service, the development of missing piece of data processing software could be outsourced with a small and well-specified offer to a developer familiar with the serverless computation service platform of a specified cloud provider. A developer specialized in a specific type of software development would take a look at the offering, and make a solution that would not only match the task but also produce an optimized outcome.

2.4 Virtualization with containers

There are two main methods of virtualizing an environment for running software written in high-level programming language, as stated by Turnbull (2014). In hypervisor virtualization, a virtualized operating system (OS) runs independently on physical hardware via an intermediate software layer called a hypervisor. The virtualized operating system is also referred to as a guest OS or a virtual machine (VM). In container virtualization, a user space is run on top of operating system's kernel. It is also called operating system-level virtualization. As modern web service software is designed to run in microservices and process requests from thousands of end-users per day, the scalability and fast deployment are important factors when the choice of virtualization method is considered. The important difference between containers and VMs is that the container environment does not include the operating system's kernel, as it shares it with the virtualization host (Figure 4). Therefore, the start-up time for a container is shorter than for a VM, which facilitates the service restoration. The size of a container image is in tens of megabytes, whereas the VMs take gigabytes of disk space. The difference in size makes the containers easy to replicate if the service has to respond to an increased load.

Docker is an open-source container engine, which provides an automated workflow to deploy applications to a container environment. It uses layered filesystems called Docker images to describe the containers. The container images are built using series of instructions in a file called Dockerfile, and each instruction adds another layer in the filesystem. This way, when there is a change in the Dockerfile, it is not necessary to run the whole building process again. It facilitates the development process as, for example, making a change in the configuration of some container application does

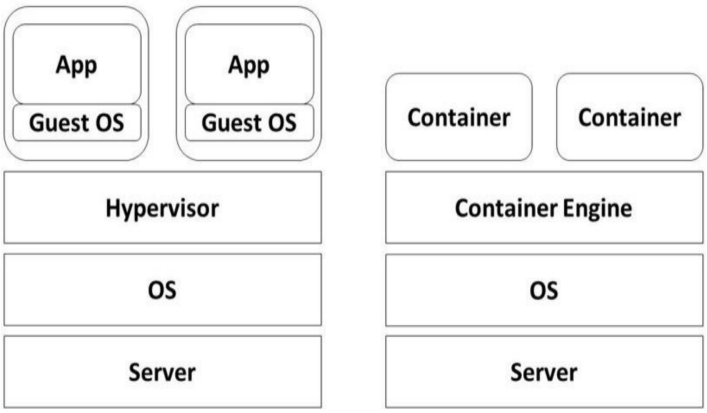

Figure 4: Comparison of containers and virtual machines (Joy, 2015).

not require installing the whole application in the container again. The layered file system is part of the whole container environment, as demonstrated in Figure 5. In the foundation of the environment is the host OS's kernel and its important features, such as control groups (cgroups), namespaces and device mapping. The kernel is usually located in a separate hard disk partition in the host OS, which is the part of the environment that the containers share with the host. By using control groups it is possible for the container engine to limit the computing resources that the container uses, such as the CPU utilization, and allocated kernel and block device memory. The memory that the kernel uses is typically random-access memory, and a block device refers to a hard disk storage. Namespaces are used by the kernel to isolate processes running on the system and, for example, to enable a virtualized network stack for the container. Device mapper is a Linux kernel feature that enables the building of a layered filesystem. According to Turnbull, the Docker uses a union mount to form the layers. In union mount, several filesystems may exists in the same location under a directory tree and still appear as one filesystem for applications.

Every instruction in the Dockerfile creates a new filesystem by either adding new files, or modifying the ones in the old filesystem by copying them to the new filesystem and writing changes to them. The modified files exist in the previous file system, but remain hidden and are accessible by only new builds that are that initiated with changed instructions in the Dockerfile. In Figure 5, the blue blocks represent the new layers added by the instructions. The base image is an optional layer adding utilities that are assumed to be needed in the container. Several base images exist that developer communities have made to resemble the utilities in popular Linux distributions, such as in Ubuntu or Debian. Other layers in the example are a text editor application Emacs and a webserver application Apache. The top layer is the storage space that is reserved for the container to use during its run time.

There are several types of instructions available to use in the Dockerfile, but there are three instruction types needed to create a purposeful container, FROM, RUN and CMD (Turnbull, 2014). The FROM-instruction describes the base-image which, if not available locally, may also be downloaded from a specified remote registry. The RUN instruction executes a command inside the container and when finished, adds another

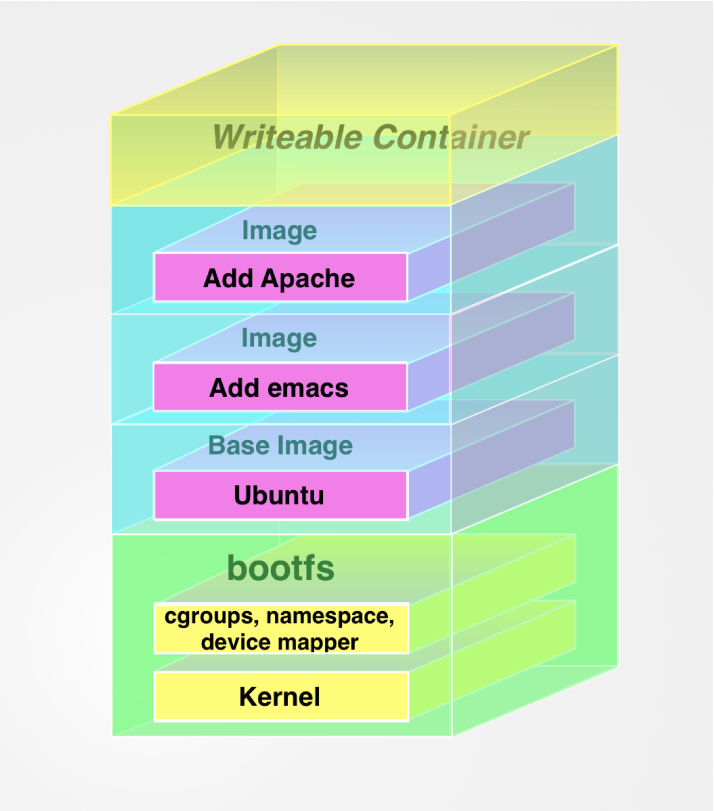

Figure 5: Layers of filesystems in the Docker container environment (Turnbull, 2014).

layer to the filesystem. For example, as in Figure 6, the RUN instruction is used to update the Debian software package repository to ensure that the latest versions of applications will be installed, and later to install Emacs and Apache, using *-y* option to automatically answer to the confirmation prompt during installation. The CMD instruction is used in defining an executable to run after starting the container. As container images are not able to store the kernel's memory, it is not possible to save the state of a container that is running an executable binary. Therefore, the binaries need to be executed separately. In the example, Apache-webserver is started as a foreground process, meaning that other container processes need to wait for it to stop. This prevents the container from shutting down, as it does without the presence any running foreground processes.

```
1 FROM ubuntu:latest
2 RUN apt-get update; apt-get install -y emacs
3 RUN apt-get install -y apache2
4 CMD ["apache2ctl","-D","FOREGROUND"]
```

Figure 6: Example of a Dockerfile containing instructions for building a container image with text editor and webserver software.

2.5 The journey planner web service

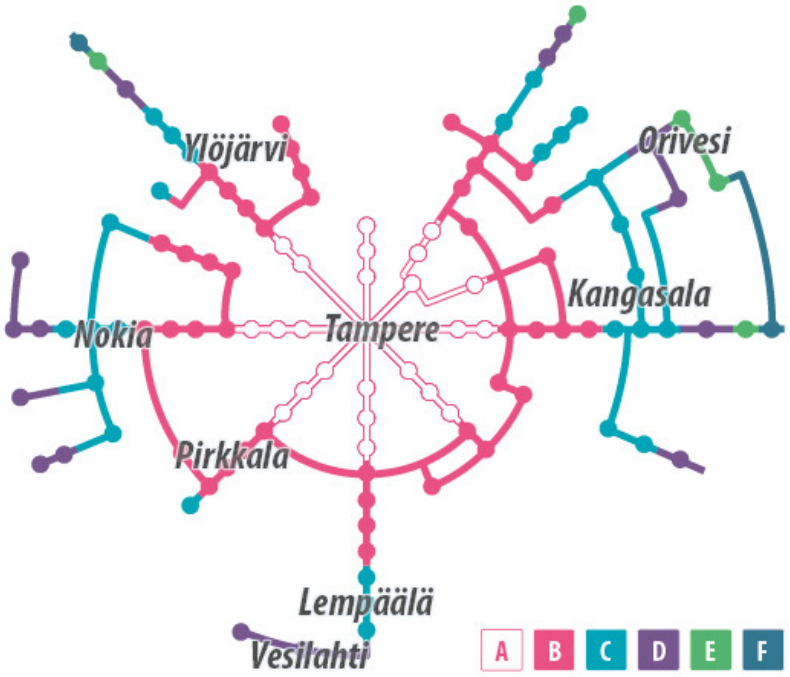

Figure 7: A simplified line model of the Tampere region public transportation system with traffic zones (The City of Tampere, 2019).

The software to be migrated to cloud platform was a public transportation web service for the city of Tampere. Being the most populous inland city in the Nordic countries, there is an essential need for a functioning public transportation system (Figure 7), and the service has a great role in providing the key travelling information to the people living in the area. The goal of migrating the service was to not only extend its lifetime by making a transfer to a new platform, but also improving the service without clients experiencing downtime.

The web service consisted of a multimodal journey planner, a timetable service with real-time bus arrival time estimates, a traffic monitor service for visualizing bus traffic on a map, a transit map for visualizing bus lines, and a cycling and walking trip planner (Figures 8 and 9). The software was designed to serve multiple users simultaneously over the Internet. As the users make requests with browsers, the web service software processes these requests. In general, it uses routing algorithms, databases and third-party application programming interfaces (APIs) to fetch and process information according to the requests, composes the information to an understandable format and sends it back to the clients.

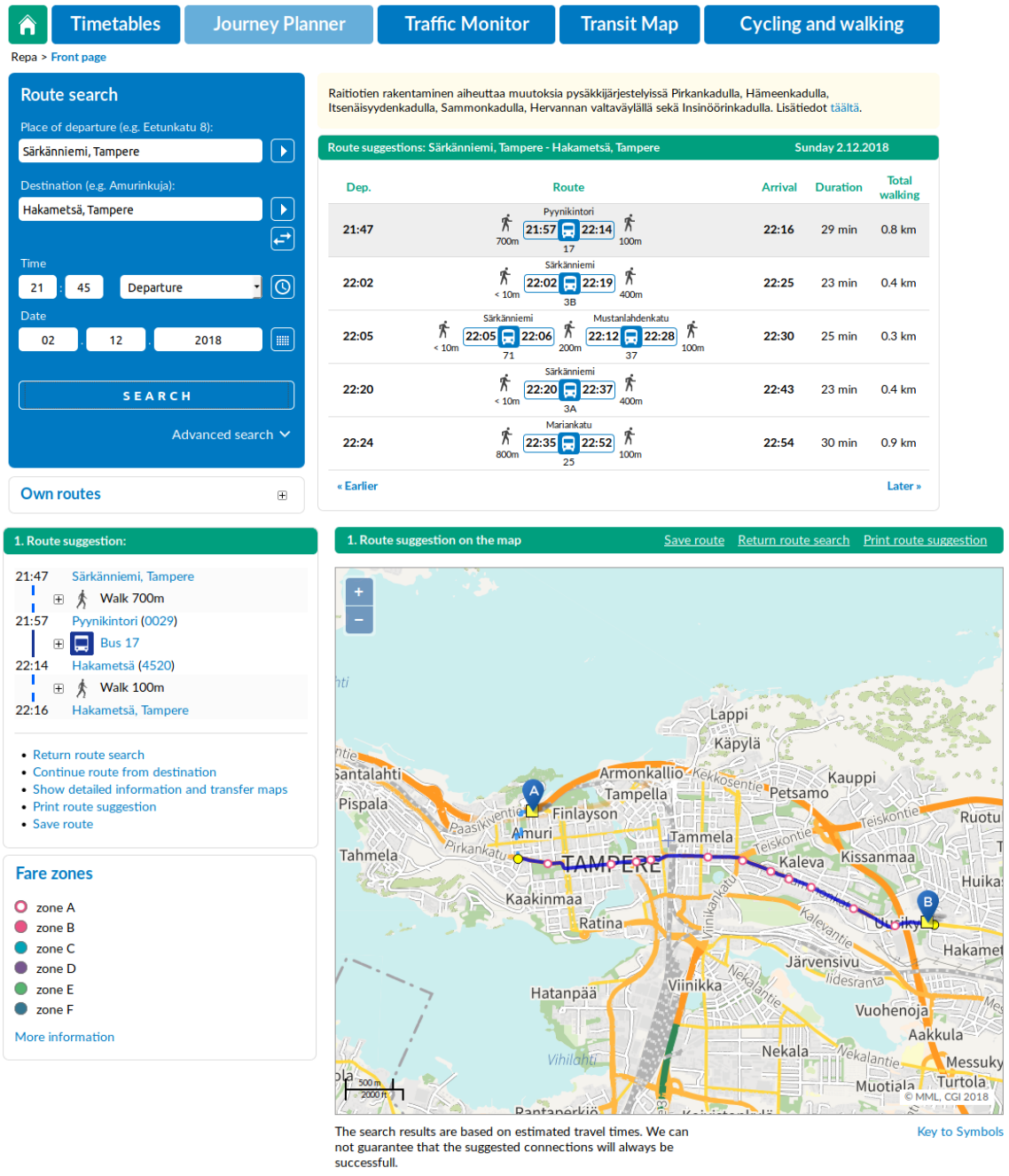

Figure 8: The graphical user interface of the multimodal public transportation journey planner web service for the city of Tampere and an example route search (The City of Tampere, CGI, National Land Survey of Finland, 2019a).

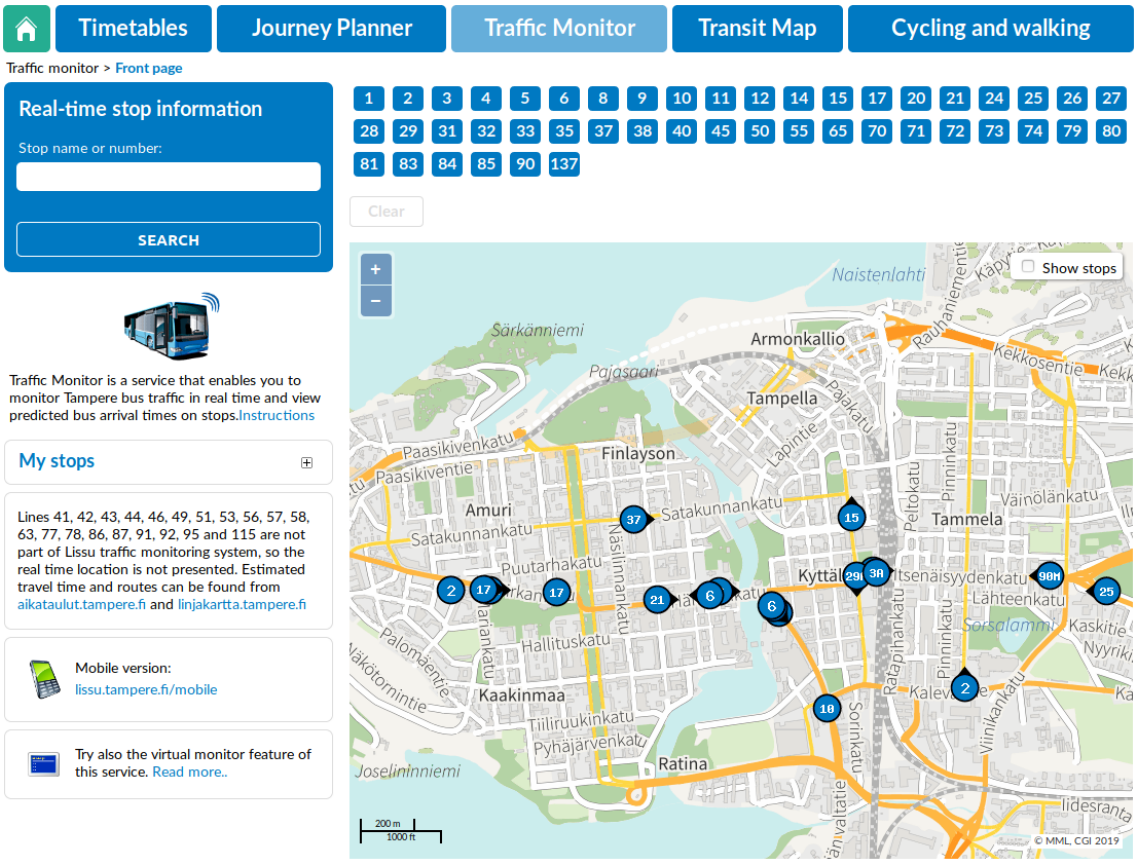

Figure 9: The graphical user interface for the traffic monitor web service (The City of Tampere, CGI, National Land Survey of Finland, 2019b).

The hardware requirements for the web service to process one request were not considerable for a modern computer. The web service software consisted of a webserver, algorithm, and database software, which were all possible to be run simultaneously on a computer with a single CPU, 4 gigabytes of random-access memory (RAM), and 10 gigabytes of disk space reserved for the software. However, running the official service with that setup as *production environment* is not possible, as the hardware requirements for processing multiple requests per second are not met. By not having enough resources, the response times for requests will grow and the requests get rejected when the length of a request queue exceeds a maximum limit. Therefore, the production environment was constructed by installing each part of the service software on multiple computers that were dedicated to run a specific part of the service. The computers with equal tasks formed clusters, among which the processing was distributed (Figure 10). The computers had static Internet protocol-addresses (IP-addresses) associated with them, by which they were discoverable in the private network. A separate load balancer was used for distributing the requests to one of the machines in the webserver-cluster, which in turn used routing algorithm on one of the algorithm cluster machines and database on one of the database clusters machines. The domain name of the service (reittiopas.tampere.fi), had been configured to point

to the IP-address of the load balancer in the domain name records of the public name servers. The algorithm used in the routing was based on Dijkstra's graph search-algorithm. The databases contained information of the routing network and timetable data for the public transportation in Tampere region.

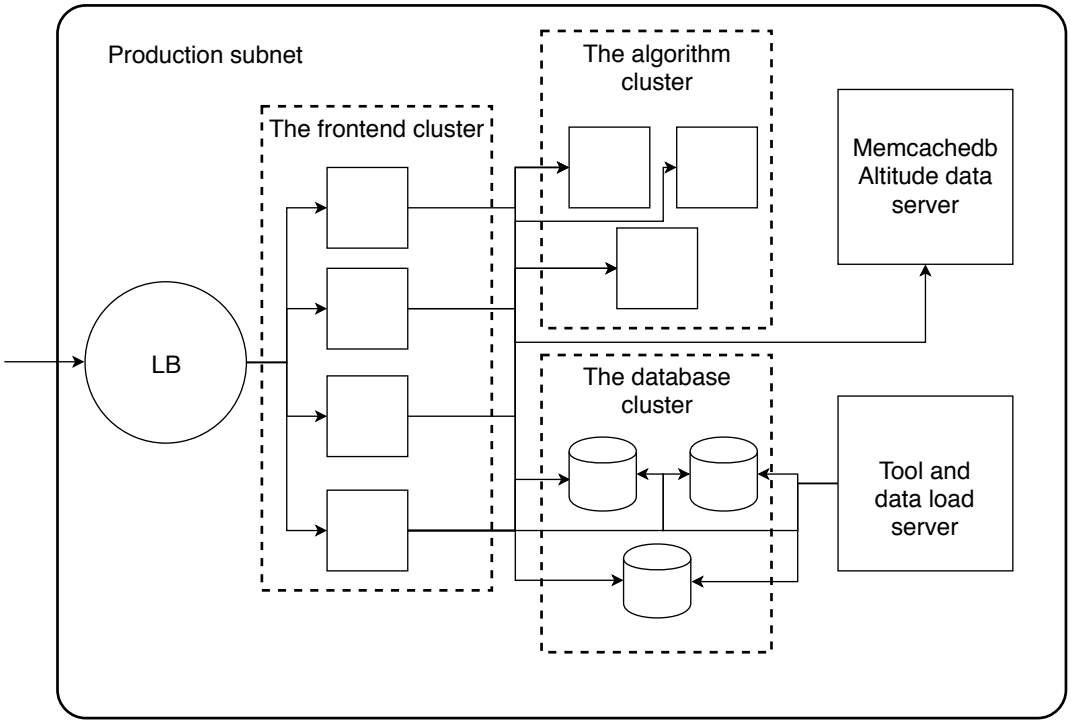

Figure 10: The structure of the old server environment.

The webserver software, which was also called *the frontend*, was also responsible for enabling a secure connection between the server and the client. This was done by providing an encryption key for an asymmetric encryption. The key was certified by a certificate authority, which was used for ensuring the authenticity of the service provider to the end-users. There was also a feature that enabled the client to send journey suggestions via email, which was enabled by installing a mail transfer agent on the machines that run the frontend.

2.6 The DevOps philosophy and maintaining site reliability

According to Brunnert et al. (2015), the need to continuously adapt software to changes in the evolving business environment has intensified among the competition between global software providers. A hard division between development and operations teams slows down the process of software changes ending up to the production environment. Whereas the development team (Dev) aims to respond to changes in the functional and quality requirements for the software, the operations team (Ops) monitors the performance of the software in its execution environment and responds to environmental changes, such as increased workload and security updates on the host operating systems. As stated by Brunnert et al., DevOps practices concentrate

on facilitating the process of deploying updated software to production environment, as well as minimizing the issues being generated back-and-forth between the teams. These practices involve automation in configuration management, testing, monitoring, and version control as well as in infrastructure management and software deployment. However, it should be mentioned that the concept of DevOps remains controversial and thus might have different meaning based on the context.

The ability to track changes made in software and isolating the latest production version is important in all areas of software development. Making small incremental changes in software eases the tracking of faults and keeping the knowledge of how the software works. As version control can track changes of files, nearly everything produced in software development can be kept in version control. It is common for developers to have both local version control for storing their own changes in the software, and a common version control hosted on a company server. The shared version control holds the *master* code repository, and is responsible for storing a version of the software that has the most recently developed new features.

In their book of site reliability engineering, Beyer et al. (2016) promote a self-service model in web software development, where the development teams should be as capable as possible to manage the process of releasing new software version to the production environment themselves. Beyer et al. mention the importance of high *release velocity* in user-facing software, as frequent releases result in fewer changes between the software versions, which makes testing and troubleshooting easier. An example solution of automating release processes is using automation server, such as Jenkins, that is able to detect a change made to the version control, and automatically deliver a new version of the code to the production environment (Jenkins, 2019).

The release process has typically intermediate steps for quality assurance and to facilitate the restoring of the previous software version. For example, a software installation package may be built in some archive file format, such as Debian package. The package contains the necessary files in order run the software and some additional information, such as version numbers of the software itself and of the dependent software, that also need to be installed in order for the software to work correctly. An automatic installation of the software is enabled through the software package managing, and usually there are different release processes configured to deploy the software to server environments of different purpose, such as test, demonstration and production environments. The different environments having different software configurations is considered in, for example, the package building process, by including different configuration files to the software depending on which environment the package is supposed to be deployed.

2.7 Software migration project

The coordinated approach for the project combined Scrum-based methods and Lean philosophy (Sutherland and Schwaber, 2017; Poppendieck and Cusumano, 2012). The known steps in the project were ordered by priority to a task list called backlog, and every week this list was updated in a meeting called weekly Scrum. In the

meeting, the team members involved in developing the Docker and configuring the new cloud environment discussed about the status of existing tasks and the new tasks and their work estimates. The motivation for the project was not only the aging hardware, but also one of the principles of Lean Software development, constant learning. According to Poppendieck and Cusumano (2012), in the end, development is all about creating knowledge and embedding that knowledge into a product. In the migration project, a learn first-approach was used, which means to first explore multiple options for expensive-to-change decisions (such as virtualization method and cloud architecture) and delay critical decisions to the last responsible moment. Finally, the decisions that best optimize the overall system are made and they are based on the best available knowledge at the time. Additionally, after exploration, there will always be alternative options that work (Poppendieck and Cusumano, 2012).

In his book about software engineering, Sommerville (2013) discusses different process models for software engineering, one of them being a model for reuse-oriented development (Figure 11), based on integration and configuration. According to the model, a first step of the migration process is to specify the initial requirements for the system. The requirements do not have to be in detail, but give rough guidelines for the project. In the following steps, a search for suitable software components are made, and their applicability is evaluated. The initial requirements are refined according to the evaluation, but the software discovery and evaluation steps can be done again if it is impossible to make changes to the original requirements. The discovered software may either be configured to be used off-the-shelf, or components of it may be adapted to the existing system, and the missing parts developed. When migrating web services to a cloud platform using PaaS and container virtualization, the evaluated software components are the different script languages, APIs and GUIs that the virtualization and cloud service providers offer. After choosing the right components, the existing application is adapted to use them by configuring their settings according to the new environment.

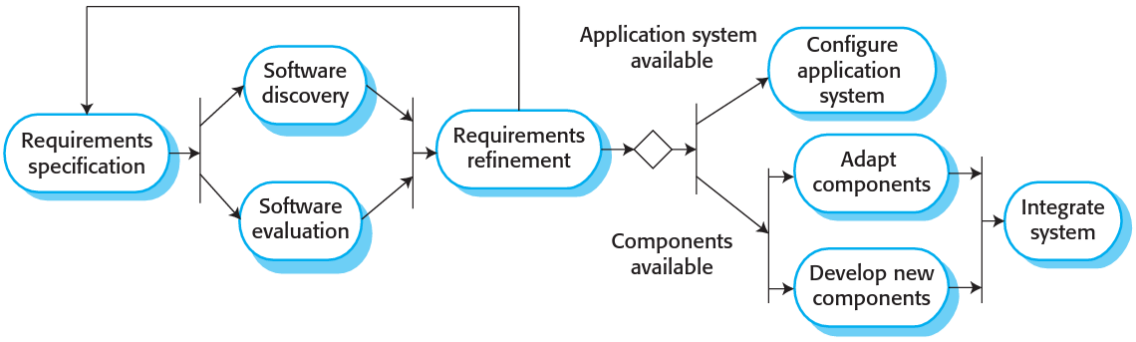

Figure 11: A flowchart of a process model to be used with reuse-oriented software engineering (Sommerville, 2013).

The existing studies of migrating service-oriented systems to cloud computing provided insight on the scale of the migration project and also helped to specify the initial requirements, which conformed to the process model described by Sommerville (2013). In their experience report on migrating a data processing automation framework to a cloud platform, Chauhan and Babar (2011) identified four main requirements after inspecting the properties of SaaS and IaaS cloud models, that affect the architectural decisions:

(1) The system should be scalable to respond to changing performance requirements.

(2) System components should be deployable on different IaaS cloud platforms.

(3) Components of the system should use sustainable data storage solutions in the IaaS cloud platform.

(4) The client interfaces that provide access to the system should remain unchanged.

To evaluate whether the requirements are met, Chauhan and Babar defined attributes related to the requirements to verify the system design qualitatively (corresponding requirement in brackets):

- Scalability: the support for replication of system components to respond to the varying performance requirements. (1)
- Modularity: the capability of running the replicated copies of the system components on multiple physical or virtual cloud instances. (1)
- Changeability: the feasibility to adapt the system to changes in the cloud platform. (2)
- Portability: the possibility to change the cloud infrastructure and service provider while retaining the system components as they are (2). Also the ability to separate database implementation to its own component (3).
- Backward Compatibility: The extent of which the end-users experience changes in the client interface. (4)

Chauhan and Babar implemented the migration in four steps that were coherent with the quality attributes. First, each component of the system was evaluated for scalability. For a component to be replicable, it has to be stateless, which means that multiple requests from the same client can be processed by different instances of the same component. Components implemented as Representational State Transfer (REST) services are stateless by nature, so they were possible to be replicated. REST service component is, for example, a server software that handles the client requests by fetching data from database, processing it and sending the end result to the client. The database components are not stateless as it is not possible for the REST instances to have their own database, so they were left out from replication. In the second

step, Chauhan and Babar evaluated their self-developed component orchestrator, that provided interface to the replicated components and managed the replication process. The third step consisted of re-factoring the existing components in order to make them scalable and function in the cloud platform. In the final step the whole solution was evaluated in a target cloud environment. Chauhan and Babar made conclusions that it is important to consider the target IaaS infrastructure during the system design, that cloud service components that are not hosted by several providers should not be used, and that the software systems that consist of stateless components are easier to migrate with IaaS.

3 Materials and methods

To facilitate the use of all cloud features and to use them in their full potential, architectural changes were made in the journey planner software. Load tests were used to verify the features, especially the automatic scaling of the service. In this chapter, the process of evaluating the transfer of the software between the old and new server environment is described.

3.1 The cloud service provider

Amazon Web Services (AWS) was chosen as a cloud service provider, as it was recognized as the most relevant provider at the time of thesis project. It had many characteristics that were suitable for the software being migrated and the current cloud-service experience of the developers. Also, the size of its market share was promising for choosing a sustainable service provider and AWS provided extensive documentation including step-by-step examples of using the cloud resources. Furthermore, online introduction seminars were organized regularly, as well as local training events to help first-time users. The final incentive to choose AWS was the launch of Fargate, a new virtualization platform in the Europe region. In AWS, the virtual infrastructure can be configured with a web browser client with a graphical user interface (GUI), but a command line interface (CLI) is also provided that enables the automation of the infrastructure management.

3.2 The container structure

The created Docker containers were made to resemble the clusters of the original software architecture. Separate containers were made to include frontend webserver software and routing algorithms. All the varying amount of traffic was directed to scalable instances, and the logical division between the original clusters remained to aid maintenance and debugging. The containers were based on Debian Docker base images, corresponding to the original choice of OS in the server environment. As Debian packaging was already implemented for the software, the container images were constructed by installing software packages and adding configuration files to the base image. Also, the Dockerfiles included installations of tools for debugging purposes, setting up correct time zone and defining user rights for increased security.

As a by-product, the containerization made it possible to easily set up a local development environment. So far the development had been made in a remote server, where the webserver software, database, and algorithms were constantly running. As well as easing the transfer to cloud environment, keeping the necessary components for the application in containers removed maintenance tasks related to the separate development environment.

To deploy containers in the AWS, one available solution was to use Docker-compose, which is a tool that can build multiple Docker images and deploy them in containers according to instructions in a compose-file, which is in YAML-format (Yet Another Markup Language). YAML is a human readable data interchange format,

and as Ben-Kiki et al. (2005) explain in their article, it supports more complex formatting and is therefore more readable than many other common languages for data transfer and storing, such as XML (Extensible Markup Language). As an example, Figure 12 shows that while clarifying indentation is possible in XML (Bray et al., 2006), it is hard to create as clear nested structure as it is with YAML, as every information element requires a start and end an tag. As stated by Ben-Kiki et al., it is possible to use indentation in YAML to create nested elements, use dash to indicate sequence entries, and map keys and values with colons.

```yaml
version: '3'
services:
  frontend:
    build: "./frontend"
    ports:
     - "80:80"
  routing_algorithm:
    build: "./routing-algorithm"
```

```xml
<root>
        <version>3</version>
        <services>
                <frontend>
                        <build>./frontend</build>
                        <ports>80:80</ports>
                </frontend>
                <routing_algorihm>
                        <build>./routing-algorithm</build>
                </routing_algorihm>
        </services>
</root>
```

Figure 12: A comparison of YAML- and XML-markdown languages.

The Docker-compose was able to launch containers to a same virtual private network without explicit configuration. In the compose-file, the network ports of the containers were mapped to the ports of the host operating system, so that the system could be tested with a web browser in the host OS by making hypertext transfer protocol (HTTP) GET-requests to the hosts local IP-address. An identical structure was launched on the Amazon Elastic Computing Cloud (EC2) instance with Linux operating system. This was done by installing Docker and Docker-compose to it, and transferring the compose file and container images to the cloud. The container images were stored in Amazon Elastic Container Registry (ECR). By configuring a security group for the EC2 and allowing a network port for HTTP, the service was publicly discoverable by the public IP-address of the instance. This was the simplest

solution for the whole service migration. However, the solution lacked capabilities for monitoring, scaling, and automatic service recovery.

To add the missing functionalities to the container platform, it was convenient to use container services that already existed in AWS. The Elastic Container Service (ECS) was able use other services in AWS to provide monitoring features (the Amazon Cloudwatch), automatic scaling (EC2 scaling groups), and automatic recovery. It also provided a method to utilize the already developed compose-file. A command line tool called ECS command line interface (ECS-CLI), was able to parse the same YAML-file that was made for the Docker compose, and launch the containers to a pre-defined cluster of EC2-instances. Thus, the same compose files could be used in defining both local container environment and a container service in AWS. However, ECS-CLI required another YAML-file to configure the container platform, as the compose files did not define, for example, the security groups that were used in limiting the network access to the containers.

Continuing the migration project with ECS-CLI confronted challenges as the service structure created with the compose files was not efficient enough to be used in the cloud. In ECS, the containers to be run in a service were defined by a task definition. Using the ECS-CLI, there was a limitation to create a single service from one compose file, as the ECS-CLI interpreted the file as a task definition. Running multiple containers in one task means that if one container goes in an unhealthy state, the whole task is restarted and all the containers with it. Scaling the service during a high traffic load would then only be possible by starting new identical tasks and increase the number of all containers. Thus, creating an efficient service structure with Docker compose and ECS-CLI would have required multiple compose files, which would have taken away the benefits of describing the whole container environment with one file. An efficient microsevice architecture allows the independent scaling of different services, which was possible in ECS only by running every container in their own service. During a high traffic load only the number of the needed containers would be increased. For example, during a period of high demand for routing services, only new routing algorithm containers would be started, if the amount of other containers was already enough to withstand the increased load.

Another approach in ECS was Fargate, which instead of an IaaS, resembles more of a PaaS. With Fargate-launch type in AWS, the virtualization hosts were completely managed by automation, and only the container images to be started were chosen. With Fargate it was possible to easily create services for different containers only by using the graphical user interface in the web browser called Amazon Management Console. Also, instead of using static IP addresses for different containers, the AWS Fargate used AWS Route 53 Domain Name System (DNS). The Route 53 allowed the Fargate to assign any IP to a container from the address space of the private subnet, and create a name record for the address to an inherent DNS-server. This way, the containers were able to locate other containers launched with the Fargate by their names, as the container hosts in Fargate used the DNS-server for resolving IP addresses that were associated with the names. It allowed easier dynamic scaling of the different container types in the cluster, as it was possible to create multiple records with identical names. When requesting an IP address for a container name

with multiple records associated to it, the DNS-server returned the different IP addresses in a round-robin fashion. Each address was picked from the list of possible addresses one after another and returned after the request. Therefore, the traffic for all the containers with same name was distributed equally. In the case of a crashed container, Fargate also took care of removing its IP-address from the DNS-server. Also if some of the containers would require an update, the task definition could be revised from the management console to use an updated container image.

The database solution was also PaaS, as a Relational Database Service (RDS) was available in the chosen AWS region (Ireland). The database was fully managed by AWS, and was easily scalable for different purposes.

3.3 The cloud structure

For Tampere public journey planner services, the final solution consisted of four different containers (Figure 13). These were the frontend container, two different algorithm containers, and a container for memory caching. Amazon RDS was used in storing the routing and timetable data. One EC2-instance was used to host the altitude-data for the cycling and walking trip planner, and one for hosting the data loader tool for updating the journey planner data. For every container, a Fargate service was created with a task to run the container. Should the container become unhealthy, the service automatically creates a new task and starts another container. An application load balancer (ALB) was configured to route traffic to the frontend containers, and the encryption keys and their certificates were set to the load balancer to enable encrypted connections for the end-users.

The containers were deployed into a virtual private cloud (VPC), which had an IP-address range of a local subnet. The communication between containers was handled by specifying security groups for each service. Transport Control Protocol (TCP) communication with each necessary container network ports was allowed separately for each service. The only cloud resource to also have a publicly reachable network interface with public IP-address was the ALB, as its default TCP-ports for HTTP-traffic were configured to allow connections with machines in the public Internet.

The RDS instance was configured to allow connections only from inside the subnet. The configuring of the database and its users was automated using Structured Query Language (SQL) and shell scripts, as well as uploading the actual data to the database. For configuring the database and uploading the data to it, an EC2 instance was used from which the database was connected to.

The frontend had a feature in which the end-user can send a route suggestion via email. Amazon Simple Email Service (SES) was taken into use to as a email sender with a Simple Mail Transfer Protocol (SMTP) interface. An email transfer agent (Exim) was installed in the frontend container, and it was configured to use SES with SMTP credentials.

The existing continuous integration (CI) solution was sufficient for the new environment, and it was not running on aging hardware, so it was decided to be left in use. Jenkins automation server was configured to use local CI-slave machines in

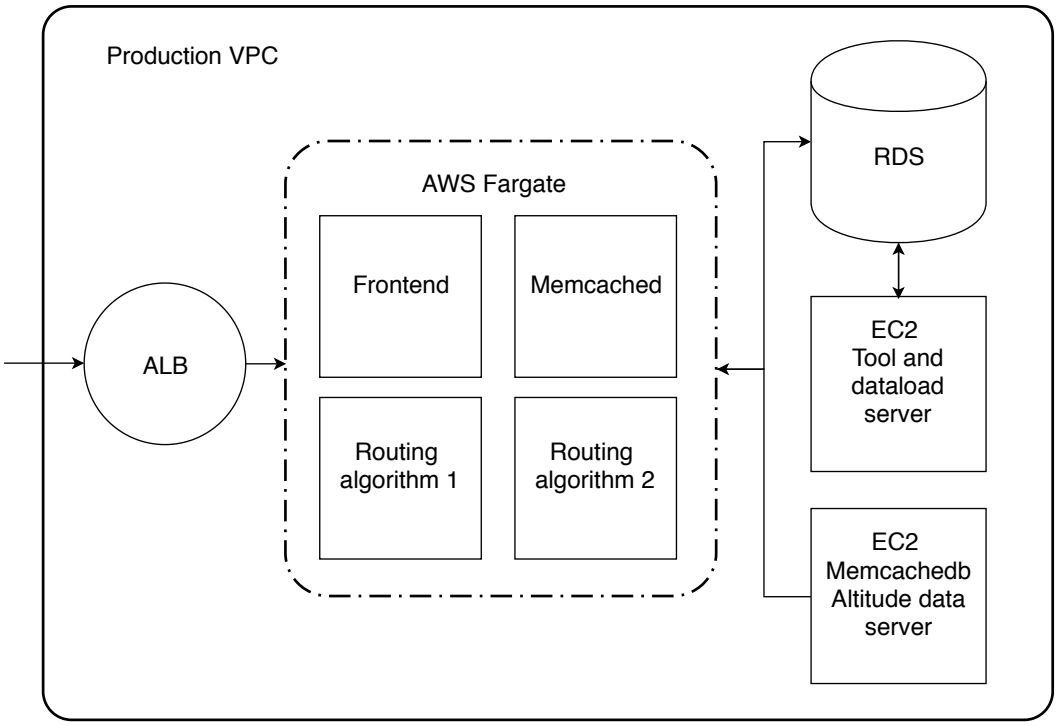

Figure 13: A model diagram of production environment.

the old server environment to build container images, transfer them to container registry and deploy containers based on the images on the cloud platform. The container orchestrator was able to perform a rolling update, which means that a container with a new image is deployed before shutting down the existing container, and the new instance is taken in to use gradually with minimal downtime.

The containers were configure to read essential log files and pass them to the host's standard output stream, so that debugging would be easier for the maintainers. In Amazon Fargate, the host instances of the containers had a log driver that passed the stream further Amazon Cloudwatch, a cloud service that stores the logs from other services. The Cloudwatch is also able to send notifications after the specified amount of terms appear in the log data. For example, if the routing algorithm containers have not been able to process routing requests and produce error messages to the log data, the system maintainer receives a notification mail. Fargate also sends data about CPU and memory usage to the Cloudwatch, from which also a trigger threshold for the sending of notification mail can be configured. The logs and, for example, graphs for the performance data can also be viewed in the AWS management console, which makes Cloudwatch usable for quickly viewing the status of services. The ALB was also configured to create service access logs to Amazon's Simple Storage Service.

3.4 Automating infrastructure with Terraform

AWS offers public REST API, working over HTTP, for customers to use their resources. Based on the API, an open source tool known as AWS-CLI has been developed to ease the control of resources. The AWS-CLI is written in Python and establishes the interface in either Linux terminal or Windows Power-Shell. The AWS-CLI abstracts out the using of HTTP request methods into giving simple one-line commands to create, delete, and modify resources. After giving a command, text response is received. However, while the AWS-CLI allows the nearly complete control of the resources, Terraform adds another layer of abstraction by enabling the user to describe the whole cloud environment in a configuration file. Terraform is an open-source software written in GO, a programming language created by Google. It supports multiple cloud service providers, and therefore works as a complete cloud infrastructure manager. Its basic functions include obtaining the status of the resources, comparing them with the properties described in configuration files, and creating a plan to update the resources. As AWS resources are mainly charged by the time they are used, the automated creating and deleting of the resources makes a cost-effective building and testing of cloud-computing environment possible.

RDS is the single most expensive AWS resource that was used in the project. Also the workload of setting it up using the AWS management console in web browser took time and required selecting many specific options. Therefore automating its creation and deletion with Terraform was the first logical step towards describing the whole infrastructure with code. After creating the resource in AWS management console, it was possible to import the details of the resource to Terraform which eased describing it in a configuration file (Figure 14). The details of a database-instance in AWS include the database engine, storage type, and allocated storage. However, using Terraform required understanding the function of different computing resources in AWS. AWS management console supports first-time-users by giving instructions about how to use the resources (Figure 15), and was therefore a better approach in building the environment for the first time.

```
1 resource "aws_db_instance" "production" {
2   name                 = "mysqs-rds"
3   engine               = "mysql"
4   engine_version       = "5.5.59"
5   instance_class       = "db.t2.medium"
6   multi_az             = "true"
7   storage_type         = "gp2"
8   allocated_storage    = 100
9   ...
```

Figure 14: Example of a Terraform file describing an RDS-instance.

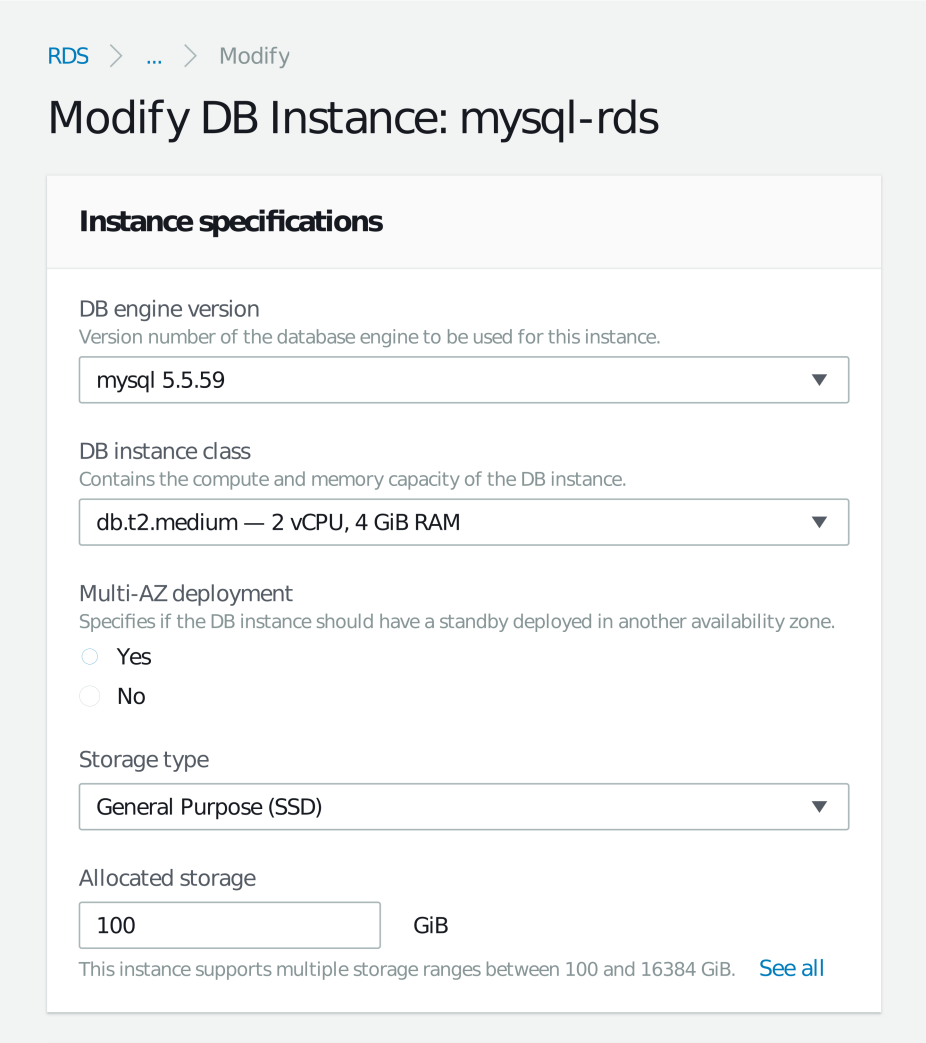

Figure 15: Example of an AWS management console view, while modifying an RDS-resource.

3.5 Load testing

For the application to be ready to deploy on production, it was load tested. The program used for stress testing was an open-source performance test application called Apache JMeter (Apache, 2019). It allowed the system to be load tested by making HTTP GET-requests to the frontend container, or by making queries for routing algorithms and database via plain TCP. Different test setups for the routing algorithm of cycling and walking journey planner were made including various amount of simulated users. Every user requested routes multiple times between randomly selected coordinates in a city (Figure 16).

The delays between the requests of a single user were randomly generated to simulate a real situation. The delays were values from a Gaussian distribution with a mean value of five seconds and a standard deviation of two seconds. The time between the opening and closing of a TCP connection between the test application

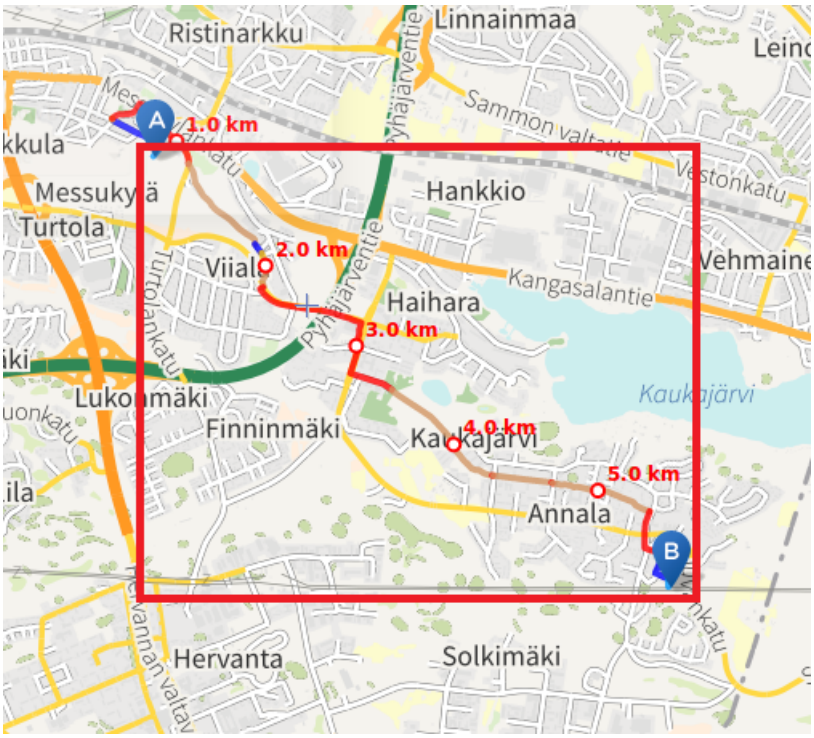

Figure 16: Test area for routing and an example route.

host server and the algorithm container was measured. The maximum number of concurrent users to simulate in the tests was limited to the amount that the algorithm instance inside the container was still able to serve without declining connections for an excessive amount of reserved ports. Thus, in the first test setup there were 300 concurrent users, each making 40 routing requests. A 30-second ramp-up period was defined in which the amount of concurrent users grew linearly to the maximum amount. Three tests with equal load parameters were performed for different amount of container instances, for which the requests were routed through an inner load balancer. A load test to verify the automatic scaling feature of the container platform was performed with 350 simulated users, everyone making 140 requests towards the load balancer with equally distributed delays as in the previous tests. The ramp-up period was set as 120 seconds, and an automatic scaling policy was added for the service. An average CPU utilization limit of 10 percent for the service was defined to trigger an alarm, after which the Amazon Fargate would scale the number of containers up to a necessary value to restore the average CPU utilization of the service to remain below the threshold level.

4 Results of load testing

With one container registered in the load balancer target group, the average response time for routing requests was over 60 milliseconds (Figure 17), whereas with three or five container instances, the average response times stayed nominally between 20 and 40 milliseconds (Figures 18 and 19). The average response time seemed therefore to be inversely proportional to the amount of active container instances. However, after a certain limit, deploying more containers did not lower the average response time anymore, which in this case happened after deploying five container instances. As the delay between the requests of a single user was random, the average response times did also vary with the same amount of users making the requests.

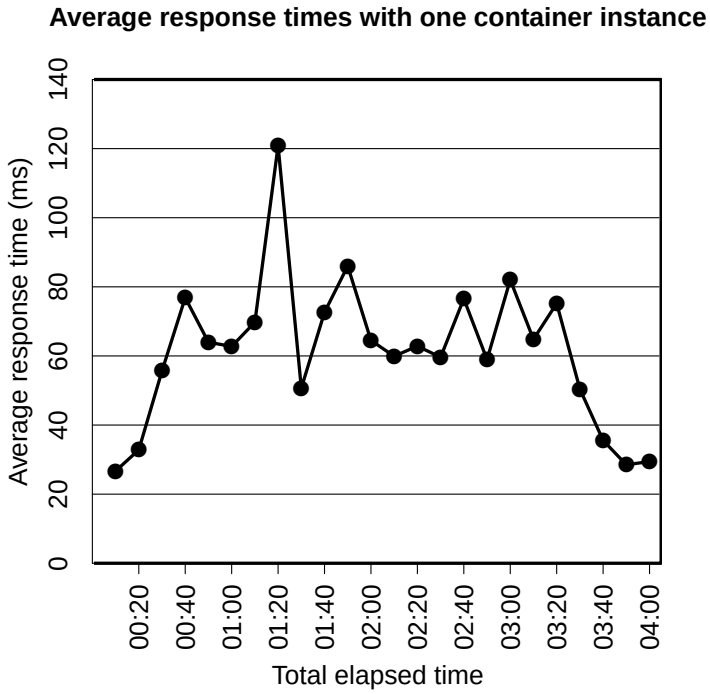

Figure 17: Results from a load test made for a single routing algorithm container.

In the automatic scaling test, the average response time increased approximately 1500 ms during the ramp up period, and after six additional containers had been started and registered to the load balancer target group, the average response time returned to its normal value which was around 30 ms (Figure 20). Therefore, the routing service had a period of slow response (approximately 5 minutes) until the normal state was restored. The busy routing algorithm containers also rejected connections and produced some errors on the client side, with total number of 37. The total number of requests in the test was 49000, out of which the 37 errors would have appeared as an unreachable routing service for some users.

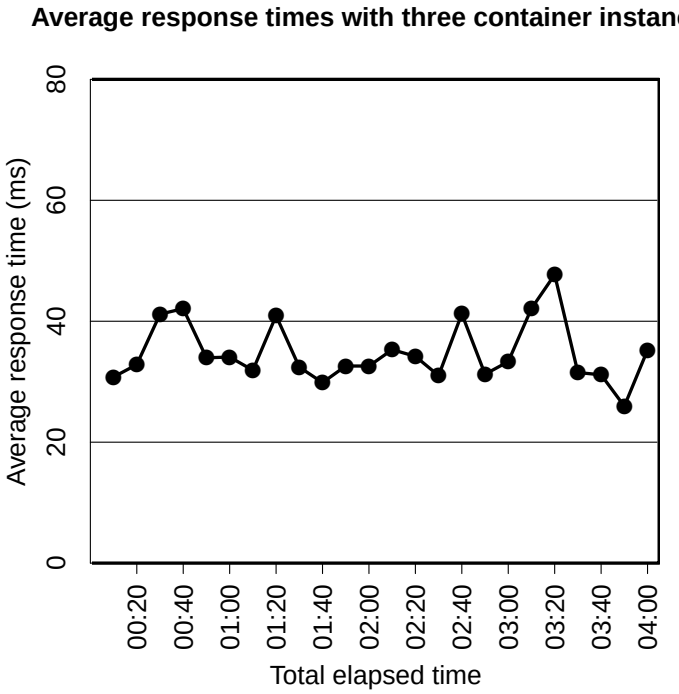

Figure 18: Results from a load test made by distributing routing requests evenly between three routing algorithm containers.

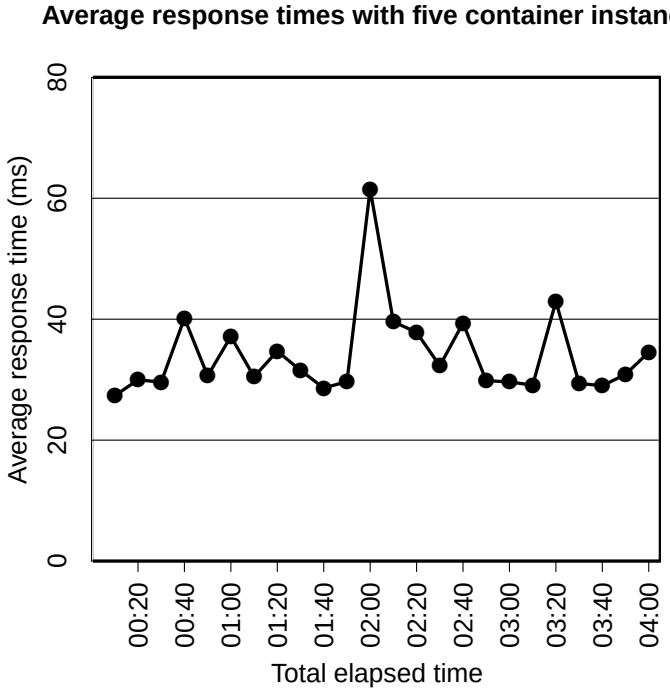

Figure 19: Results from a load test made by distributing routing requests evenly between five routing algorithm containers.

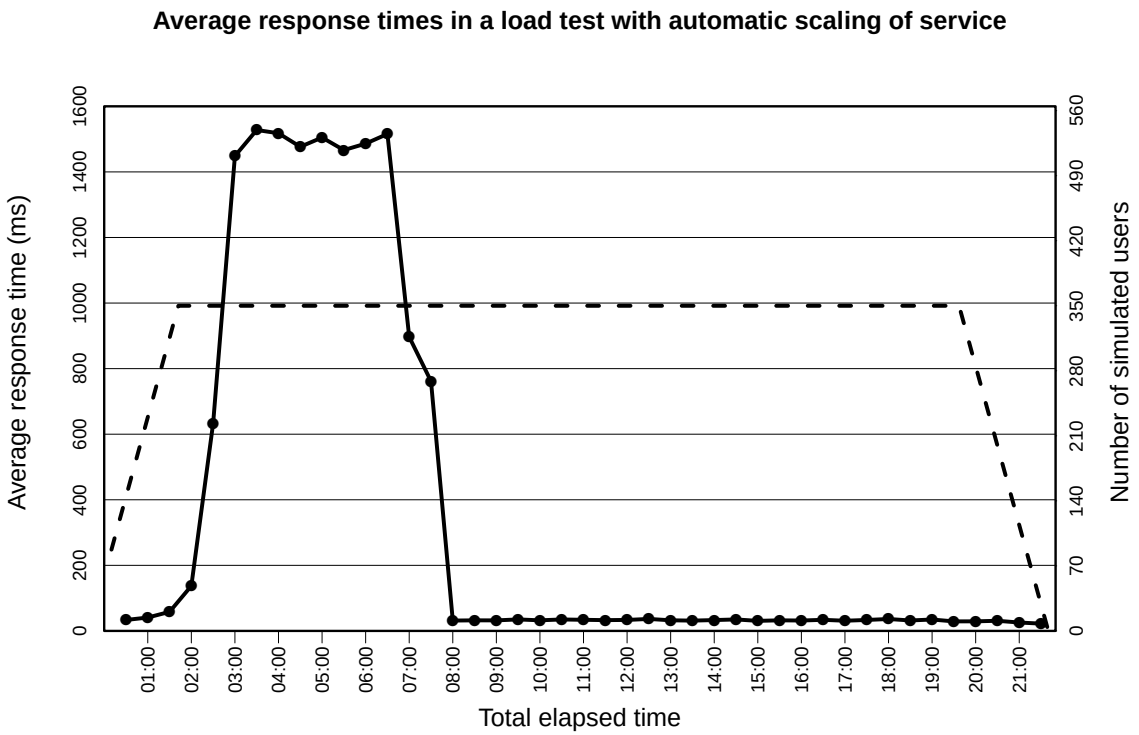

Figure 20: Results from a load test made for a scalable routing algorithm service. The solid line represents average response times during the test run, and the dashed line the number of simulated users.

5 Discussion and conclusions

In this chapter, the selected methods are evaluated from the virtualization technique and cloud platform to the software development process model. The challenges during the migration project are discussed and an attempt is made to offer patterns for similar migration projects in the future. Also the steps for taking the new cloud environment into use as the production environment are described and tasks that still need to be done in the future.

5.1 The most notable challenges

The most difficult single component to adapt to the cloud environment was the journey planner data loader. The code needed for the data loader was in Debian-packages. Setting it up in new environment required transferring the packages to cloud, installing them on EC2 instance, and making the data load process to point to the new database. The data loader did not tolerate any faults in its environment and it demanded many specific configurations in both the PHP-engine and database service to get the data load process through. However, the data loader works as a part of the data quality assurance. By stopping its progress when facing an error in the data, the data loader prevents the insertion of corrupted data into the database. In order to perform data loads without disrupting the service, there were two databases in the RDS that the journey planner could use. New data is loaded to the inactive database, and the journey planner is switched to use that database after the data load has completed. Configurations that were deprecated in the old environment and left in the code prevented the change of the database from the GUI of the data load tool in the cloud environment. However, after thorough debugging, the misconfiguration was detected and the change of the database was successful.

The email feature of the frontend container was challenging to implement in the cloud environment, as the SES does not support sending mails with an arbitrary address as a sender address. Instead, it required sender addresses to be registered, which was not possible for addresses that could not receive mail (*no-reply-addresses*). Furthermore, as the customer controlled the address domain that was used in the sender's address in the route suggestions mails, the address was changed to facilitate the migration process.

The migrated system was connected in several ways to a vehicle tracking system which had components that did not exist in the original server cluster. These were two Windows-servers that were not managed by the team executing the migration. They were running, for example, a vehicle tracking application and a related database. The tracking system provided the journey planner bus locations, arrival time estimates and live disruption info. The servers did not exist in the same subnet as the servers under migration, but they were discoverable internally through network configurations that were created by the company's network team. The Windows servers also had APIs facing public network, from which the cloud environment was first intended to receive real-time information. However, the traffic disruption information was shared inside the company network through a private API which delivered the information

in different XML format than the public API. Therefore, a change was made to share the old XML format publicly, and new configurations needed to be made to the Windows server hosting the API. Also, a previously configured proxy was taken into use in the old environment to deliver the other real-time information to the cloud environment.

5.2 Evaluation of the used methods

The suitability of the container virtualization technique for data processing has certain limits. One of them is the stateless container image, which means that the containers are designed to run without an administrator directly interacting with them, or their automatic platform. The administrator has to rely on the promised performance of the cloud service, and the cloud providers rarely report the details of their used hardware. There is also no guarantee about the amount and the quality of software abstraction layers between the cloud platform and the host operating system. However, as an automated container platform such as Amazon Fargate manages the replication and state monitoring of the containers and the possible updates on the container platform (for kernel or container engine), the trade-off in changing to Fargate-like service favors those, who value time over complete control. This is a characteristic of the PaaS-model, as described in the background-section. The boundaries between infrastructure and platform are blurring out and, for example, the Amazon Fargate may be considered more of a PaaS rather than IaaS, since the computing resources are almost entirely controlled by the service provider. The automatic provision of the resources and state monitoring steers the focus of development only to the container images and the software inside them.

Performance-wise, using Amazon Fargate as a container orchestrator gave promising results. Automatic scale-up in five minutes ensures the function of business-critical components during sudden surge of users. As the automatic scaling feature did not itself bring any additional costs to the cloud service, setting it up seems useful for any web-developer expecting growth for their product. However, it is necessary to have a stable and deterministic software before setting up automatic scaling, since a software misbehavior might appear as a high traffic load, thus creating unnecessary costs after the accidental service scale-up. Additionally, a malicious user could be able to utilize the automatic scale-up and intentionally generate costs, which should be considered by improving the detection of misuse. AWS provides web application firewall, which has features for detecting and avoiding distributed denial of service attacks. However, the domain name of the ALB is automatically configured to point to different IP addresses by default, which lowers the risk for traffic load attack.

As Sommerville states in his book, the requirements of the software must be let to evolve during a component-based software development process. In such process, the software is made by reusing components that are functional higher-level abstractions. Such components in the migration project were the containers, ECS tasks and services, and the container orchestrator. The created composition included also VPC, Route 53, security groups, EC2-instances, RDS, SES, and other cloud resources. Some requirement changes had to be made during the project, as some

of the AWS components were not completely compatible with the solutions in the original environment. For example, the database engine was differed from what the old database had, and therefore loading data to RDS using exports from the old database prevented the database in cloud to be completely restorable from backups. This causes the data load process to be renewed in the future, and currently, in case of a database corruption, the data has to be loaded to the database again by using the data loader. Throughout the project the advantage of using the containers in other cloud platform in the future remained, as the adaptation that had to be made to the existing application software were minor. Therefore, the component based software development process turned out to be applicable in the process of migrating data processing software to cloud platform.

5.3 Patterns for future migration projects

Considering the structure of the old software environment, one of the key aspects regarding the workload of the migration to cloud environment is the software configuration. For many pieces of third-party software, that is going to be installed on the new environment, there are custom configurations made during the service lifetime to optimize the server environment for the self-developed software. The two main categories of third-party applications in the project were the webserver software (Apache2 and NginX) and database software (MySQL and Postgres). The self-developed software has eventually evolved to work in the optimized environment, and may suffer from the tiniest differences in the configuration of the third-party software in the new environment. In the old environment, there were changes made in, for example, the limit of the data size that can be processed by the journey planner data loader and stored to the database to prevent the server disk space from filling up. Some changes had also been made to the default configuration of third-party software to intentionally create obfuscation, which is a standard and well-known procedure to increase security in the server environment by making it harder for a malicious user to do exploitation. Nevertheless, a good configuration management with version control existed in the old environment, which eased the tracking of configuration changes made and their purpose.

The key challenge in the project was therefore related to configuring software in the new environment. As explained in the background section, container virtualization was chosen in order to make the most of the cloud environment and decrease continuing costs of operation. Also for the same reasons, database service was taken into use. Adopting the components in the old environment to use PaaS required leaving out the old service discovery and database configurations. In practice, this meant making more changes in the configuration files of both the self-developed and third-party software. The re-configuration phase of the migration was notably the part in which help was needed the most from the team members that were most experienced in working with the old software. The final solution was a compromise between making the cloud platform resemble the old environment as much as possible, and adapting the existing software to work in the cloud platform.

From a project management perspective, few aspects stood out having the greatest

effect on reaching the desired end result. First of all, constant knowledge transfer between team members remained important throughout the migration. People specializing in different areas of software development have different background and views on how to execute the migration steps, and keeping up the discussion to gather the composition of the best ideas was a fruitful approach, but also took time and effort. Secondly, while every team member possesses a knowledge of a specific part of the software, it is beneficial that everyone is committed to exploring the areas of the software that are unknown to them. Finally, the courage to try out different approaches and fail is important in order to move the project forward and put the lean-first approach into practice, as stated by Poppendieck and Cusumano.

5.4 Learning outcomes of the project and future work

Migrating a web service to cloud using containers required using many different services from the cloud service provider. Therefore, the knowledge of all of the cloud service provider's tools is essential in a cloud migration project. This knowledge was improved while carrying out the project through hands-on learning and studying the documentation.

Regarding DevOps, the migration itself was an attempt to automate as many maintenance tasks in the application environment as possible, but more work is required to optimize build processes and minimize the manual work related to continuous integration. For example, quality assurance should be implemented for the Docker images and test automation is needed to improve the reliability of the application.

Site monitoring should be improved to include alerts for service downtime and dysfunction. This is attainable with Cloudwatch, as it is possible to configure multiple types of alerts for the platform regarding container health and traffic load. Some of the alerts configured while using the old environment were still valid after changing to the new production environment, as they include tests made against the domain-names of the service, which were configured to point to the ALB in the last step of the transfer process. In the end, to use the new environment as a production environment, the DNS service provider for the city of Tampere was requested to redirect the DNS queries made for the service domain names to the domain name of the ALB. The transfer process went out smoothly, as shown in Figure 21 and only minor flaws were detected and repaired in the system during two weeks after the service load had stabilized.

The project generated both measurable and unmeasurable benefits and value. However, the difficulty to measure the generated value of a certain project outcome should not diminish its importance by default. To sum up, not only did the project evolve the whole run-time environment of the software, but also brought in data of current cloud performance and significantly increased the cloud experience of the operations team.

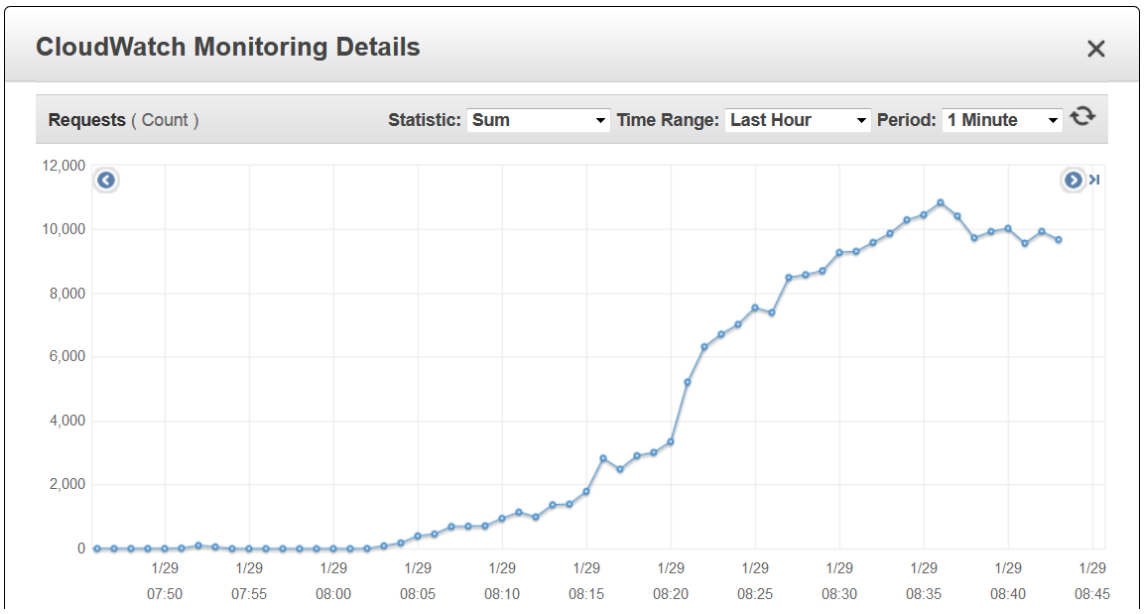

Figure 21: The increase in the number of requests made to the ALB after the DNS change of tampere.fi subdomains (aikataulut, vanharepa, lissu, kevytliikenne, and linjakartta), which completed the transfer process at 8 a.m. UTC on January 29, 2019. The time to live value for the DNS records was set to 30 minutes in advance, which resulted in a half an hour ramp-up period.

References

Amazon (2018). Structure of physical cloud regions in AWS. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html (Accessed: 2018-11-29).

Amazon (2019). Aws certification. https://aws.amazon.com/certification/ (Accessed: 2019-02-16).

Apache (2019). Apache JMeter. https://jmeter.apache.org/ (Accessed: 2019-02-09).

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia (2010). A view of cloud computing. *Communications of the the Association for Computing Machinery (ACM) 53*(4), 50–58.

Balalaie, A., A. Heydarnoori, and P. Jamshidi (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software 33*(3), 42–52.

Ben-Kiki, O., C. Evans, and B. Ingerson (2005). YAML ain't markup language (YAML™) version 1.1. *yaml.org, Tech. Rep.* https://yaml.org/spec/cvs/spec.pdf, (Accessed: 2019-01-05).

Beyer, B., C. Jones, J. Petoff, and N. R. Murphy (2016). *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media, Inc. https://landing.google.com/sre/sre-book/toc/index.html (Accessed: 2019-01-12).

Bort, J. (2014). IBM Scores Another Win Over Amazon's Cloud Thanks To SAP. https://www.businessinsider.com/ibm-and-sap-score-win-over-amazon-2014-10 (Accessed: 2019-01-07).

Bray, T., J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan (2006). Extensible markup language (XML) 1.1 (second edition). *The World Wide Web Consortium (W3C) recommendation.* https://www.w3.org/TR/xml11/ (Accessed: 2019-01-06).

Brunnert, A., A. van Hoorn, F. Willnecker, A. Danciu, W. Hasselbring, C. Heger, N. Herbst, P. Jamshidi, R. Jung, J. von Kistowski, et al. (2015). Performance-oriented devops: A research agenda. *Technical report SPEC-RG-2015-01, SPEC Research Group—DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), arXiv preprint arXiv:1508.04752*.

Chauhan, M. A. and M. A. Babar (2011). Migrating service-oriented system to cloud computing: An experience report. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, Washington, USA, pp. 404–411.

Dean, J. (2009). Designs, lessons and advice from building large distributed systems. In *Keynote from International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky Resort, Big Sky, Montana, USA.

Fox, G. C., V. Ishakian, V. Muthusamy, and A. Slominski (2017). Status of serverless computing and function-as-a-service (FaaS) in industry and research. *arXiv preprint arXiv:1708.08028*.

Google (2019). GCP Region in Finland. `https://cloud.google.com/about/locations/finland/` (Accessed: 2019-02-03).

Hossain, M. S. and G. Muhammad (2016). Cloud-assisted industrial internet of things (IIoT)–enabled framework for health monitoring. *Computer Networks 101*, 192–202.

Jenkins (2019). Build great things at any scale. `https://jenkins.io/` (Accessed: 2019-02-03).

Joy, A. M. (2015, March). Performance comparison between Linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, Ghaziabad, India, pp. 342–346.

Khajeh-Hosseini, A., D. Greenwood, and I. Sommerville (2010). Cloud migration: A case study of migrating an enterprise it system to IaaS. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, Miami, Florida, USA, pp. 450–457.

Mell, P., T. Grance, et al. (2011). The NIST (National Institute of Standards and Technology, USA) definition of cloud computing. `http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf` (Accessed: 2018-01-06).

Panettieri, J. (2017). Cloud market share 2017. *ChannelE2E*. `https://www.channele2e.com/channel-partners/csps/cloud-market-share-2017-amazon-microsoft-ibm-google/` (Accessed: 2019-01-05).

Poppendieck, M. and M. A. Cusumano (2012). Lean software development: A tutorial. *IEEE Software 29*(5), 26–32.

SAP (2019). What is SAP HANA. `https://www.sap.com/products/hana.html` (Accessed: 2019-01-07).

Sommerville, I. (2013). *Software Engineering: Pearson New International Edition*. Pearson Education Limited.

Sutherland, J. and K. Schwaber (2017). The scrum guide. *The definitive guide to scrum: The rules of the game*. `https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf` (Accessed 2019-01-05).

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30*(3), 202–210.

The City of Tampere (2019). Nysse - Tampere regional transport. https://joukkoliikenne.tampere.fi/en/ (Accessed: 2019-01-12).

The City of Tampere, CGI, National Land Survey of Finland (2019a). Tampere regional transport journey planner. https://vanharepa.tampere.fi/en (Accessed: 2018-12-02).

The City of Tampere, CGI, National Land Survey of Finland (2019b). Tampere regional transport traffic monitor. https://lissu.tampere.fi/?stop=&lang=en (Accessed: 2019-02-16).

Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization*. James Turnbull.

Walter, C. (2005). Kryder's law. *Scientific American 293*(2), 32–33.

Zheng, S. (2017). Developing a cloud-based solution for cardiographic data collection and processing. Master's thesis, Aalto University, Espoo, Finland.