Implementing a container-based build environment: a case study

Paavo Pärssinen

School of Science

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

Espoo 20.12.2019

Supervisor

Prof. Casper Lassenius

Advisor

M.Sc. Risto-Matti Vuonnala

Aalto University
School of Science

Aalto University
School of Science

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

Author Paavo Pärssinen

Title Implementing a container-based build environment: a case study

Degree programme Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Casper Lassenius

Advisor M.Sc. Risto-Matti Vuonnala

Date 20.12.2019 **Number of pages** 63+1 **Language** English

Abstract

Containers have become a widely adopted way to package and distribute software. Their portability, ease of use and small computational overhead are some of the main features driving their popularity, as well as their adoption into modern cloud architectures. Some developers have also come to realize the advantages of using containers as a way to package dependencies and tools, easing the set-up and migration of development environments across computers.

In this case study, we studied the process of moving an entire software department to a workflow, where the default way of building and testing the project software is in a container. The department consists of nearly 200 developers developing Layer 1 software for 5G radio base stations. Two interview rounds were conducted, of which the first analyzed the initial state of the workflow in the department. The second interview round was conducted after taking a containerized build environment into use.

In addition, a literature review was conducted on the technologies related to the solution, as well as on previous studies related to the subject. The literature review lead us to choose Docker for our container engine, because of its adoption rate and positive user experiences. We found few studies relating to the containerization of a build system, but multiple blog posts and instructions on the subject were found.

The first interview revealed that most developers are open to adopting new ways of working and thought positively about the adoption of containers in our use case. Ease of use was an important feature to the developers, as well as proper instructions for its usage. The solution came to include a tool for interacting with the container and ease its usage for even developers not familiar with containers. To measure the success of our solution, we asked for feedback from our interviewees and found that mainly people were satisfied with it. One pain point, however, was found to be the size of our image, which can create problems in an environment with a slow network connection. According to our data, our solution makes the development and testing of our build environment more effective, saving approximately 1900 hours of developer time per annum.

Keywords Software build, containers, Docker, CI

Aalto University
School of Science

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

Tekijä Paavo Pärssinen

Työn nimi Kontainerisoidun build-ympäristön toteuttaminen: Tapaustutkimus

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja Prof. Casper Lassenius

Työn ohjaaja DI Risto-Matti Vuonnala

Päivämäärä 20.12.2019 **Sivumäärä** 63+1 **Kieli** Englanti

Tiivistelmä

Konttiteknologiasta on tullut yleinen tapa pakata ja jakaa ohjelmistoja. Konttien siirrettävyys, helppokäyttöisyys ja korkea suorituskyky ovat niiden tärkeimpiä syitä niiden suosion takana, kuten myös niiden laaja käytettöönotto pilviarkkitehtuureissa. Jotkut ohjelmistokehittäjän ovat huomanneet konttien soveltuvan myös työkalujen ja riippuvuuksien pakkaamiseen, joka toimii tapana helpottaa kehitys- ja käännösympäristön jakamista sekä pystyttämistä.

Tutkimme tässä tapaustutkimuksessa ohjelmistoprojektin siirtymistä työmalliin, jossa pääasiallinen kehitys- ja käännösympäristöön on konttipohjainen. Projektiin kuuluu noin 200 ohjelmistokehittäjää, jotka työskentelevät 5G-verkon tukiaseman fyysisen kerroksen ohjelmistojen parissa. Diplomityön aikana toteutimme kaksi haastattelukierrosta, joista ensimmäisessä tutkimme osaston lähtötilannetta. Toinen haastattelukierros toteutettiin siirryttyämme konttipohjaiseen työmalliin.

Toteutimme lisäksi kirjallisuuskatsauksen, jossa tutkimme aiheeseen liittyviä teknologioita sekä aikaisempaa tutkimusta aiheesta. Katsauksen pohjalta päädyimme käyttämään Dockeria konttiteknologianamme sen laajan käyttöasteen ja hyvän vastaanoton takia. Aikaisempaa tutkimusta käännösympäristön pakkaamisesta kontteihin löytyi vain vähän, mutta käytimme niiden sijaan blogipostauksia ja kehittäjien kirjoittamia oppaita, joita löytyi paljon.

Ensimmäinen haastattelu vahvisti, että kehittäjät ovat valmiita hyväksymään uusia työskentelytapoja, ja suhtautuivat myönteisesti konttiteknologioiden käyttöönottoon. Tärkeimpiä asioita haastateltavien mielestä olivat helppokäyttöisyys, sekä tarkkojen käyttöohjeiden jakaminen. Ratkaisumme tuli sisältämään tarkoin määritellyn työkalun, joka käynnistää Docker-kontin käyttäjän puolesta, helpottaen niiden työtaakkaa joille Dockerin käyttäminen ei ole tuttua. Saadaksemme arvioitua ratkaisumme onnistumista, kysyimme haastateltavilta palautetta sen käytöstä. Pääasiassa vastaanotto oli myönteinen. Suurimmaksi ongelmakohdaksi muodostui jaettavan Docker-levykuvan suuri koko, joka aiheuttaa ongelmia verkkoyhteyden ollessa hidas. Tutkimuksemme mukaan ratkaisumme tehostaa käännösympäristömme kehitystä ja testausta säästäen yli 1900 työtuntia vuodessa.

Avainsanat Software build, ohjelmistokontit, Docker, CI

Preface

First and foremost I want to thank my supervisor Casper Lassenius and my advisor Risto-Matti Vuonnala for their excellent guidance and help throughout the thesis. Your input greatly helped me when I was unsure of how to proceed. I also want to thank my fellow thesis workers for all the peer support during this year.

I also want to express my gratitude towards my family and friends. Thank you for always supporting me and encouraging me to choose my own path. I especially want to thank my dear wife Suvi, for always being there through the best and the worst of times. Also, a big thank you to all the members of Syndikaatti. The years at the university would have been a lot less interesting without your company.

Otaniemi, 20.12.2019

Paavo Pärssinen

Contents

Abbreviations

3G	3rd Generation
4G	4th Generation
5G	5th Generation
API	Application Programming Interface
CD	Continuous Delivery
cgroup	Control Group
CI	Continuous Integration
CP	Control Plane
GCC	GNU Compiler Collection
GID	Group ID
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
HDPA	High Data Packet Access
IDE	Integrated Development Environment
I/O	Input/Output
IoT	Internet of Things
L1	Layer 1
LXC	Linux Container
LTE	Long Term Evolution
MTC	Machine-Type Communications
NFS	Network File System
OFDM	Orthogonal Frequency-Division Multiplexing
OS	Operating System
PaaS	Platform as a Service
PBCH	Physical Broadcast Channel
PDCCH	Physical Downlink Control Channel
PDSCH	Physical Downlink Shared Channel
PHY	Physical layer protocol
PRACH	Physical Random Access Channel
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
RHEL	Red Hat Enterprise Linux
SoC	System on a chip
SSH	Secure Shell
SVN	Subversion
UE	User Equipment
UFS	Union File System
UID	User ID
UP	User Plane
URL	Uniform Resource Locator
VCS	Version Control System
VM	Virtual Machine
VNC	Virtual Network Computing
WSL	Windows Subsystem for Linux
YUM	Yellowdog Update, Modified

1 Introduction

Building software often requires a very specific build environment. The build environment may, for example, require a specific operating system, software package dependencies, mounted file systems and presence of certain software repositories. Additional requirements may also be needed if Continuous Integration is part of the development process (Meyer 2014). The requirements become a problem when such an environment must be usable on a multitude of different platforms. Platforms may include for example cloud servers, bare-metal servers, Kubernetes clusters, developer's laptops, virtual machines, and many others. The underlying OS on the platform may also vary, from different Linux distributions to macOS and Windows. The configuration of the requirements is a large workload. The maintenance of these environments also poses a problem; when the software project matures over time, the build environment may also need to be changed. This leads to the already deployed build environments needing an update to be able to build the software. Such an update, if done completely manually, can be a very large workload.

1.1 Research Problem

The aim of this thesis is to find a tool for packaging a build environment and a way to distribute the packaged build environment. We propose the use of containers and Docker for this, and during this thesis, we discuss the selection of this technology and evaluate the use of containerization as a means to counter the previously mentioned problem and allow easier deployment and maintenance. Containerization is a technology used for running software in a contained environment and it also functions as a more light-weight alternative to virtual machines. Whereas a virtual machine virtualizes the kernel and operating system, containers only virtualize the operating system and provide close to native performance. Docker is a popular container manager system, it is available for all major platforms and can be used to orchestrate, control and distribute containers. (Merkel 2014).

During this thesis, we create a Docker image that contains all the build environment requirements, distribute it in our internal Docker registry and keep it updated. We will evaluate the solution with quantitative and qualitative data, that will be recorded prior to taking the contained build environment into use as well as after the fact. The thesis will also explore the challenges and limitations of using Docker as a build environment in our use-case. For example, we will discuss some security aspects of granting containers the needed privileges. Another example is the use case where a developer wants to set up the build environment on a Windows laptop. Docker for Windows uses a workaround to run Linux containers; it sets up a Hyper-V Linux virtual machine, on which it runs the Docker containers. How will this affect the performance and functionality of the build process in the container? We will also discuss the challenges in distributing the Docker image, keeping the docker image version and software version synchronized and doing everything in a user-friendly fashion.

Solving the research problem will allow our software department to work more

efficiently when it comes to the development and testing of the build environment. It will also allow flexibility, as developers no longer need to rely on IT provided pre-configured development machines. As little research is previously done on the subject of using containers to maintain and distribute a computing environment, this thesis will provide valuable data to the scientific community on the practicality of this approach.

1.2 Structure

The thesis is structured as follows. Section 2 discusses related technologies, to provide the reader with an understanding of all the key technologies, terms and ideas. The section includes background on technologies such as Virtualization and Docker, and on basic software development practices such as software building and Continuous Integration (CI). Related work is also discussed.

Section 3 discusses the case at hand, meaning the company and department the thesis was conducted in and the organization of the department and the technology used in the department. Section 4 discusses the research methods chosen for the study, as well as the way the research methods requirements are met. For example, details on a conducted interview and other ways data were gathered are explained here.

In Section 5 focuses on the analysis of the main problem we are solving in this thesis. This includes a detailed explanation of the initial situation of the department, which contains the results of the first interview. Section 6 discusses our approach to solving the problems introduced in the former section, and contains a detailed explanation of our solution.

Section 7 shows the results of our solution, including data gathered from interviews and timed tests. A cost-benefit analysis is also included. Finally, Section 8 summarizes the thesis and discusses the possible future development ideas based on our work.

2 Background

This section discusses the relevant background related to the subject of the thesis. These background subjects include software development and virtualization technologies, which are both discussed in their respective sections 2.1 and 2.2. We also looked for work related to the idea of a containerized build environment, and discuss our findings in Section 2.3.

We begin by discussing software development and its modern practices. In Section 2.1.1 we discuss a common step in multiple software development workflows know as software building. We continue on the evolution of software build tools in Section 2.1.2, where build automation and test automation are discussed. Continuing on the same line of thought, Section 2.1.3 focuses on a modern software development practice known as continuous integration (CI). After this, we focus briefly on the subject of reproducibility in software development in Section 2.1.4.

Section 2.2 discusses different virtualization technologies. We begin with the older technology known as hypervisor-based technology in Section 2.2.1 and continue with the more recent technology known as container-based virtualization in Section 2.2.2. Finally, we discuss the most popular container technology, Docker, in Section 2.2.3.

2.1 Software Development

Software development is the process in which software is created. It features steps such as designing, programming, testing, documenting and maintaining the software and its components. To support these steps, multiple software development methodologies have been invented, and as information technology is an ever-evolving field, the preferred methodologies have changed multiple times and will change multiple times in the future. This chapter discusses some software development practices that are relevant to our study.

2.1.1 Software build process

Software development often consists of a cycle, where developers write code in a programming language, compile the code to executable format, run and test the program and if needed go back to the first step. This process can be called the build process of the software, and the result of a build is often a numbered release of the software. Depending on the technologies and programming languages used, the build process may include the compilation of source code files, packaging of the compiled files, production of an installer or even database creation. Ideally, the build process also includes a thorough test suite designed for verifying the functionality of the software (Meyer 2014). Some programming languages, like Python and JavaScript, do not require the compilation step, but other steps like the ones mentioned previously may take their place. We focus on the build process of C/C++ code, in which the compilation of the program is a vital step.

To compile a software's code, the computer handling the compilation must meet some requirements. One of the requirements is the compiler for the programming

language in question. For example, the GNU Compiler Collection (*GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF)* 2019) can be used as a compiler for C/C++ programming language code. Another requirement for the build environment are the libraries needed; for example in the case of Python, multiple packages maintained by other developers may need to be installed in order to build the project. In the case of C/C++ it may mean that the inclusion of related code is required, for example, if the hardware for which the software is designed for has pre-made libraries for its use.

Other possible build environment requirements include, but are not limited to:

- Dynamic analysis tools, such as Valgrind that is used for detecting bugs in memory management and threading (*Valgrind Home* 2019)
- Static code analysis tools, like Coverity for finding defects and security vulnerabilities (*Coverity Scan - Static Analysis* 2019)
- Tools for generating documentation, such as Doxygen (*Doxygen: Main Page* 2019)
- Build management tools, like make and CMake (*CMake* 2019)

The environment must also have a tool for fetching the software source code from the projects Version Control System (VCS). The most commonly used tool for this at the moment is Git (*Git* 2019).

2.1.2 Build automation

Build automation refers to a setting where the developer can start a process that automatically and without intervention completes all of the build steps needed for the software in question. Manually completing the build process with its multiple steps containing complicated commands has a lot of potential for accidental mistakes, creating variation in the builds. The goal of build automation is to eliminate the source variation and therefore reduce the number of defects. Simultaneously the time required to complete the build is reduced, as the response time of the developer between each build step is removed. (Fowler and Foemmel 2006)

Multiple systems feature automated build tools, such as the make utility in Unix systems. Make is a software utility designed to automatically determine the parts of a large software that are in need of recompilation, and issue necessary compilation commands. Make utility requires the use of makefiles that are written to describe the relationships of files in the software and the necessary commands for updating each file (*make(1) - Linux manual page* 2019). The automation can be taken one step further with the help of utilities such as CMake, which generates the makefiles needed by the Make utility (*CMake* 2019).

Proper build automation also includes test automation. As we mentioned before, thorough test suites are also an important part of a good build process. Test automation allows the developer to catch a majority of software bugs, although in the case of a complicated software no amount of testing will be enough to identify

all bugs and errors. A larger amount of tests also lengthens the duration of testing and building, which will in the long term reduce the productivity of the developer. A good balance in the size of the test suite and the exhaustiveness of the tests is said to be a situation where tests do not run notably over ten minutes (Meyer 2014). Build automation is a kind of prerequisite for implementing Continuous Integration (CI), which is discussed next.

2.1.3 Continuous Integration

Continuous Integration is a software development practice, in which developers integrate code changes into a stable revision of the software frequently, even multiple times a day. The aim of this practice is to reduce the costs originating from problems with the integration of new changes in the software. The integration phase consists of combining separately developed features into one integrated build. Unexpected problems often arise when multiple developers have done code changes simultaneously, and it is very difficult to anticipate the amount of work needed to fix integration issues. Problems in this phase, therefore, cause delays to the release of the software and the correction of the problems causes extra costs. As CI practices instruct developers to integrate changes often, the changes will be small. The smaller the changes, the fewer integration issues arise, and the more money and time is saved. (Fowler and Foemmel 2006)

The practice of Continuous Integration relies on all the developers working on the project to work with the same software code, leading to the requirement of maintaining a single mainline that always contains a stable revision. In addition to the software code, the mainline should also contain all the scripts, test environments and test cases related to the software. When all the necessary tools for building the software are stored in the same source, developers are ensured the same development and testing environment. (Fowler and Foemmel 2006)

Testing is integral to the CI practice, as keeping the mainline stable and free of defects is priority number one. If the mainline has defects, all the developers basing their change on the newest update on the mainline will suffer from the defects. Finding such a defect can be difficult and use up multiple developer's time. The first testing of a new change in the software is done by the developer; they test the new feature after integrating it into the mainline in their local environment. This itself is not enough to prevent all defects from entering the mainline, and thus test automation is also necessary. The CI principles instruct that the CI tools run the build automation steps, including testing, on every single change introduced to the mainline before integrating it. Testing every change individually allows early catching of faults, reducing the amount of time it takes to fix the fault. (Fowler and Foemmel 2006)

At its simplest, CI can be a tool monitoring your version control system for changes, compiling and testing your application with every detected change. CI has the potential to be much more though, as with the help of CI tools it is easy to also monitor your codebase health, code quality, and code coverage. The direct consequence of this is that technical debt and maintenance costs stay low. Indirectly,

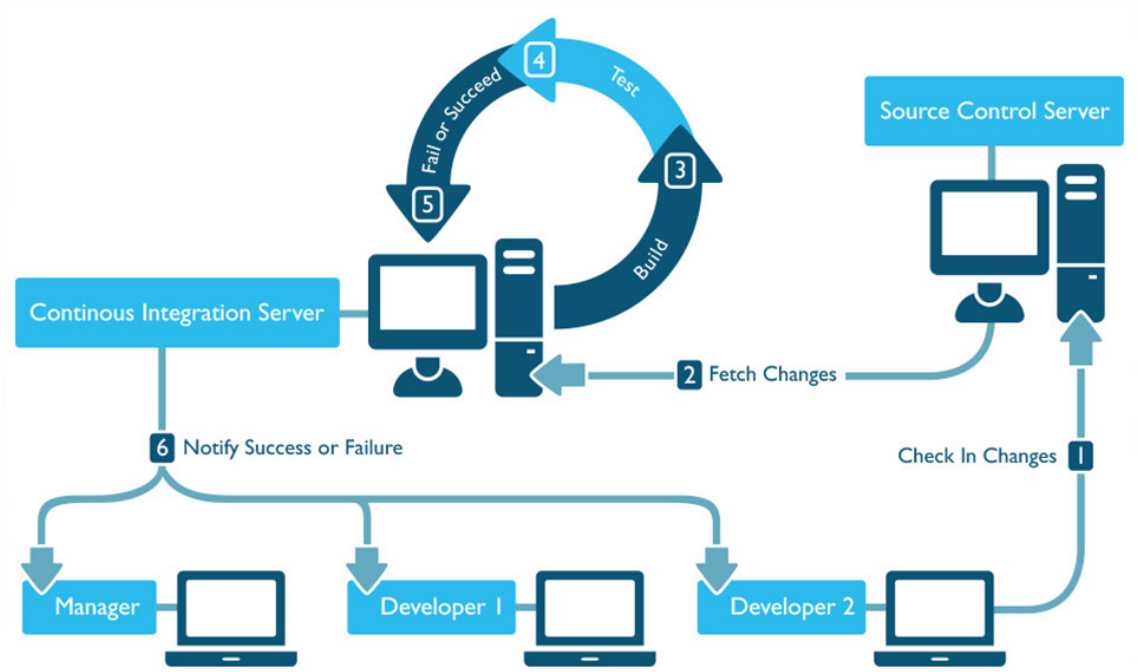

Figure 1: Technical implementation of Continuous Integration

Automated software testing in Continuous Integration (CI) and Continuous Delivery (CD) – Continuous Improvement 2019

CI report results can also motivate developers to improve their code quality and strive for better results (Smart 2011). Figure 1 shows a common technical implementation of CI. It usually relies on a server handling the source control and a server handling CI operations. After a developer has pushed the change to version control, the CI server fetches the change. CI server automatically builds the software, runs the necessary test suites and finally reports on the results. The results are displayed publicly to the development team, allowing each member to keep track of the current stage of development. (Smart 2011; Fowler and Foemmel 2006)

2.1.4 Reproducibility in software development

In scientific research, reproducibility is described as the repeatability of a process that establishes a fact, or the repeatability of the conditions in which we can observe the fact (Boettiger 2015). That is to say, another researcher must be able to repeat the steps described in research in a different computing environment and come to the same solution. Scientific contributions in the field of computer science and software engineering rely on the reproducibility of the software artifacts published with the research, such as algorithms, tools, prototypes and other computational analyses (Cito and Gall 2016).

Common software development also greatly relies on reproducibility. To contribute to the development of software, a developer must be able to build, test and run

the software. This requires the developer to repeat the conditions in which the building and running of the software are possible, in other words, the build and run environment conditions must be met.

To enable reproducibility, software development projects rely on best practices, such as extensive documentation and automated tests. The documentation should include a sufficiently detailed description of the algorithms and solutions found in the software, in addition to the test setup and parameters. However, as mentioned earlier, the computing environment plays a large role in reproducibility, and as such needs to also be thoroughly documented (Fehr et al. 2016). Documentation may require for example information about the hardware used, the operating system in use and the libraries and dependencies needed. Automated tests, on the other hand, are used to easily and reliably check that the changes made to software still run on the documented computing environment, reaffirming that the software is still reproducible (Fehr et al. 2016).

To assist with setting up the environment, multiple technologies such as virtualization and containerization are used. These tools allow software designers to package as much of the computing environment into a distributable image as possible. The technologies vary greatly on their implementation and are discussed thoroughly in the following subsections.

2.2 Virtualization

Virtualization is a technology, that allows division of the partitions of a computer system into multiple isolated virtual environments. In short, it allows computers to virtualize an altogether different computer environment than is installed on the actual system.

Common uses for virtualization include using it for desktop virtualization and using it for server virtualization. Desktop virtualization refers to one computer being able to run several OS instances and can enable the desktop user to use applications that are restricted to a specific OS. Server virtualization, on the other hand, refers to a use-case where a data center administrator creates multiple virtual instances on a single server machine. It is common practice to rent these instances on a subscription basis, usually referred to as Platform as a Service (PaaS) (Bui 2015).

Different virtualization technologies have different approaches to which partitions are virtualized, and how many are reused from the host machine. In this chapter, we discuss two virtualization technologies: Hypervisor-based Virtual Machines and the lightweight alternative, Containers. We also discuss the most popular container technology Docker.

2.2.1 Hypervisor-based virtualization

Computer software or hardware that can create and run a virtual machine is called a hypervisor. Hypervisor-based virtualization provides a high level of isolation and has widely been used for virtualization for over a decade. Hypervisors work on the hardware level, therefore supporting virtual machines that are totally independent

and isolated from the host system (Morabito, Kjällman, and Komu 2015).

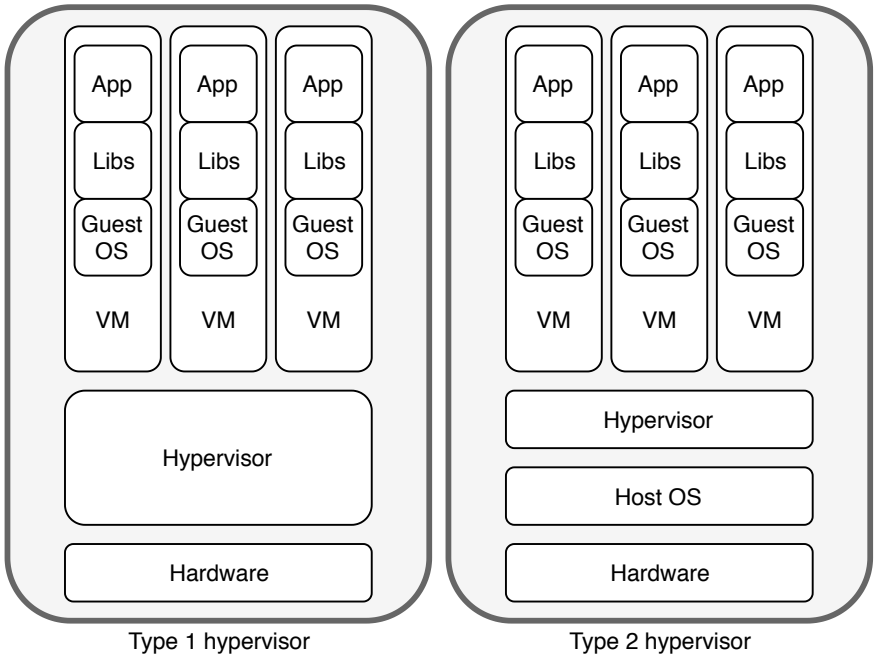

Figure 2: Hypervisor type comparison

As stated in Popek and Goldberg 1974, hypervisors are divided into two separate types known as Type 1 hypervisors and Type 2 hypervisors. Type 1 hypervisors are also known as native or bare-metal hypervisors, whereas Type 2 hypervisors are also known as hosted hypervisors. A comparison of the hypervisor types is found in Figure 2.

Type 1 hypervisor has the hypervisor layer directly on top of the hardware, with no host Operating System running in between. As a result, the hypervisor does not depend on the Operating System for its operations, making it highly applicable to server environments. Type 2, on the other hand, is installed on the Operating System running on the hardware, and therefore runs the VM operating systems on top of the host OS. This results in total dependency of the host OS, where any problems in the underlying OS can affect the Virtual Machines. This type of hypervisor is commonly used on Personal Computers to simulate another OS for testing or development purposes.

Hypervisor-based virtualization is often used in data centers, which contain multiple physical servers. It provides a way to have a single machine host multiple applications that depend on differing Operating Systems, and may even be running on behalf of unrelated organizations. As multiple clients may have their applications running on the same hardware, it is highly useful that hypervisor-based virtualization provides full isolation (Soltesz et al. 2007).

The downside for hypervisor-based virtualization is that the computing efficiency is low, as each Virtual Machine runs its own separate kernel and operating system. The images created of virtual machines are also notably large, as the full operating system is included in it (Popek and Goldberg 1974).

2.2.2 Container-based virtualization

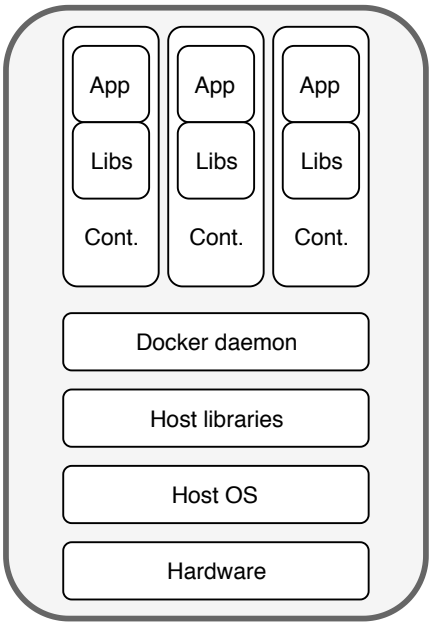

Figure 3: Docker stack

This section discusses containers; the more recent, more lightweight alternative to virtual machines. In comparison to hypervisor-based virtualization, containers work on a different level of abstraction when it comes to virtualization and isolation. Instead of virtualizing hardware and having virtual device drivers and a full operating system on top, containers isolate processes only on the operating system level. That is to say, the containers run on a shared operating system kernel of the host machine, with separated processes and applications running in each one (Morabito, Kjällman, and Komu 2015). Because of this, container-based virtualization is also known as operating system-level virtualization. Visualization of one container solutions architecture can be seen in Figure 3, with most container solutions having a similar stack, only with the Docker daemon replaced by the container engine of each solution.

Container-based virtualization also provides isolation between containers running on the same host machine. The isolation is normally achieved via the use of kernel namespaces. Kernel namespaces are a feature in Linux kernel, that provide different processes a different view of the system and its resources (Xavier et al. 2013). Namespaces can be used to wrap global resources in a layer, which results in the illusion of the container being its own system. Resources such as the filesystem, process IDs, inter-process communication and network can be isolated with the use of namespaces (Biederman and Networx 2006). Other resources, such as CPU, memory and I/O usage, can be isolated and restricted for containers with the use of Control Groups (cgroup) (Xavier et al. 2013).

The approach containers take to virtualization brings multiple benefits, but also some setbacks. Due to a shared kernel, all the containers running have no computational overhead from running their own OS and kernel, as virtual machines

do in the case of hypervisor-based virtualization. This also brings the performance of the containers very close to native performance, because no simulation of hardware is necessary. Also, a great benefit from the shared kernels is the highly reduced image size, as no OS needs to be packaged in the image (Morabito, Kjällman, and Komu 2015). The bootup time of containers is also very fast compared to VMs; it usually measured in seconds, whereas for a VM it can take multiple minutes. Sharing the kernel also brings some trouble, for example, this makes it impossible to run Windows-based containers on a Linux machine and vice versa. Running Docker on Windows actually uses a Linux VM for enabling Linux containers, which naturally lowers the performance to the level of VMs due to the computational overhead. Some concerns have also been raised on the security of the isolation containers provide, as the host kernel is exposed to the containers (Morabito, Kjällman, and Komu 2015).

2.2.3 Docker

This section discusses Docker, its implementation and its relevant concepts. Docker has recently become the most popular container engine and is, therefore, the most relevant for this study (Pahl 2015). It can be described as an engine for automating the deployment of applications into containers, and as open-source software has become increasingly popular. Docker builds its solution on top of the Linux Container (LXC) project, thereby inheriting the usage of namespaces and cgroups to isolate containers and distribute computational resources to containers (Pahl 2015).

Docker approaches the dependency in a similar fashion as virtual machines, which is by providing a binary image. This image has all of the needed dependencies already installed and configured. The main difference between Docker images and VM images is that Docker images don't contain the kernel, as the containers use the host machines Linux kernel directly. This eases the load on the host machine, but also requires the host machine to be running Linux (Boettiger 2015).

Docker handles version control and image distribution with the help of Dockerfiles. Dockerfile is a small text file, which provides a simple script that defines the image and how to build it. The Dockerfile is ideal for storing in a version control system and is an easy way to distribute the Docker image to a fellow developer. The Dockerfile is also a good human-readable summary of what the container image includes (Boettiger 2015). Docker images consist of a series of data layers, which each correspond to a single instruction in the Dockerfile. This allows an efficient way of storing Docker images, as one layer needs to be stored only once but can be used in many images. Docker uses a solution known as Union File System (UFS) to allow this layered approach by combining files and directories in multiple file systems to a single consistent file system. (Bui 2015)

Images for Docker can also be distributed through online repositories. Docker uses commands familiar to developers from Git, such as push and pull, to ease the distribution to an online repository. The official Docker repository known as The Docker Hub is used by default and has become a popular way for companies and developers to publish their Docker images (Combe, Martin, and Di Pietro 2016). It is also possible to run your own Docker repository, that can be run for example in a

company's internal network.

The Docker daemon engine uses features of the Linux kernel to isolate the containers from each other and the host system. Especially Linux namespaces are used for the task, and creating namespaces requires privileged capabilities. This has historically been done by running the Docker daemon as the root user, but a lot of work has been done to relax this requirement. A new feature still in development known as rootless Docker allows non-root users to run their own Docker daemon and run Docker containers on it. Linux allows users without root access to create user namespaces, but they can only map a single user, meaning that multiple containers would not work. To mitigate this problem, rootless Docker uses a package known as uidmap to handle the remapping of user namespaces. With this solution, the uidmap package still has to run as root internally, but the Docker daemon itself does not. The namespace mapping differences between rootless Docker and default Docker are shown in Fig. 4. (*Experimenting with Rootless Docker - Tõnis Tiigi - Medium* 2019)

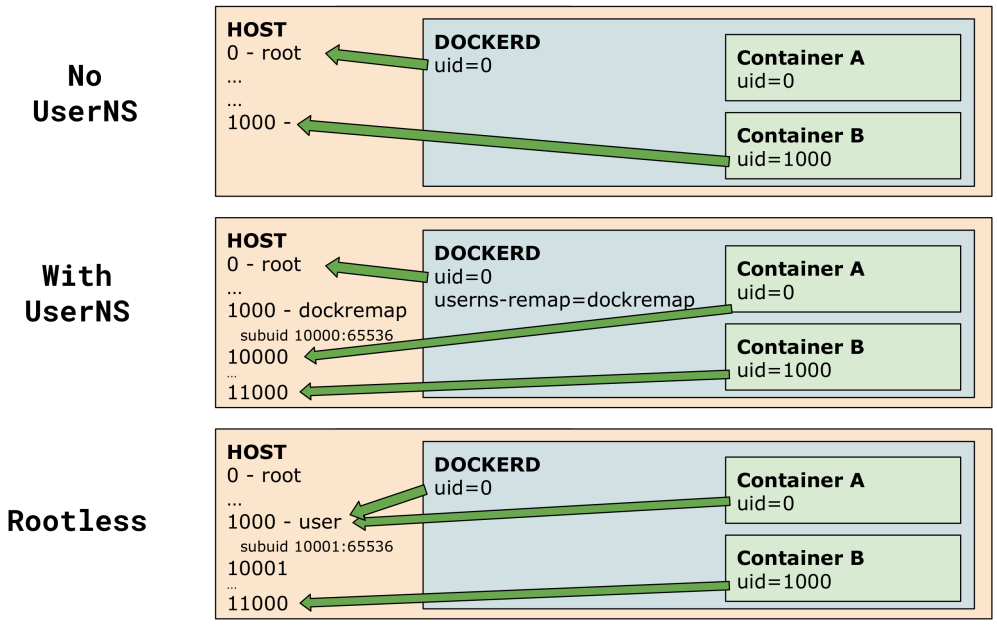

Figure 4: Namespace mappings in rootless Docker daemon compared to default Docker daemon

Experimenting with Rootless Docker - Tõnis Tiigi - Medium 2019

2.3 Related work

The idea of using containers and Docker to create a build or development environment is not a new one. The use case is often not mentioned in scientific texts, but multiple blog posts and instructions on the use case can easily be found. Some computer software focusing on development have also found ways to integrate containers

into their workflow, a solution that also shares some characteristics with our build container idea.

In the blog post *How to build a project inside a Docker container | Bartosz Mikulski* 2019 Mikulski discusses his recent experience in a hackathon, where he had to download code written by other developers and run it on his machine. Mikulski opted to use containers to build the software, instead of their host machine, to mitigate the possible harmful code found in the software. He achieved this easily by adding all required dependencies into a Dockerfile, building the image and running it with the project code mounted as a volume. This relates to our needs, as we are looking for a simple and easy solution.

Another blog post, *Using Docker Containers As Development Machines - Rate Engineering - Medium* 2019, discusses common problems in software development and ways Docker containers can be used to mitigate them. The post describes how developers often need to download a large number of tools and dependencies to set up their development environment. It is said, that as different developers use different operating systems, there is a high chance of cross-platform compatibility issues arising. Another issue they mention is how their developers run all their integration tests on a dedicated platform, even though they could run it on their own machine. Setting up the test environment has the same problems as the development environment. For solving these problems, the blog post showcases the use of a Docker container, that contains all the dependencies related to the development and testing of the software. They mention, however, that their development workflow with Go language installs the dependencies in the project folder and mounting it to the container creates problems, so the solution is not without its troubles.

A popular code editor, Visual Studio Code, has released a feature in their software that allows the user to connect the editor into a container. With the connection in place, the editor can run commands to for example build the software in the container, and otherwise utilize its full feature set in the container. (*Developing inside a Container using Visual Studio Code Remote Development* 2019)

The fact that a significant amount of solutions similar to the one we are planning shows that the idea is not without merit. It also makes the development of our solution a lot easier, as we have many sources where we can find instructions and new ideas.

3 Case description

This chapter discusses our case study environment. First, we discuss the company in question to provide some context on the business it operates on. Then we discuss more in detail the department and its specialization. Lastly, we provide information on the project technology; what programming languages and tools are used. We mostly discuss the tools related to the build process and CI, as they are the most relevant to this thesis. We also discuss the computing environments which are used to allow the development of the project software.

3.1 The case company and department

Nokia is a large, multinational telecommunications company originating from Finland. It provides it's customers with a large variety of equipment and software related to mobile and fixed networks. In 2018 Nokia employed over 100 000 people in over 100 countries and conducted business in over 130 countries, reporting annual revenue of approximately 23 billion euros. (Nokia 2019)

Nokia as a company has a history dating back to 1871 when it was operating pulp mills generating hydro-power. During the early 1900s, Nokia also worked in rubber and cable industries, and it was only in the 1970s when it started operating in the networking and radio industry. Nokia is most notably known for its successes in the mobile phone industry in the 1990s and early 2000s, reaching its peak in 2007 with a 40.2% market share in the smartphone market share. (*Nokia grabs 40% of phone market for first time • The Register* 2019) Ultimately, however, Nokia sold its mobile phone business and has since focused on its networking business. Most recently 5G networks have become one of Nokia's main areas of expertise.

The thesis was conducted in the Mobile Networks business group of Nokia, more precisely in the Layer 1 business unit. Mobile Networks hosts all business units related to Nokia's mobile network ecosystem and therefore drives the development of all 4G and 5G base station hardware and software. Layer 1 business unit handles the physical layer software in 4G and 5G base stations, which is also known as Layer 1 software. To further explain the context in which Layer 1 software is used, we discuss the basics of mobile network technology as well as the technology behind 5G networks in the following section.

3.2 The case technology

As mobile networks and 5G are such a broad and complex subject, we discuss them here to provide the reader with a better understanding of the context behind the case project. We begin by discussing the basics of modern mobile networks and the building blocks needed to build one. We then focus more on the technology behind 5G mobile networks, and finally, we discuss Layer 1 software in 5G.

3.2.1 Mobile networks

Mobile networks, also known as cellular networks, are networks of base stations, providing connectivity to a range of mobile devices connected to the network. Base stations themselves are connected to a larger network and the internet by a physical cable connection. The connected devices are called user equipment (UEs) and are mostly composed of smartphones, laptops, and tablets. The variety of devices is on the rise however, with the advent of internet of things (IoT) that can bring connections to household devices, vehicles and so on (Xiang, Zheng, and Shen 2016). As the amount of mobile devices has risen dramatically at the beginning of the 21st century, more and more effort has been put forth into creating technology that can provide fast mobile networks to billions of devices. Here we focus on modern mobile networks, and their division to three planes: the user plane (UP), control plane (CP) and management plane. (Arnold et al. 2017) Also the basics of base stations are discussed briefly in addition to the different generations of mobile networks.

The user plane is also known as the data plane, and it handles the forwarding of data from the data source to its destination. This includes all the processing included in the different steps of the process. The control plane, on the other hand, is used to control the user plan. It is responsible for example for settings the routing paths of packets and radio resource management. Other responsibilities of the control plane include connection management and base station system information broadcasting. (Arnold et al. 2017). Management plane, on the other hand, defines the physical composition behind the network and handles control plane configuration and monitoring schemes (Gember-Jacobson et al. 2015). CP and UP are discussed more in Section 3.2.2 in relation to their implementation in 5G networks.

Every base station has a limited range from which UEs can connect to it. This range is known as a cell, and every UE inside the cell will connect to the base station corresponding to the cell. To create an extensive coverage for a mobile network, base stations must be geographically placed in a way such that no gaps exist between the cells. It is not possible to achieve this without some overlapping of the cells, and so it is required of the cells located next to each other to use different frequencies to prevent the mixing of communications between a UE and two base stations (Xiang, Zheng, and Shen 2016). An example of base station placement and corresponding cells is shown in Fig. 5. Frequencies of the base stations are also marked in the figure, and order of frequencies where no same frequency is next to another is showcased.

Mobile network technologies are often referred to as generations, the latest being fourth-generation (4G) and fifth-generation (5G). First-generation mobile networks consisted of simple analog radio signals and were used in the 1980s. The industry switched to digital signals in the advent of Global System for Mobile Communications (GSM), which was designed for voice transmission and had very little support for data transmission. An update known as General Packet Radio Service (GPRS) was introduced to GSM to increase data transmission capabilities, and it is considered to be the 2.5 generation in mobile network technologies. With the internet becoming more and more used by the public the needs of mobile networks have grown to need more and more data transmission capabilities. This was the reason behind

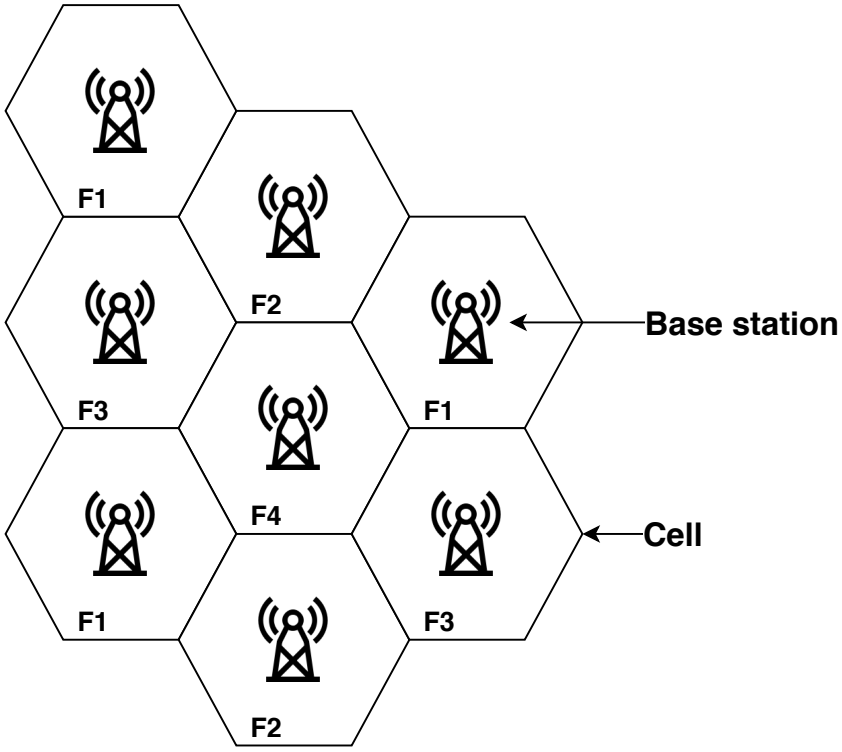

Figure 5: Base stations and corresponding cells

the upgrade to 3G and an updated version known as 3.5G known as High Data Packet Access (HDPA). This network is still used as a fallback when more modern networks are unavailable. 4G was designed to handle the increase of mobile demand for multimedia content and interactive applications and was also upgraded with Long Term Evolution technology known as LTE. Previous generation upgrades have focused on bringing about a higher transmission rate and increased quality of service. However, the advent of IoT has brought numerous new UEs, such as cars and utilities, to mobile networks. This network traffic is known as Machine-Type Communications (MTC), and has some special characteristics such as small data exchange transmission, need for energy efficiency and a very large amount of devices. Needs such as these have been taken into consideration in 5G, which is the latest generation in the mobile network industry. 5G is discussed separately in section 3.2.2 (Xiang, Zheng, and Shen 2016).

3.2.2 5G Layer 1 technology

5G protocol stacks for UP and CP are visible in Fig. 6 and Fig. 7, we focus on Layer 1 technology in this section, as the department works on software for that specific layer. Layer 1 refers to the physical layer, which is closest to the actual hardware of the UP and CP. It is marked on the figures as PHY. Next Generation Node B (gNB), as displayed in Figure 7, is a type of base station supporting 5G New Radio communications. The objective of Layer 1 is to provide information on the data that is to be transferred to layers above it. (Räbinä 2019)

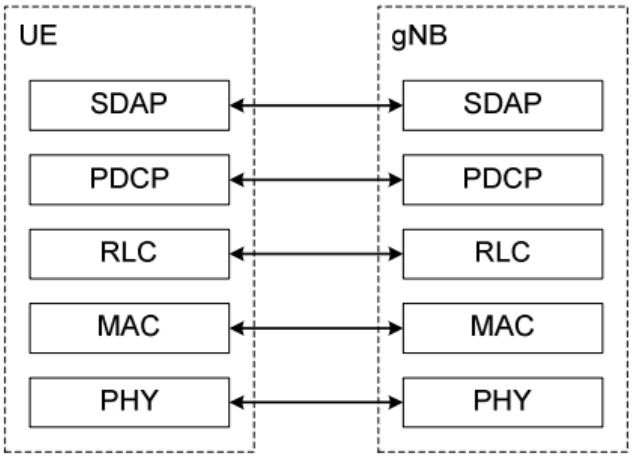

Figure 6: 5G User Plane protocol stack

5G NR Overall description 2019

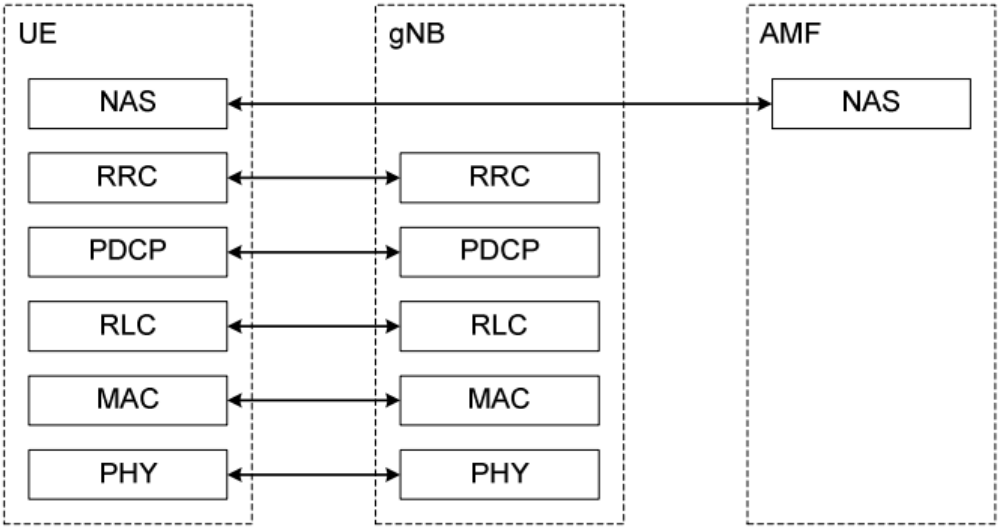

Figure 7: 5G Control Plane protocol stack

5G NR Overall description 2019

The data transferring in L1 communications uses orthogonal frequency-division multiplexing (OFDM), which is a digital encoding method for multiple carrier frequencies. The benefit of using OFDM is the ability to transfer with several closely spaced frequency bands without interfering with each other. Normally when using multiple bands to transfer data, an empty space between the bands must be in place between the bands to negate the effects of overlapping. In the case of OFDM however, the bands can be right next to each other, as the bands are orthogonal to each other and thus overlapping becomes a non-issue. (Diniz, Martins, and Lima 2012)

Layer 1 communications can be divided into three channels in both the downlink and uplink. The first channel is the physical shared channel, known as the physical downlink shared channel (PDSCH) in the downlink and the physical uplink shared channel (PUSCH) in the uplink. These physical shared channels transfer user data, higher layer data related to the UE, paging and system information. The second channel is the physical control channel, known as physical downlink control channel (PDCCH) in the downlink and as physical uplink control channel (PUCCH) in the uplink. The physical control channel is responsible for carrying control information and in the case of PDCCH the scheduling of transmissions in PDSCH. In uplinks case, PUCCH carries control information from UE to gNB. The third channel in the downlink is the physical broadcast channel (PBCH), it is used to broadcast simple system information. The third channel in uplink, on the other hand, is the physical random access channel (PRACH), which transfers random-access preamble from UR to gNB, with the purpose of notifying the gNB of a random-access attempt and having it adjust parameters of the UE. (Räbinä 2019)

3.2.3 System on a Chip

System on a Chip (SoC) is a single independently working chip, that contains all the components of an electronic system integrated to the chip. The electronic system can be for example a computer, or something more specific, like a base station. The various components that can be integrated into a SoC often contain components such as a central processing unit (CPU), graphical processing unit (GPU), memory, radio transceivers and so on.

3.3 The case project

As discussed in Section 3.2, 5G networks rely on a dense network of base stations to provide connectivity. The base stations consist of hardware capable of working on network challenges such as modulation and demodulation, synchronization and beam-forming, and often all the hardware is integrated into a single SoC.

The base station SoC naturally requires software to perform the previously mentioned network functions, as well as other Layer 1 functions mentioned in section 3.2.2. Our case project focuses on this exact software, specifically developed for a new type of SoC being taken into use.

The project department consists of nearly 200 developers, most of which work with the actual software development. The developers work in groups, each of which is specialized in a specific part of the 5G stack introduced in Section 3.2.2. The department also contains groups that are specialized in helping the other developers by developing the way of working, CI tools and DevOps tools. Each of these groups is formed by less than 10 developers. Architectural decisions in the software are made by a group formed of lead developers.

The main software code is in C++, but the code-base also contains MATLAB, Python and C code that provide helpful functions to be used by the main software or designed to help developers working on the project. The code-base also hosts

unit-tests, test automation configuration, CI configuration, and multiple software tools.

4 Research methods

This chapter discusses the research methods that were in use in this study. We begin by discussing the chosen research questions, and the reasons for choosing them. We then discuss the methods we used to gather data and explain the phases of research we went through.

4.1 Research questions

The goal of this thesis is to solve a problem in the case company's department's context. The problem consists of the department having a complex build environment, that should be usable in multiple computing environments. Manual work in setting up the environment takes time away from the developers, and an automated solution is needed.

The second problem is maintaining the environment. More manual labor is created by the fact that multiple computing environments need to be updated whenever the software dependencies change. These two problems lead us to have two research questions for this thesis:

1. How to create a universally reproducible build environment?
2. How to maintain a reproducible build environment?

To find the answers to these questions, we use methods discussed in the following sections.

4.2 Design Science

Design science is a widely used practice in Information Technology research that focuses on the engineering of artifacts that are used to solve problems in a real-life context. (A. Hevner and Chatterjee 2010). The artifacts can be anything tangible created in the research process, such as a piece of software or a process model. As the purpose of this thesis is to provide a new way to launch and maintain a build environment, design science is very suitable for our purposes.

Guidelines for Design Science Research are the following, according to A. Hevner and Chatterjee 2010:

1. Design as an Artifact
2. Problem relevance
3. Design evaluation
4. Research contributions
5. Research rigor
6. Design as a search process

7. Communication of research

Each guideline is briefly explained below.

Design as an Artifact states that Design science must produce an artifact to solve a problem. As mentioned earlier, the artifact can be for example a piece of software, process model or a method. The artifact we introduce is explained and evaluated in detail in Section 6. Problem relevance guideline states that the problem we are trying to solve with our technology-based solution should be important and business relevant. The relevance of our problem is discussed in Section 5. The Design evaluation guideline states that the design artifact must be evaluated in utility and quality with well-executed evaluation methods. This thesis discusses the evaluation in Section 7. (A. Hevner and Chatterjee 2010)

THe guideline for research contributions requires that the research must provide verifiable contributions to the areas of the design artifact and design foundations. We discuss the contributions of this thesis in Section 7.5. Research rigor guideline states that rigorous methods must be applied in the construction and evaluation of the design artifact, we discuss the methods we are using in Section 7 as well as at the end of this chapter. Design as a search process guideline says that researchers must utilize available means while satisfying the laws in the problem evaluation to find an effective design artifact. We discuss the related literature and other background data in Section 2, and the laws of the problem environment are discussed in Section 5. Finally, Communication of research guideline requires us to present the research in a way that is effective to both technology-oriented audiences as well as more management-oriented ones (A. Hevner and Chatterjee 2010). To achieve this, we will communicate our findings to the scientific community through this thesis. A more practical communication will be done to the company in question in the form of documentation, instructions, and support on company communication channels.

More insight into the functionality of design science research can be gained by acknowledging and understanding the three design science research cycles that are a part of design research projects, the cycles are visualized in Fig. 8 (A. R. Hevner 2007). The three cycles are known as relevance cycle, design cycle and rigor cycle and they are discussed below.

The Relevance cycle refers to the change of information between the environment and the actual research. The environment, in this case, refers to the application domain and includes the organization, people and systems that will be using the artifact produced in the research. The relevance cycle feeds requirements from the environment into the research, and the research conducts field testing in the environment. The testing conducted can, of course, bring about more feedback and requirements starting the cycle again. This cycle is closely related to the previously mentioned guidelines of problem relevance and design as a search process, which state that the research must produce a solution to a relevant business problem and must satisfy laws in the problem environment (A. R. Hevner 2007; A. Hevner and Chatterjee 2010). In our case, this cycle is seen when information is gathered from interviewees on how to make the solution more relevant to our case.

Rigor cycle links the research to the common knowledge base, which includes all related scientific data, theories, and methods, as well as similar previous implemen-

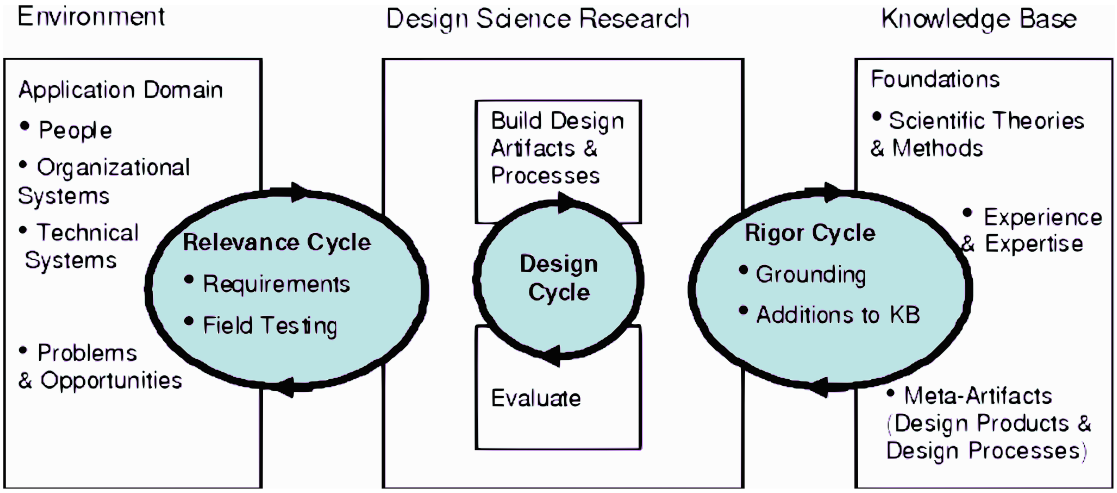

Figure 8: The three design science research cycles

A. R. Hevner 2007

tations. The rigor cycle feeds the grounding knowledge to the research, and in time the research itself will feed the knowledge base with new information, completing the cycle. This of course again has ties to one of the previously mentioned guidelines, research contributions, which states that verifiable contributions to the scientific community must be present in design science research (A. R. Hevner 2007; A. Hevner and Chatterjee 2010). In this thesis, this cycle is visible in the background research on the used technologies, as well as in the lookup of relevant previous studies.

Finally, the design cycle is the research's internal cycle, in which the researchers build design artifacts, evaluate them, and based on that improve the artifacts. This cycle also feeds on the knowledge brought by the relevance cycle in terms of requirements and rigor cycle in terms of scientific theories and methods. This cycle corresponds to previously mentioned guidelines known as design as an artifact, design evaluation, and research rigor (A. R. Hevner 2007). Our solution will work according to this cycle, by developing the solution step by step and gathering feedback from developers to guide the development process.

4.3 Literature review

To find out about relevant technologies and related previous implementations, a literature review was carried out. Relevant keywords that were used in searching for the related literature included: build system, reproducibility, containerization, Docker, and maintainability. Keywords were chosen based on our previous knowledge of containers and Docker in relation to reproducibility.

For scientific publications, Google Scholar search engine was used. Another source of scientific publications in the form of previously written Master's theses was Aaltodoc, which contains the Aalto University Master's thesis database search engine. In order to find literature on previous implementations of using Docker for maintaining a build system, we also used Google Search engine to find blog posts

and guides on the subject.

The knowledge achieved in the literature review is discussed in detail in Section 2.

4.4 Quantitative data

To gain an understanding of the impact of our artifact, we collected some quantitative data before the artifact and after the artifact. The data can then be compared to achieve information on the quantity of improvement the artifact has produced.

For our use case, the amount of ways to gather quantitative data is quite low. Setting up the build environment takes time, but no developer keeps timing the process. Also the time it takes to conduct the process varies as a developer does not have to conduct each step of the process every time, and can also become more proficient in the process steps after they have done them multiple times.

A small test was also conducted, where a first-timer sets up the environment before and after the publishing of the artifacts created in this thesis. This should provide us with at least some level of information on the time reductions we have achieved. Another test we can conduct to get interesting quantitative data is to compare the build system performance when running on Docker for Linux compared to Docker for Windows. The data is not directly related to our research questions, but it is of interest to the department as it sheds light on the possibility of running the build environment on Windows laptops.

The majority of data relevant for our research question comes from the qualitative data, which is described in the following section.

4.5 Qualitative data

We gathered qualitative data about the situation before the introduction of the artifacts, as well as after the introduction. Qualitative data was gathered with the use of semi-structured interviews. Semi-structured interviews are a type of interview where predetermined questions are used, but the interviewer can also seek clarification and explore new conversation paths as they emerge during the interview. The open nature of these questions leads to a conversational style of interview, which has been found to help new concepts emerge. (Doody and Noonan 2013)

The interviews are conducted on a group that contains developers and other professionals from our department. The interviews focus on inspecting the backgrounds of each individual, as we all any prior experience with relevant technologies and the department software itself. We asked for opinions about the state of the build system and suggestions on improvements.

Section 4.5.1 discusses the set-up of the interviews, what kind of questions we asked and other details of arrangements. The interviewees are introduced and their backgrounds are discussed in section 4.5.2. The results of the interviews are mainly discussed and analyzed in Section 5.3 and Section 7, but they are also mentioned in other chapters when relevant.

4.5.1 Interview format

The interview was conducted as a semi-structured interview, which used a planned outline but was allowed to get side-tracked. The outline of the interview can be seen in Appendix A and it contained the following subjects.

1. The interviewee background

   To gain context on the interviewee's opinion, we gathered data on the interviewee's background. We wanted to gather information on the role of the interviewee, in addition to their experience with fields and technologies related to our project and plan. These fields and technologies included topics such as mobile radio networks in general, Layer 1 software, build environment of our product, Docker, and containerization and Linux management and configuration. With this information, we can better estimate the interviewee's understanding of the big picture in our environment as well as gauge the technical level to which we must aim our planned containerized build system.

2. Opinion on the current build system

   We wanted to gain honest opinions on the current system. It was important for us to know what works well and what is in need of improvement. To get this information, we asked the interviewee their general opinion on the build process, as well as what is good and what is bad. We also asked what the interviewee thinks the current difficulty level of operating the build system is, along with what they think could be improved in the build system.

3. Interviewee build flow

   We gathered information on how the developers actually use the build system in their daily work, as it is very possible that not all developers use the preferred and instructed workflow. We asked the interviewees to explain step by step how they develop and test the software.

4. Manual set-up of the build environment

   To get to know how many developers in our department know the details of what it takes to set up the build environment, we asked if the interviewee has ever manually set up the build environment. This question also revealed what kind of manual setups developers have created, what problems they have encountered and how difficult the process was.

5. Containerization plan and feedback

   At this point, we introduced our plan of using containerization to ease the setting up and maintaining of the build environment. We asked the interviewees what they thought of the plan, what potential benefits and drawbacks it includes and how the plan could be improved.

6. Open feedback

To check if we missed anything, we also gave the interviewees a possibility of providing us with open feedback on anything related to the build environment. This allowed us to gain feedback also on subjects that are not in the scope of this thesis, such as ways to speed up the building of the software and how to make the user actions clearer.

For the interview we chose five developers in a couple of different roles, to gain a wider understanding of the situation. Slots of one hour were reserved for each interview, which proved a suitable amount of time. The interviews were conducted partly face to face and partly through a remote voice call.

Table 1: Summary of the initial state interview results

	Interviewee 1	Interviewee 2	Interviewee 3	Interviewee 4	Interviewee 5
Role	Senior developer	Product owner, technical leader	technical leader, software architect	Developer	Lead Product Owner
Experience: Mobile radio networks	18 years	12 years	11 years	13 years	14 years
Experience: L1 software	12 years	8 years	10 years	13 years	10 years
Experience: Project build environment	Uses daily	Uses every other day	Uses daily	Uses daily	Multiple times daily
Experience: Containers	No experience	Medium experience	Low experience	No experience	No experience
Experience: Linux	Low	High user level experience	High level	High level	No experience
Opinion on current build system	Sometimes slow	Usually fine	Easy but crowded	Remote connection slow, otherwise okay	Reliable but slow
Manual set-up experience	No	Knows how to	Yes, had trouble finding documentation	Done in previous project, was slow on laptop	No
Opinion on plan	Good, as long as easy to use	Good, needs planning to make work, enables new features in CI	Excellent, helps developers who use laptops for build	Workflow change fine if everything works well. Possibility to use laptop for build is nice	With clear instructions sounds good

4.5.2 Interviewee background

We had 5 interviewees, all with a long background in the telecommunications field as well as Layer 1 software development. Interviewee 1 is a senior developer, who has worked with mobile radio networks for 18 years and Layer 1 software for 12 years. They have been in the case project for 1 year, meaning from the beginning of the project start. They build and test the software every day on the IT provided server. They have no experience with containers or Docker and have some experience with Linux.

Interviewee 2 is a product owner and a technical leader, with some responsibilities in CI development. They have over 12 years of experience on mobile radio network technologies and have worked with Layer 1 technologies for 12 years. They have worked in our project for a year and build the project software at least every other day. Docker is familiar to interviewee 2, they mention they do not consider themselves experts on the subject, but has used the technology extensively and has pushed for its usage in previous projects. They also have high user-level skills in Linux.

Interviewee 3 is a technical leader and software architect. They have been working with mobile radio network technologies for 11 years and 10 years with Layer 1 software. They joined our project 1 month ago, and now build the software every day developing the software. The interviewee is familiar with Docker and has used it occasionally, is interested in using it more. The interviewee is a high-level Linux user, with a significant amount of experience configuring Linux machines.

Interviewee 4 is a developer who has worked in telecommunications and Layer 1 technologies for 13 years, starting from 3G development and now working in 5G development. They build our project's software daily and are very familiar with the build system. They have very little experience with containers and Docker but have an extensive amount of experience on Linux systems.

Interviewee 5 is a lead product owner, meaning they are responsible for overseeing the development process of the product and guiding its vision and strategy. They have worked for 14 years in mobile radio networks and 10 years in Layer 1 software and they build the project software multiple times daily on the IT provided server. The interviewee has no Docker experience and has low-level skill in Linux.

During the Section 5.3 where we discuss the results of the interviews, interviewees are referred to as "interviewee x" where "x" denotes the same number that is mentioned in this section, and in Table 1.

5 Problem analysis

This section analyzes the problems in our current build process solution. First, we describe the starting point; what kind of technology are we using, how the software development cycle looks and what kind of environments are used to build our software. Secondly, we describe a concrete example of the steps needed to reproduce the build environment before our container environment. Lastly, we discuss interviews that were conducted on the employees working closely with the build environment and summarize the problems that can be observed in the initial state of our build process.

5.1 Starting point

The purpose of this chapter is to explain in detail the current technical situation in our department. We discuss the technology that we have in use, with the term technology in this context referring to the computers, servers and cloud machines as well as the installed software. We also list the software requirements for our build environment, to provide some insight into the challenge of setting up the environment. The build automation system and CI tools of our department are also discussed.

To gather data on the initial state of the build system, an interview was conducted. It is analyzed in this section and is the main comparison point when we evaluate the success of implementing our solution at the end of this project.

5.1.1 Current technology

The computing environments that are in use in the department are listed in Table 2. The table has the following columns: environment, usage purpose, OS and maintainer. The environment column specifies each environment with a name that is used to discuss the specific environment later in this section. Usage purpose gives a brief description of the intended purpose of use for the environment. OS column informs on the Operating System running in the environment and the maintainer column informs on who is responsible for keeping the environment up to date in regards to it functioning as a build environment. Each row is discussed separately in the following text.

IT-provided server machines are powerful computing environments, which developers can access via Secure Shell (SSH). Developers use the machine for building the software during development and test it. The machines are provided by IT and are pre-configured to match the software build environment conditions. Any changes to the build environment conditions must be informed to IT, who will update the servers to match new conditions. The servers have Red Hat Enterprise Linux (RHEL) as the operating system.

OpenStack cloud is used to provision build agents for our Jenkins instance in charge of our CI pipelines. Multiple automated jobs that require the building of our software are running on these agents. Our CI team is in charge of the configuration of these agents, and changes in the build environment must be conveyed to the CI team in order for them to update these agents. The agents also base themselves on RHEL.

Table 2: Different computing environments used in the department

	Usage purpose	OS	Maintainer
IT-provided server	Building software during development	RHEL	IT
OpenStack cloud	CI build agents	RHEL	CI team
DevOps framework cloud	Personal Linux development environment	Linux	DevOps team
Lab Kubernetes cluster	Running services and build agents	Linux (containers)	Lab team
Personal work laptop	Company tools, web access, development	Windows, Linux	Developer

Another cloud that is in use is the DevOps framework cloud. This cloud is used to provide developers with a private Virtual Machine instance for development purposes. Many flavors of Linux can be requested for the instance, with the preferred being RHEL.

The department laboratory contains multiple bare-metal servers, which have been configured to function as a Kubernetes cluster. Any Docker containers can be launched here, with an internal IP allocated to each container. Server machines are running RHEL, which enables the use of Linux based containers with minimum computing overhead for the containers. The cluster is mainly used for piloting and running services that help developers, but also some Jenkins agents are launched from the cluster.

Each employee at Nokia is also issued a personal work laptop. The laptop usually is running Windows 10 as the operating system, but developers also have the possibility to run a Linux distribution of their choice on the laptop. Most developers opt to use the internal Windows 10 distribution, as it eases the use of company tools. Developers in our department have laptops with a high-end CPU and a large amount of memory, so development work on the laptop is possible.

5.1.2 Build environment requirements

In order for a computer to be able to build our software, it has to meet some requirements. Our build environment requirements are the following:

- OS: Red Hat Enterprise Linux 7

- Approximately 3 GB worth of software packages, installed by YUM package manager
- In specific use cases, Matlab installation may be needed. Takes 10GB space
- A very large read-only file systems, taking dozens of GB of space
- Git repositories stored in Gitlab and Gerrit
- SVN repositories

The list is quite long, and manually setting up all of the items is a large task.

5.1.3 Build automation and CI tools

The tools we discuss here are Gerrit, Zuul, and Jenkins. These three tools form most of our CI pipelines, with Gerrit functioning as the version control and code review system, Zuul working as a gating system and Jenkins working as an automation server. We begin by introducing the tools more in detail and later we discuss the connections between these tools.

Gerrit is a web-based tool for hosting Git repositories, discussing code and reviewing code. In other terms, Gerrit manages Git repositories and integrates itself into the Git workflow. It provides Git with a more complex project structure with access control and integrated code review. With the code review function, each change into a repositories code can be forced to undergo a code review, in which other developers and automated tools check if the code is functioning and clean. Depending on the results of the code review the change can be either accepted, after which the developer can submit the code change, or rejected, in which case the developer will continue development and fix the found flaws. (*Gerrit Code Review | Gerrit Code Review* 2019) (Laster 2016)

Jenkins is an open-source build automation tool and a Continuous Integration tool. It is a software running on a server, that hosts a web user interface for managing the build automation. Jenkins is designed to be extensible, with hundreds of open-source plugins available. With the plugins, Jenkins is capable of communicating with version control systems, running build tools and tests, creating code quality metrics and build notifiers. Plugins also allow Jenkins to integrate with other servers, containers, clouds, and clusters, allowing it to use them as agents to run builds and tests. It is also possible to use command-line tools and Application Programming Interfaces (APIs) to control the functionality of Jenkins. (Smart 2011)

Zuul is a project gating system, functioning as the middleman between Gerrit and Jenkins. It runs on a server and connects directly to Gerrit code reviews and connects to Jenkins through a Gearman server and plugin, as seen in Fig. 9. The functionality of Zuul revolves around the concept of pipelines. A pipeline is a configurable workflow process that can be applied to multiple projects found on the connected Gerrit server. Pipeline configurations consist of the preliminary condition, trigger and assigned Jenkins jobs. The preliminary condition can, for example, be that a reviewer has given their approval on Gerrit code review. The trigger can be any action in the

Gerrit event system, such as a particular comment or the addition of a new patchset. The jobs can be any jobs known to the connected Jenkins. Example workflow could be one, wherein the case of a new patchset to a code change triggers a Jenkins job that runs a static code analysis on the code and returns the result to the code review. (*Zuul Concepts — Zuul documentation* 2019) As of version 3.0 and up, Zuul has dropped support for Jenkins connections and has instead opted to use OpenStack Nodepool as a framework for running jobs (*Nodepool — Nodepool documentation* 2019). Due to this fact combined with our CI team's familiarity with Jenkins, we are using Zuul 2.6.0, which is the latest Zuul version supporting Jenkins connections.

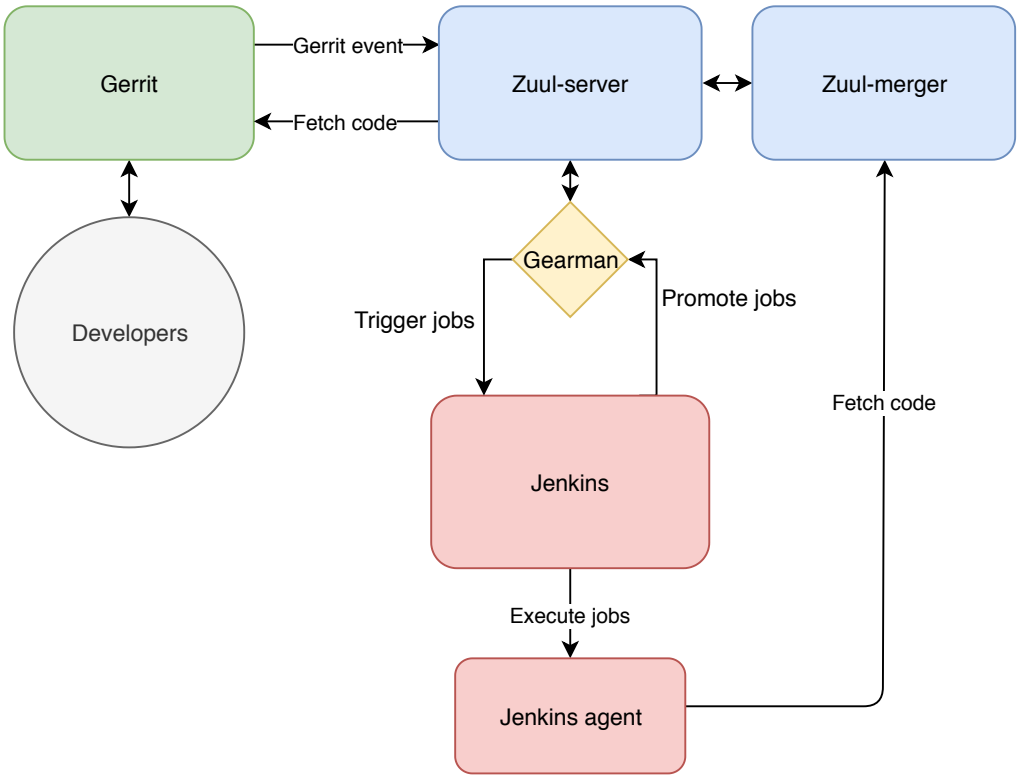

Figure 9: Zuul connections and workflow

The workflow in our department is the following. Each developer works in the same repository and usually on the master branch. When change is done and tested by the developer, the change is submitted to Gerrit. Gerrit automatically adds reviewers who have previously modified the now changed code and the developer adds other related reviewers to the change. Gerrit launches a suite of tests in Jenkins that checkout the code on a cloud instance, build the software and run simple tests. If tests are executed correctly, Jenkins sets a Verified +1 tag to the review in Gerrit. Reviewers check the code and give it either -1, +1 or +2, basically indicating that the change is either lacking, okay or okay to submit. When the change is given a +2, Zuul is triggered and it launches a more comprehensive suite of tests. Zuul rebases the change to the newest revision on master or a change queued before it for the same tests. This ensures other simultaneous changes haven't broken each other. After Zuul has triggered the test suites, Jenkins starts running the comprehensive test suite on

multiple remote build agents. If the tests are executed correctly, Zuul submits the code change to the master branch in Gerrit and the workflow has completed. At any point in the workflow in the case of a failed test or -1 review from any developer, the original developer may submit new patches to fix any found flaws and the workflow process will restart.

5.2 Build environment set-up example

A build environment set-up was conducted, in which a Virtual Machine running a Linux distribution was manually configured to match the build environment requirements and build the software and run build-tests. The host machine for the Virtual Machine was a personal work laptop running Windows 10 as its operating system. The guest system in the VM was based on an image for Red Hat Enterprise Linux 7.6, which corresponds with the build environment requirement. It is assumed the host computer has the necessary software for running VMs pre-installed, in this case meaning Oracle Virtualbox. Steps to produce the build environment were timed with the accuracy of minutes.

The first step in the process was finding the appropriate image for installing the correct operating system on the VM. The requirements state that the OS should be RHEL 7.4 or newer. As RHEL is a Linux distribution designed for enterprises paying for a RHEL subscription, the image is not easily available online. Searching for the image online easily leads to suggestions to use a CentOS image, an open-source version of RHEL. Some time was spent looking for the correct image online and in the company's internal network. Eventually, the correct image is located in the intranet, the search taking 27 minutes. Installation of the OS on the VM takes 11 minutes.

Configuration of the system begins, as the OS is now installed. As the company intranet requires the use of proxies, some time is taken by their configuration. In order to install the required packages with the help of YUM (Yellowdog Update, Modified), the default package manager in RHEL (*yum - Trac* 2019), some internal software repositories must be configured. Some time is also taken by the installation of Virtualbox guest additions, which result in better user experience in operating the VM. The previous steps take 13 minutes, and the following installation of required YUM packages takes 15 minutes.

Next follows the configuration of the Network File System required for software building. Time is taken by the search for correct URL and port, in addition to the commands used to mount NFS to the machine. After some problems with access control, the total time comes down to 32 minutes.

Cloning the required Git repositories takes too long at over an hour, and after some debugging the implementer debugs the problem and finds the installed Git software version is too old. Git is upgraded, and the repositories re-cloned, resulting in a much better clone time of 3 minutes. Debugging and upgrading took 17 minutes. Cloning some Subversion sub-modules also runs into problems, as a shared library object is not available. The implementer spends some time debugging the problem and finally manages to fix the problem by exporting an environment variable that

adds the correct shared library object to the library path, resulting in the SVN checkout taking 55 minutes. Checking out Git sub-modules takes 2 minutes.

Finally, the user runs the command that builds the software and runs the necessary tests. The process takes 68 minutes and ultimately fails in one section of testing. After two repeats of the failed tests, the implementer spends time debugging and finally re-traces their steps and restarts from the repository cloning stage, using up three hours of time. After this, the command for building and testing is re-run, with the process taking 66 minutes with all of the tests succeeding. In total, it took eight hours and nine minutes for the implementer to create a functioning build environment and build and test the project software in the environment.

5.3 Interview

In order for us to grasp a better picture of the initial state of our build environment and tools, we interviewed five developers in our department. We wanted to also gain an understanding of the technical skill level of our department in addition to gaining new input on our plans for enhancing the build environment with the use of containers. A summary of the results is found in Table 1.

Sections 5.3.1, 5.3.2 and 5.3.3 we discuss the results of the interviews divided into categories.

5.3.1 Opinions on current build system

We asked for opinions on the current build system, referring to the recommended way of building, where a developer connects to one of the IT provided servers and builds the software there. The connection to the server can be done with either SSH or VNC (Virtual Network Connection).

The IT server-based build process was thought to be an easy and somewhat reliable solution. Interviewees 1, 2 and 3 mentioned the build process is easy to execute. Interviewees 1 and 2 mentioned specifically that it is easy because with the use of the IT servers only one command "make build" is really needed to execute the basic build of the software. Interviewee 2 mentioned that this is much easier than manually having to use CMake commands. However, interviewee 2 also mentions that the build system is hard to use if you want to customize the build commands, for example enabling Valgrind and sanitizers is hard. Interviewee 5 mentions that the current system has been working well and has been reliable, and it is a common sentiment between all the interviewees that in general, the system has been mostly reliable.

One notable finding was, that the build system often suffers from slowness. The build servers are usable by every developer working on the project, and as mentioned by interviewees 1, 3, 4 and 5 they sometimes become crowded and work slowly. On the other hand, interviewee 2 mentions the servers have become less crowded since a "niceness" parameter was introduced to the build system, which improved an individual builds ability to stop hogging all resources if other builds are in need of them. Other reasons for the slowness of the build system are also mentioned.

Interviewees 1 and 5 mention that the build system is slow because it builds every part of the software every time, and think it should be possible to build only some parts of the software. Interviewees 3 and 4 agree with the sentiment and also mention that a software called scache, an internal tool that can be used to host a shared cache for parts of a software compilation, could help with the problem and speed up the build. interviewee 3 also mentions a software called distcc, that could be used to outsource the computation needed for building to a server (*GitHub - distcc/distcc: distributed builds for C, C++ and Objective C* 2019). Interviewee 1 also mentions that a previous project they worked on had a feature that allowed the build system to build only the parts of the software that had been changed since the last build, interviewee 2 explains this could be done with software known as ccache (*ccache — Compiler cache* 2019).

Other interviewees mention that they use the IT provided servers almost exclusively, but interviewee 3 mentions they prefer using a VM they have created instead. They mention that this is a common way of working in their team and that they have put some effort into making the build working on VMs and Linux laptops. They mention that this simplifies the development workflow, as you can develop, build and test the software in the same environment and it is all done locally. They mention that the environment being local significantly helps with the integrated development environment (IDE) being smooth to operate, as opposed to the over the network style of the environment provided by the servers.

Some other feedback was also given on the build system. For example, some improvement in instructions is thought to be needed. Interviewee 5 mentions that it would be nice to have more examples in make help and have more descriptive instructions all-around. Interviewee 1 also hopes for more instructions on the style check and CMake usage. Interviewee 4 mentioned also, that they have found it annoying that the build workflow has been changing frequently, but admits that it is understandable as the system is still being developed.

5.3.2 Manual set-up experiences

Most interviewees do not use a manually set up build environment, but instead, opt to use the IT provided server environment with a pre-configured build environment. Here we discuss the reasoning behind developers the choice to use the servers, and the experiences some interviewees have had setting up the build environment manually.

Interviewees 1 and 5 mention they have not tried to set up the environment manually, and also mention that they do not see a point in trying to do so. It seems many developers are satisfied with the current solution or do not see the benefits outweighing the work needed to create the environment.

Interviewee 3, however, uses a manually configured environment exclusively. They mention this is a common habit in their team, and they have co-operated on setting up the environment. Interviewee mentions that they had some trouble setting up the environment. According to them, the first problem comes from the fact that not every developer wants to use the recommended RHEL distribution of Linux. This leads to problems, as not every dependency is easily installed on other distributions.

For example, on a Ubuntu machine, they had to use an older version of CMake than is default on Ubuntu, resulting in some debugging. Also, some problems were created by the somewhat lacking documentation of the build environment, resulting in some trial and error tactics being needed.

Interviewee 3 acknowledges that building on the powerful server machines is faster than on your own laptop, but only if they are not crowded. Due to this it is often beneficial to run the build on your own laptop, either in a VM or directly on the machine if it is running a Linux OS. More benefits of using your own computer include having complete control of the software installed on the machine and not having to use editors over a network, therefore avoiding an annoying latency.

Interviewees 2 and 4 have previously tried using a manually set up build environment but mainly use the build servers as they have found them faster.

5.3.3 Reception of the idea for containerized build environment

Before this section, each interviewee was given a brief overview of the solution we aim to develop. The overview explained how the current build environment is complex to maintain and difficult to set up, and how we will try to ease the situation with the use of Docker containers. If the concept of Docker or containers were not familiar to the interviewee, they were also briefly explained. The previously asked question about the interviewee's experience with containers also provided some valuable context when we explained the solution.

The general opinion on using containers for the build environment was positive, even when the interviewee in question had no experience in the technology. Interviewees 1, 4 and 5 all mentioned that the new build solution must be easy to use. Interviewee 1 explained the importance of ease of use, "if it is too hard, people will not use it". Interviewee 4 agreed, saying that a workflow change is usually fine as long as things work and hoped that we could provide an introduction to Docker in our project documentation. Interviewee 5 mentioned the importance of instructions, "clear instructions go a long way", they said.

Many interviewees saw the benefits of our solution, especially the fact that the same environment would be available for all computing environments. Interviewee 3, who had told us their team uses pre-configured Linux laptops and VMs for building the software, was the most excited about the container. "It's great. It enables many ways to build the software", they said. Interviewee 4 shared the same thought, and said "If it enables cloud, laptop, and server to have the same environment, it will be useful". Interviewee 2 mentioned it would also allow the usage of Kubernetes clusters as a source of Jenkins agents in our CI, enabling better scaling for when a large amount of computing is needed.

Interviewees 2 and 3, who also had the most Docker experience also gave some of their ideas on improving the solution. Interviewee 2 mentioned they have seen similar solutions before and instructed us to look into Docker-compose, which is a tool for defining and maintaining Docker containers (*Overview of Docker Compose | Docker Documentation* 2019). They also explained that Docker image layers can be used to easily provide multiple similar images, with small changes. For example, a

base image can be provided that simply contains all the dependencies of the software. On top of this image, you could then build a developer image that adds for example compilers and other software that only developers need. More layers could be added to have the image function in even more scenarios. Interviewee 3 mentioned, that a Windows-compatible solution could benefit multiple people, because "Many of those who want to develop on their laptops have Windows, and setting up a virtual machine would be too much work". They also mentioned that it could prove useful to have a bare-bones version of the environment, as well as versions that have all the software that can be needed.

Some interviewees also mentioned possible problems and challenges in our solution. Interviewee 2 mentioned the possible problems related to versioning and promoting of the Docker images. For example, we have to choose if we want to simply have a commit hash as an id for each Docker image, or if we want to use numeric human-readable version numbers. Promoting images may also prove problematic, which version is the latest and which should a developer use when they need an older version of the environment. Interviewee 4 also wondered about the versioning and if the speed at which a laptop or cloud instances can build the software would be too slow.

6 Design and implementation

In this section, we discuss our process of finding a solution to the problem of creating a maintainable and reproducible build environment. The discussion also includes how we implemented the solution, and how we took feedback from the interviews into account when building the solution.

As described in more detail in Section 5, the problem we had in our build system was mostly related to maintenance issues and usability issues. The build environment had grown into a large complicated system, that was hard to replicate in your own machine. Instead, multiple separate entities were maintaining this environment in separate locations; one for developers to connect to and build and test their software changes, one for Continuous Integration operations such as automated testing and so on. Developers who wanted to run the build environment on their machine had to put time into researching the specifics of the system and then spend more time installing the dependencies and solving compatibility issues. These problems lead us to research possible ways of packaging the build environment in such a way, that developers and maintainers could all use the same package.

For packaging the build environment, we decided to use containers. We wanted the solution to be as lightweight and fast as possible, so Virtual Machines were out of the question. Preferably, we wanted a way for the user to simply run a command and after thatn they would already be in the build environment. Containers provided these features, and as Docker seems to be the most supported container runtime engine in addition to the fact that we had previous experience with it, we chose Docker for our main technology.

Docker revolves around Dockerfiles that are a kind of a human-readable instruction file that Docker uses to generate an image. The files contain, for example, all the dependencies and scripts that provide the entry functionality in the container, so the way we implement it is a large part of the solution. This is discussed in more detail in Section 6.1.

An important aspect of the solution is a way to start the container in a way that is easy and intuitive even for the developers who have no prior Docker experience. As Docker commands can become quite complicated, especially since we need to mount multiple volumes from the host machine to the container and handle the access to them inside the container, we want to provide a start script that can be used to launch the container. Preferably, the script should be as simple to use as possible, while also providing advanced possibilities for the more experienced users. This script is discussed in Section 6.2.

The image we create is stored in a Docker repository running in our internal network. It provides developers and other users the possibility to fetch the image instead of having to build it according to the Dockerfile mentioned previously. This saves time, as it is much faster to download the image than separately download the dependencies in the image while building it from the source. As the build environment evolves, we will have to upgrade the image and version it accordingly. Challenges related to these things are discussed in Section 6.3.

6.1 Docker image

The usual first step for creating a new Dockerfile is finding a suitable base image on which to add new layers. As our old build environment setups are based on Red Hat Enterprise Linux 7, we decided to also use a RHEL. Our IT hosts a Docker image for RHEL 7.6, which we use as our base image. Using RHEL ensures we can install the same dependencies and packages as the previous build environments, which means there will be no compatibility issues and we have clear instructions for installing everything.

Each instruction in the Dockerfile creates a layer, which in the image contains all the changes from the previous layer. As such, to keep the image size to a minimum, it is beneficial to try to reduce the number of layers as much as possible. For example, when we are installing the dependencies with RHEL package manager YUM, we want to install all the dependencies in one instruction as well as clean the YUM install cache in the same instruction. Also, by adding all of the dependencies into on YUM install command, YUM works much more efficiently compared to it having to look up every package and their dependencies separately. To demonstrate the effects of placing multiple YUM install commands into on instruction, we measured the time it takes to build the image and the image size on three different scenarios: running each dependency install in their own layer, running every dependency install in one layer and finally running every dependency install in one layer and cleaning YUM install cache in the same layer. The results can be seen in Table 3.

Table 3: Build time and image size in different Dockerfile scenarios

	Build time	Image size
Dependencies in layers	34 min	15.6 GB
Dependencies in one layer	21 min	14.4 GB
Dependencies and clean cache in one layer	21 min	9.67 GB

According to the results, we can reduce the build time from 34 minutes to 21 minutes by moving the dependency installations to one layer. The build time is not that relevant in the big picture though, as the build only needs to be done once for every change in the Dockerfile. The image size is a lot more important, as every developer and user must download the full image to run it. Moving every dependency installation to one layer decreases the image size from 15.6 GB to 14.4 GB. Major additional size reduction can be done by cleaning the YUM cache in the same layer, which leads to the image size being 9.67 GB. Although the cache cleaning takes some time, the following total time of building the Docker image stays the same, probably because I/O (Input/Output) operations are faster with the smaller image size. The 5 GB of reduction in size also leads to major time savings for developers and lessen the network load created by the transfers of the image. Due to these reasons, we decided to go with the last option.

Another important part of the Docker image is the start script running inside it. It is discussed in Section 6.2, but in relation to the Dockerfile, we followed the usual way of working with Docker. This way includes us using the COPY command in the Dockerfile to copy the wanted script to the image and setting the script as the entrypoint in the Dockerfile. With a setup like this, we get an image that when started runs our start script.

6.2 Start script and entry script

As we started developing the Dockerfile, we noticed that quite a lot of data needs to be passed to the container from the host machine. Passing data to the container in this context means providing environment variables when running the container and mounting volumes to the container. Docker supports both of these very well, you can do it by adding parameters to the Docker command. However, the Docker command for running a container like this grows very complex. We decided to provide a start script, that automatically provides as many of these parameters to make the usage easier. This section discusses the decisions we made when developing the script. Another thing we discuss here is the entry script inside the container, which contains some configuration necessary for getting the container working.

6.2.1 User handling

Due to us mounting multiple volumes from the host machine and them possibly having strict access control, we need to make sure the user inside the container has access to these. Linux access control looks at the current users UID (User ID) and GID (Group ID) to decide if the user has access to a file or directory. Files and directories always have a designated owner and owner group, both of which can be given specific access rights to the object. In order to ascertain the user's access to mounted files inside the container, we dynamically create a user inside the container with the same UID and GID as the host machine user.

In order for the container to get information about the host users UID and GID, we have to provide them as environment variables. We have added this functionality to our start script; the user's identity is fetched with the use of `id -u` and `id -g` Linux commands and automatically passed to the container by using the following option in the Docker run command:

```
-e ENV_VARIABLE_NAME:ENV_VARIABLE_VALUE
```

The entrypoint script sees the provided env variables and creates the user with matching UID and GID inside the container. The host username is also provided and used with a similar method.

Creating the user dynamically when starting the container creates some issues. One issue is created from the fact that the `useradd` command we use to create the user creates a home folder for the user as well. However, if a directory like `/home/<username>` already exists, the command fails and leads to unexpected behavior. This scenario would happen if a volume is mounted to that exact home path,

which is very possible as we find out later. We discuss this scenario and the ways we avoid the problem in Section 6.2.3.

6.2.2 NFS volume mount

One required part of the build environment is a large NFS (Network File System) volume, that needs to be mounted over the network. In practical terms, NFS is used to mount a directory residing on another computer in a network. The volume is not intended to be packaged into an image, as it is almost 180 GB in size, but it contains some data valuable to our build process.

We first thought to implement the NFS mounting inside the container, using the common Linux mount command as shown below:

```
mount -o ro <IP of server>:/path/on/server /path/on/host
```

Here the option `-ro` signifies that the mounting happens as read-only, meaning we have no access to change any of the data residing in the volume. This approach, however, did not work as expected. This is due to Docker security principles, which restrict the access of a Docker container to the Linux kernel features such as our mount command. We could circumvent this by using the –privileged flag with Docker, which would give total access to kernel features. It is, however a significant security risk, which we do not want to take.

Instead of mounting the volume inside the container, we decided to use Docker volumes as an intermediary to mounting the NFS to the container. Docker volumes are, according to Docker itself, the preferred way for persisting data used by Docker containers. The volumes are completely managed by Docker, and can, therefore, be safely and reliably used on different hosts and containers (*Use volumes | Docker Documentation* 2019). Docker volumes can also be created directly from a NFS, with the following Docker command:

```
docker volume create \\
    --driver local \\
    --opt type=nfs \\
    --opt o=addr=<IP of server>,ro \\
    --opt device=:/path/on/server \\
    --name named-volume-1
```

Here we can see multiple arguments that are needed to have a working NFS volume. The first argument, `--driver local`, means that the volume is created physically on the host machine running the container. The argument `--opt type=nfs` informs the Docker volume system on how to mount the volume. We can use an argument to define the IP of the NFS server and the fact we want to mount it read-only like this: `--opt o=addr=<IP of server>,ro` and `--opt device=:/path/on/server` to define the path to mount from the NFS server. The final argument defines a name for the volume we created, which can be used later to mount the volume to a container or delete the volume.

After creating the volume, it is easy to mount it to a container when one is launched. This is done with the following argument added to the Docker run command, as stated in *Docker Documentation* 2019:

```
-v named-volume-1:/path/in/container
```

This approach was functional and is in use in the start script. However, we also wanted to allow the use of an existing NFS mount on the host computer. Using an existing NFS mount would allow us to skip the docker volume handling and creation, and also saves us from some network traffic overhead when multiple users do not have to mount their own NFS volumes. To achieve this, we implemented an automatic check in the start script. If the NFS volume is already mounted in the default path on the host computer, it is used instead. The start script also accepts a user-provided argument that overrides the Docker volume creation and the mount from the default path.

A simple volume mount like the one described above, is done with the following argument in a docker run command:

```
-v /path/on/host:/path/in/container
```

This way of mounting volumes inside a container is called a bind mount. This method does not allow Docker to manage the content of the volume, but in our case, that is not required. (*Use bind mounts | Docker Documentation* 2019)

In summary of how our script functions in regards to the NFS mounting, the following applies. The script first checks if the user has provided a path for the mount. If the path exists, the path is mounted as the volume. If it has not been provided, the script checks if the host machines default location for the NFS mount exists. If it does exist, it is mounted. If not, a Docker command to create a NFS volume directly from the NFS server IP and path that are hard-coded into the script and the NFS volume is mounted. This flow ensures we always have a functioning NFS mount inside our container when it is launched.

6.2.3 Working directory mount

To provide good user experience, we want the user to be able to continue their work from the same folder they ran the start script from. To do this, we mount as much of the working directory as possible and dynamically move the user to the same directory inside the container. This section discusses the methods we used to make this happen.

The first problem is identifying how much we should mount to the container. The start script is found inside the project repository, so the first approach we took was to mount the repository itself and move the user inside that directory in the container. For this purpose, we use a shell script function that starts in the directory the user launched the script and moves upward in the directory tree until it finds a directory that has a name identical to our repository. The absolute path of this directory is used to mount the directory to the container.

At this point, we used a hard-coded path where we mounted the working directory. Inside the container, we then created a symbolic link to that directory inside the user's home folder. This resulted in a scenario, where the user launches the start script inside the project repository, the container launches and the user ends up inside their home directory inside the container, with the project repository visible in the same directory. However, as we found out in Section 5.3, the developers we interviewed asked for a solution that was as easy to use as possible. We decided to try to provide a better experience, by keeping the absolute path inside the container the exact same as it was inside the container, in addition to having the user start the container usage in the exact same directory as they were in on the host machine.

To achieve this goal, we tried to have the start script dynamically mount the found working directory to the exact same absolute path as on the host machine. This had the possibility of leading to a critical issue we already mentioned in Section 6.2.1. The critical issue could occur in the following way. The user clones our project repository to their home folder on the host machine and the project repository path is, therefore, `/home/<username/<project>`. The user then moves to the repository and launches our script for starting the containerized build environment. The start script finds the formerly mentioned repository path and mounts it to the exact same path inside the container. The mounting of volumes happens before the entry script inside the container, so when the `useradd` command inside the entry script tries to execute and create a home directory in the path `/home/<username>`, it fails as the directory already exists. This leads to undefined behavior and the user lands in a container that is not prepared correctly.

To counter this issue, we decided not to mount the working directory in the exact same path inside the container initially. Instead, we mount it to a known safe path inside the container. We still want to provide the user with the project folder in the same absolute path as it was on the host machine, so we included a solution for this in the entry script that runs inside the container. We could have simply set the script to copy the working directory to the path we want to have it available in, but this would result in a scenario where the user's changes inside the container would not be reflected in the host machine working directory. This would then lead to unexpected loss of data, as the user would at some point make changes inside the container, exit the container and notice their data is missing. Instead of copying, we, therefore, create a symbolic link to the working directory mount in the path we want the directory to be available from.

The results we saw with this implementation were satisfactory, in the regard that we were able to always have the working directory available in the same absolute path as it was on the host machine. The second thing we wanted to accomplish was to have the user start inside the same directory they were in when running the start script. This was simple to accomplish, at least when we expect the user to be inside the project repository. As the working directory is now mounted to the same path inside the container, we can simply collect the user's location inside the start script on the host machine and pass it to the container in an environment variable. The entry script inside the container then adds a command to move the user to the same path when starting the container. However, were the user to be outside the project

repository at the time of running the start script, this approach had the potential to fail.

While testing our solution, we noticed that sometimes it is handy or even necessary to have access to other repositories or files that contain important data or code. As only the project repository was mounted to the container, these were not easily available. We thought of a better solution, where we mounted as much of the host user's data as possible. However, we could not simply mount a static predetermined part of the host machine storage, as the script and container will be used on a machine that has multiple users. This means users will not have access to everything, and we should only mount the directories in control of the user. In order to accomplish this, we made a change to our start script. Instead of looking if the start script launcher is inside the project repository and mounting the repository, we search the directory for the first directory the user is in control of and mount it. For example, let's look at the following scenario.

User is inside the directory they use to keep all their software project found in path `/var/work/<username>/projects`. The user clones the project repository to `/var/work/<username>/projects/<project>`, launches the start script for the build environment from the projects-directory. The start script starts looking for the first directory owned by the user, starting with the projects-directory. The directory belongs to the user, so the script looks up `/var/work/<username>` next, and as it also belongs to the user `/var/work` is checked as well. As `/var/work` is owned by root, the lookup stops and returns `/var/work/<username>` as the first directory owned by the user. This directory is mounted to a static location inside the container, and finally, a symbolic link is created in the same path to point to the directory. The entry script places the user inside `/var/work/<username>/projects`, as that was the place they launched the start script from.

This approach also provides us a solution to the problem explained above, where we could not launch the start script outside the project repository. Two restrictions remain: we can not allow the mounting of either path `/` or `/home/<username>`, as these would override important data inside the container. In the case where either of these is the first directory owned by the user, the previous directory is mounted instead.

For the more advanced use-case, we also provide the option to run the start script with an argument indicating the path to a custom working directory. Using this argument overrides the previously described behavior, and instead mounts the indicated working directory to the container and change the starting location of the container inside the working directory root.

6.2.4 Other mounts

Our project software requires code from multiple other repositories, which are assigned as submodules in our project repository. These repositories contain Git repositories and SVN (Subversion) repositories, both of which are fetched from a dedicated server and their respective protocols. The user of our containerized build environment can fetch these submodules on their host machine and launch the container, in which

case these submodules are automatically mounted to the container with the main project repository. However, we also want to provide the users with the possibility to run the Git and SVN related commands inside the container. In order to achieve this, we have to include Git and SVN settings and credentials inside the container.

In the case of Git, there are two things we want to include in the container: .gitconfig file and the private SSH key. The formerly mentioned .gitconfig file contains information on the user, usually meaning the name and email address of the user. These are used to display the author of each code change, which in Git are called commits. We want to allow the developer to commit and push their changes even from inside the container, so this must be included in the container. The SSH private key is a key matching to the public key set on the user's profile in the Git repository, allowing access to authenticate as the user in Git operations. Without this in the container, Git pull and push commands are not functional when the SSH protocol is used. Also in the case of Subversion, we must provide a way for the user to authenticate inside the container. The user could also manually input the credentials inside the container, but it is more convenient to mount the .subversion directory from the host machine, which contains the SVN credentials.

As the .gitconfig file, SSH private key and .subversion directory are usually found in a default location in the user's home directory, our start script looks for them there and automatically mounts them to a static location inside the container. Similarly to the working directory mount, the entry script inside the container then looks at the static locations and creates static links to them inside user's home directory inside the container. As the UID and GID are the same inside the container as in the host machine, as explained in Section 6.2.1, we do not have to make any changes to the access rights of the mounted files.

As we want to allow users more flexibility in the use of our script, we also provide a way to define the paths for these three articles in arguments. This can be useful for example in the case, where the user has multiple SSH private keys on their machine and wants to use a certain one of them.

6.2.5 Shutdown and cleanup

Exiting the containerized build environment is simple. Running the exit command inside the script closes the bash running inside the container, and when it closes the entrypoint process of the docker stops as well. This stops the whole container, and as the start script uses `--rm` argument when executing the Docker run command, the stopped container is automatically removed. The directories that were mounted inside the container keep all the data changes made inside the container, but otherwise, the container is no more.

A special command is needed in the case where the NFS volume was mounted with a Docker volume, as described in Section 6.2.2. The command in question removes the created Docker volume, in order to remove the overhead caused by old Docker volumes laying on the host machine. The volume is deleted with the following command:

```
docker volume rm named-volume-1
```

Here `named-volume-1` refers to the name given to the volume, similarly to how it was used in Section 6.2.2.

With the cleanup step completed, the start script process comes to an end. The start script can be run again immediately to once again bring up the build environment.

6.2.6 CI functionality

As we want to use the containerized build environment in our CI pipelines, we have to be able to run automated tests inside the container. This means we have to provide a way to run scripts and commands inside a container based on our image, instead of the user having to interactively run the commands from the command-line. For this use case, we made some additions to the start script.

For starters, we added an option to use an argument to provide the start script with a command. The command may have spaces in it, so in order to have our bash script properly parse the complete command, it must be given in quotes. The command can then be passed to the container in an environment variable, in which case the entry script checks the variable and runs the command if it exists. In case the command does not exist, the script defaults to opening an interactive bash in the container. We also added another argument to the start script, in which the user can provide a path to a script file. The start script finds the absolute path to this file and passes it as the command to the container, in which the script executes.

The previously described solution worked well when testing it from a test machine. However, running it from an automated Jenkins job failed, however. Jenkins, which we use in our CI machinery as described in Section 2.1.3, run jobs non-interactively, meaning the programs that are run in jobs do not have access to the standard input. So far we had been using `-it` in our Docker run command, which specifies running the command as interactive and connected to the standard input (*Docker Documentation* 2019). As Jenkins jobs do not have access to standard input, the Docker run command failed. To fix this, we disable the usage of `-it` when a command or script is provided.

As Jenkins jobs rely on the return values of commands to see if they were successful, we also had to provide a way to tell if the command or script run in the container failed or not. To do this we added a `set -e` line at the beginning of our start script, which stops the execution of the script if any line in it returns an error (*The Set Builtin (Bash Reference Manual)* 2019). The Docker run automatically returns an error if the command in it fails, which now is passed to the start script and from there to the user, allowing users and Jenkins jobs to see if the command has failed.

6.3 Distribution and versioning

We have created a Dockerfile containing the necessary information for building an image with all the build requirements inside and running it as a Docker container. However, manually building this image takes a long time, and we do not expect the

users to do it manually themselves. Instead, our Docker image is deployed on an internal Docker repository, from which the users can download the image. Another thing we have to consider is the fact that as our software matures, it may eventually have multiple new software dependencies. The same applies to our build system, additional dependencies may be added and the build workflow may be changed in major ways. To keep up with all the changes, our build system image must be updated as well. This leads to us having multiple versions of our build system image, meaning we have to create a way to identify the correct version of the image and match it to the current version of the software. This section discusses the software and implementation details behind our distribution system and the versioning system.

For distribution, we are using Artifactory. Artifactory is a software tool provided by JFrog, that serves as a binary repository manager. Binary files are not commonly stored in Git, as it works on the principle that every developer has the full change history from Git on their computers and the size of the repository should be kept as small as possible. The binary files would increase the size of a Git repository substantially. Binary repository managers such as Artifactory are meant to fill this gap. Artifactory integrates well with multiple protocols and APIs, with the Docker registry API being one of them. An Artifactory Docker registry can handle Docker image hosting, versioning, access control and anything else the official Docker Hub can. (*Docker Registry - JFrog Artifactory - JFrog Wiki* 2019)

Docker image naming in a private registry works in a different way compared to using the official Docker Hub. In addition to the name of the image and the version, the image name must include the URL (Uniform Resource Locator) of the Docker registry. In the case of Artifactory, multiple Docker registries can be set up on the same Artifactory instance. In our case, we have a Docker registry specifically meant for the use of our department. This approach leads to a naming scheme where the full name of a Docker image looks like this:

```
<REGISTRY_NAME>.<ARTIFACTORY_URL>/<IMAGE_NAME>:<IMAGE_VERSION>
```

Naming our Docker images in this fashion allows us to use the Docker command `docker push <FULL_NAME>` to publish our images in the internal Artifactory, and in the same way, the users of our images can pull and use our images with the Docker command `docker pull <FULL_NAME>`.

7 Results and discussion

To evaluate our implementation of the containerized build environment described in Section 6, we gathered some quantitative and qualitative data on the solution. The quantitative data contains a measurement of the time it takes a developer to build our software with the containerized build environment. This data is compared to the similar data we gathered and discussed in Section 5.2 to get some meaningful insight into the success of our solution. We also conduct a cost-benefit analysis based on the quantitative results, to estimate the value of our solution. To gather qualitative data on our solution, we asked the interviewees introduced in Section 4.5.2 to test the solution and give their feedback. We also gathered feedback from a separate test group, which we also discuss here. This section ends with a discussion on Design Science criteria and how well this study matches the guidelines introduced in Section 4.2 and discussion on the thesis process in Section 7.6.

7.1 Build environment set-up time

A small study, similar to the one in Section 5.2, was conducted to measure the benefits gained from containerizing the build environment. The user was provided with a clean VM, and they had to achieve a successful build of the software. As previously the VM was not provided to the user, we compare the results of this section to the previous results without the time taken by the VM setup.

The user begins by opening the tutorial we provide for the build system. The user sets their SSH keys and begins cloning the project software. This seems to take too long, so the user decides to update all the packages in their VM. This process takes about 13 minutes. The user continues to the cloning of the project software repository. The cloning of the project software code takes 30 seconds. The user now has access to our build environment start script, which is described in detail in Section 6.2.

The user launches the start script and Docker begins downloading our packaged build environment image from our internal Docker repository, which is described more in detail in Section 6.3. Pulling the image from the repository takes 10 minutes, after which the user is in a functional build environment inside the container.

Continuing with the build instructions, the user initializes Git LFS and fetches and checkouts necessary binary files with it. Git LFS is already installed in the container, but the fetch process itself takes 4 minutes and the checkout process 4 minutes. The user then goes on to fetch Git submodules, which takes one minute. As the container includes the intended newer versions of Git and Git LFS, these two parts of the build process take a lot less time than in Section 5.2. Now all the necessary code and binaries are available in the environment.

The user continues with the command to build and test the software. The process takes 68 minutes. A comparison of the times it took a user to build the project software with and without the containerized build environment is found in Table 4. The times are divided into the following sections: build environment set-up, code fetching, build and test. The actual part of setting up the VM is not compared here,

as in this section the user was provided with a VM.

Table 4: Comparison of time taken to achieve a build with and without containerized build environment

	With containerized build environment	Without containerized build environment
Set-up (Installing dependencies/launching container)	10 min	60 min
Code fetching (Git, Git LFS and submodule operations)	9 min	137 min
Build and test (Building, problems and debugging included)	68 min	254 min

As seen in Table 4, in our example tests the total time saving created by using the containerized build environment was 364 minutes. The first part, set-up, takes 50 minutes less with the container than without. This is because individually fetching the dependencies from yum repositories is a lot slower than downloading the container image from our internal network. Some time was also saved by the fact that the user did not have to look up each of the dependencies, and they were instead already in the container image. In the second part, code fetching, using the containerized build environment saves 128 minutes. This was mostly because of the time it took for the user to debug the situation and finally install proper versions of Git and Git LFS, as described in Section 5.2. As the correct versions are found in the container and the fetching is done in the container, the user can count on fetching the code properly on the first try.

The last step, which is the actual building and testing of the software, takes 186 minutes less with the use of our container. One successful build takes approximately the same time in both cases, 68 minutes in the container and 61 minutes without the container. As the underlying machines in each case were quite similar, this highlights the fact that using containers brings no significant overhead to processing. The huge time difference in this step is created by the fact, that the user without the container fails the build multiple times, and has to debug and fix their environment between builds.

This small test displays the potential found in using containers for packaging build environments in an easy way. It should be noted that as only one developer was observed in both cases, we can not make an assumption that as much time is saved per every user as was saved here. Also, the fact that the build process documentation has also improved between these tests could possibly have improved the build time as well, but the results are very positive nonetheless.

7.2 cost-benefit analysis

In this section, we conduct a cost-benefit analysis to get an approximation on how many work hours are saved with the switch to our containerized build environment. In Section 7.1 we saw a time saving of over six hours when an inexperienced user sets up the build environment with the container instead of manually. In this cost-benefit analysis, we assume an inexperienced user saves 4 hours when using the containerized build environment. More experienced developers, such as the ones regularly developing, maintaining and deploying the build environment, are assumed to only save 2 hours per deployment of the build environment. We estimate there to be 10 deployments per month by inexperienced users and 20 deployments by experienced users, leading to a time saving of 80 hours per month.

We also made some assumptions based on the commit history of the repository containing the set-up for our CI machinery. The repository always needs to be changed when the build environment changes in some way, so looking at the commit history provides us with an easy way to estimate how many times per month a change is made to the build environment requirements. With this approach, we estimate the environment to need updating 10 times per month. Each change requires the CI developers to deploy the changes like these to approximately 10 machines and test them, with each test taking approximately an hour. In comparison, with the containerized build environment, this same upgrade only needs to be tested once. The container approach, therefore, takes approximately five hours per month, in comparison to 100 hours per month with the old approach, leading to a time saving of 95 hours. Similarly, the group servers no longer need to be updated through IT connection, saving approximately 10 hours per month. An added benefit is we no longer have to wait for the IT group's response, which usually takes one working day.

Developing and testing the containerized build environment took approximately 30 working days, meaning 225 hours. Additionally, we had five interviewees, each of whom spent approximately 4 hours helping in the development and testing of the solution, leading to the use of 20 hours of their time. Five additional testers also spent approximately 2 hours each testing the solution, meaning 10 hours of used time. In total, the complete solution required approximately 255 hours. As the estimated time savings per month are 185 hours, we see that after the solution has been in use for two months, we have already saved in total 115 hours. After one year since we began the development of the solution, we will have saved 1965 work hours, meaning almost 10 hours saved per developer per year.

7.3 Windows performance

As part of testing, we used the containerized build environment to build our software on multiple different machines, a Windows laptop being one of them. On other machines, we were able to achieve mostly identical results in build time with and without the container. On Windows, however, using the container resulted in a result of over three hours, whereas on native performance it should have achieved a time of less than an hour.

We investigated what the reason for this could be, and found that the I/O performance of Docker for Desktop is notoriously slow, as mentioned in the article *Docker hearts WSL 2 - The Future of Docker Desktop for Windows - Docker Engineering Blog* 2019. The same article also mentions, that as the second iteration of WSL (Windows Subsystem for Linux) is released by Microsoft, Docker will be building their Desktop solution on top of it, drastically improving the I/O performance.

At the moment WSL 2 is only available in Windows Insiders, which is an open software testing program by Microsoft (*WSL 2 is now available in Windows Insiders | Windows Command Line* 2019). Our company is not a part of the testing program, and as such we could not yet test the improvements WSL 2 Docker integration would provide on Windows machines. As WSL 2 is released to a wider audience, we can try again to use our container on Windows, but at the moment doing it is not worth the effort.

7.4 Usability

Interviewee 1 tested the functionality of our build environment, and said that "after running the start script, the process is invisible compared to the previous implementation. I think it's a plus.". In the previous interview, interviewee 1 mentioned that as long as the solution is easy to use, they are happy to use it. As they now stated, the new system feels very similar to the previous solution when the start script has been executed. Interviewee 1 also mentioned that the tests they ran on the environment were successful.

Interviewee 2 did some more thorough testing, also on the CI machines. They noticed a small flaw, where the usage of umask on the CI machines kept the start scripts hidden inside the container. The flaw has since been fixed. As interviewee 2 told us previously that they have themselves pushed for more container usage, they had no trouble launching the environment. They gave us some ideas for development though, as they noticed that downloading our Docker image takes a long time on their end. As they are based on another country, they suggested that we implement a repository proxy closer to them. Another improvement they mentioned again was dividing the image into layers and depending on the use case, to keep the base image size down. This is under planning on our end already, and will be implemented in the near future.

Our emphasis on making user interaction easy seems to have paid off, as testers find the solution simple to use and say it does not make things more difficult compared to the previous situation. A bigger problem seems to be the size of the container image, and the time it takes to fetch it from our Docker repository. We can improve the situation by creating more layered images, with different features, allowing developers to only fetch the minimal configuration they need. The proxies closer to the end-users are also a valid idea and will have to be discussed with the IT team.

7.5 Design Science criteria

Design Science research includes a set of guidelines that were discussed in Section 4.2. This section discusses the guidelines in relation to the finished study and how well the study adheres to the guidelines. Each guideline is discussed separately and the connection to this study is shown. The guidelines are described in A. Hevner and Chatterjee 2010.

Design as an Artifact. The Artifact produced during this study is the Dockerfile defining the build container, as well as the start scripts developed to ease the usage of the build container.

Problem relevance. The problem has been proven relevant to the company, as according to our cost-benefit analysis in Section 7.2 the company saves a considerable amount of work hours with the use of our artifacts. The artifacts also solve the business problems mentioned in Section 5.

Design evaluation. The design artifact has been evaluated in terms of utility, quality, and efficacy with the help of quantitative and qualitative data. The evaluation can be found in Section 7.

Research contributions. Significant contributions were made to the design artifact. The artifacts are well documented and stored in a code repository open to project developers to enable further development. This thesis will also add to the scientific community by evaluating the use of containers for maintaining and distributing a build environment, which previously has not been extensively researched.

Research rigor. The research was conducted with methods for collecting quantitative and qualitative data, such as interviews and test scenarios from which time durations were accurately measured. Backup technologies were properly researched in Section 2.

Design as a search process. We have utilized available means by first conducting a literature review to have a basis for the study, secondly analyzing the department's current situation and problem areas, and finally developing a solution based on the literature to solve the problem areas.

Communication of research. This thesis is the communication of the research from a scientific standpoint. A more practical communication has bees made available to the company, in the form of documentation and instructions on company internal network and direct communication on department discussion channels.

7.6 Thesis process discussion

The thesis process included many phases, such as background research, planning, implementation and reporting of results. Some problems arose in the background research phase, as only a few sources were found for related research. We used some more informal sources, such as blog posts, as related research, but due to their non-academic nature, we did not find much to report. They were however helpful for the implementation of the solution, as some tips and tricks could be found in them.

The planning phase included the interview of five developers to gain an understanding of the initial situation and the needs for our solution. Having more

interviewees would have helped, but due to time limits, we had to settle for five. The solution implementation was also already in progress while the planning was ongoing, leading to some back-tracking in the development when our plans changed. The reporting of results also suffered a little from the small interviewee group, once again due to time constraints. We could also have planned the use of Design Science, introduced in Section 4.2, better. Now it was not a perfect match to our process and could have been more aligned.

8 Conclusions

Software development is a complex challenge, and a single developer or even a team of developers can no longer develop every single line of code when the project grows enough in size. Instead, developers rely on package dependencies, libraries, code from old projects, code from related projects and so on. This of course greatly improves the time it takes to develop the software, but it also creates some issues. Developers now use a significant slice of their time for simply setting up the development environment; installing dependencies, fetching related code and setting up proper tools.

The thesis goal was to investigate the possibility of using containers as a way of sharing an accurately configured development and build environment and was conducted as a case study. The case project consisted of the development of Layer 1 software for 5G base stations, with the team size being close to 200 developers. The project software build environment includes a multitude of software dependencies, foreign code-bases, and tools. The project's development practices include build automation and continuous integration.

The problem identified in our case was the following. As the build environment was a complex system, and our development practices include CI and a large amount of automation, we realized it took a significant effort to maintain an up to date environment in multiple systems. The maintenance responsibility was divided between the CI team, DevOps team, and individual developers. Also, the complexity of the build environment reduced development flexibility in our project, as developers could not easily set up the environment on a machine of their choice.

Containers are a virtualization technology, that is more light-weight compared to traditional Virtual Machines thanks to it using the host system kernel instead of virtualizing it. The most popular container engine, Docker, has made it easy to create shareable container images, with support for versioning and internal distribution tools. As such, it was chosen as the main technology for the implementation part of this thesis.

Our solution includes a Docker image, the source of which is visible to every developer in our project repository. Changes to dependencies can easily be added to the source, after which an image is automatically built and can be used as a basis for the build environment. To ease the use of the Docker container for developers without Docker experience, a start script was developed. An emphasis was placed on the simplicity of the script, as well as the option to customize the start process to the users liking. Also, CI functions were taken into account so that automated commands could be run in the container with the help of the start script.

The results were positive. Comparing a single example of the time it takes for a developer to set up the build environment and build our software with and without the build container showed an improvement of over 6 hours. An interview was also conducted, which gathered info on the developer's opinion on the new containerized build environment. The opinion was mostly possible and the ease of use was complimented. Some problems were also reported by the interviewees, which were related to the access permissions of files mounted to the container.

There is a lot of potential in using containers for the build environment. Future

work in our case includes even more integration with the CI system and our container. For example, we could start using Kubernetes to automatically spin up containers based on our image, which would then be used to run the CI pipelines. This would help with the scalability of the system, as containers could be created on-demand. Other ideas for the future include the usage of more layered Docker images, meaning that we could have a base image that has all the dependencies of our build environment, and build more images on top of it for special use-cases. For example, one could be created for debugging, one for development and one with an emphasis on CI. Another improvement to our system would be easing the use of the container on Windows, however, we have deemed that the I/O performance is not good enough for our use-case on the current Docker for Desktop. One improvement would also be to investigate the use of a rootless container runtime engine, be it the rootless Docker (*Experimenting with Rootless Docker - Tõnis Tiigi - Medium* 2019) or another engine altogether. Finding a suitable solution would help in the case of a server with multiple users, mitigating some security issues created by granting users access to Docker functionality.

Bibliography

5G NR Overall description (2019). ETSI TS 138 300. 3GPP TS 38.300 version 15.6.0 Release 15. 3GPP.

Arnold, P. et al. (2017). "5G radio access network architecture based on flexible functional control / user plane splits". In: *2017 European Conference on Networks and Communications (EuCNC)*, pp. 1–5. DOI: 10.1109/EuCNC.2017.7980777.

Automated software testing in Continuous Integration (CI) and Continuous Delivery (CD) – Continuous Improvement (2019). URL: http://www.pepgotesting.com/continuous-integration/ (visited on 09/12/2019).

Biederman, Eric W and Linux Networx (2006). "Multiple instances of the global linux namespaces". In: *Proceedings of the Linux Symposium*. Vol. 1. Citeseer, pp. 101–112.

Boettiger, Carl (2015). "An Introduction to Docker for Reproducible Research". In: *SIGOPS Oper. Syst. Rev.* 49.1, pp. 71–79. ISSN: 0163-5980. DOI: 10.1145/2723872.2723882. URL: http://doi.acm.org/10.1145/2723872.2723882.

Bui, Thanh (2015). "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967*.

ccache — Compiler cache (2019). URL: https://ccache.dev/ (visited on 10/21/2019).

Cito, J. and H. C. Gall (2016). "Using Docker Containers to Improve Reproducibility in Software Engineering Research". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 906–907.

CMake (2019). URL: https://cmake.org/ (visited on 08/22/2019).

Combe, Theo, Antony Martin, and Roberto Di Pietro (2016). "To Docker or Not to Docker: A Security Perspective." In: *IEEE Cloud Computing* 3.5, pp. 54–62.

Coverity Scan - Static Analysis (2019). URL: https://scan.coverity.com/ (visited on 08/22/2019).

Developing inside a Container using Visual Studio Code Remote Development (2019). URL: https://code.visualstudio.com/docs/remote/containers (visited on 10/31/2019).

Diniz, Paulo SR, Wallace A Martins, and Markus VS Lima (2012). "Block Transceivers: OFDM and Beyond". In: *Synthesis Lectures on Communications* 5.1, pp. 1–206.

Docker Documentation (2019). URL: https://docs.docker.com/engine/reference/run/ (visited on 11/14/2019).

Docker hearts WSL 2 - The Future of Docker Desktop for Windows - Docker Engineering Blog (2019). URL: https://engineering.docker.com/2019/06/docker-hearts-wsl-2/ (visited on 12/03/2019).

Docker Registry - JFrog Artifactory - JFrog Wiki (2019). URL: https://www.jfrog.com/confluence/display/RTF/Docker+Registry (visited on 11/11/2019).

Doody, Owen and Maria Noonan (2013). "Preparing and conducting interviews to collect data". In: *Nurse researcher* 20.5.

Doxygen: Main Page (2019). URL: http://www.doxygen.nl/ (visited on 08/22/2019).

Experimenting with Rootless Docker - Tõnis Tiigi - Medium (2019). URL: https://medium.com/@tonistiigi/experimenting-with-rootless-docker-416c9ad8c0d6 (visited on 10/31/2019).

Fehr, Jörg et al. (2016). "Best practices for replicability, reproducibility and reusability of computer-based experiments exemplified by model reduction software". In: *arXiv preprint arXiv:1607.01191*.

Fowler, Martin and Matthew Foemmel (2006). "Continuous integration". In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122, p. 14.

GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF) (2019). URL: https://gcc.gnu.org/ (visited on 08/22/2019).

Gember-Jacobson, Aaron et al. (2015). "Management plane analytics". In: *Proceedings of the 2015 Internet Measurement Conference*. ACM, pp. 395–408.

Gerrit Code Review | Gerrit Code Review (2019). URL: https://www.gerritcodereview.com/ (visited on 09/17/2019).

Git (2019). URL: https://git-scm.com/ (visited on 08/22/2019).

GitHub - distcc/distcc: distributed builds for C, C++ and Objective C (2019). URL: https://github.com/distcc/distcc (visited on 10/21/2019).

Hevner, Alan R (2007). "A three cycle view of design science research". In: *Scandinavian journal of information systems* 19.2, p. 4.

Hevner, Alan and Samir Chatterjee (2010). *Design research in information systems: theory and practice*. Vol. 22. Springer Science & Business Media.

How to build a project inside a Docker container | Bartosz Mikulski (2019). URL: https://www.mikulskibartosz.name/how-to-build-a-project-inside-a-docker-container/ (visited on 10/31/2019).

Laster, Brent (2016). *Professional Git*. John Wiley & Sons.

make(1) - Linux manual page (2019). URL: http://man7.org/linux/man-pages/man1/make.1.html (visited on 09/11/2019).

Merkel, Dirk (2014). "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

Meyer, M. (2014). "Continuous Integration and Its Tools". In: *IEEE Software* 31.3, pp. 14–16. DOI: 10.1109/MS.2014.58.

Morabito, R., J. Kjällman, and M. Komu (2015). "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering*, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

Nodepool — Nodepool documentation (2019). URL: https://zuul-ci.org/docs/nodepool/ (visited on 10/10/2019).

Nokia (2019). *Nokia, 2018 Annual Report*. URL: https://www.nokia.com/about-us/investors/reports-filings/.

Nokia grabs 40% of phone market for first time • The Register (2019). URL: https://www.theregister.co.uk/2008/01/24/sa_q4_phone_figures/ (visited on 08/30/2019).

Overview of Docker Compose | Docker Documentation (2019). URL: https://docs.docker.com/compose/ (visited on 10/29/2019).

Pahl, Claus (2015). "Containerization and the paas cloud". In: *IEEE Cloud Computing* 2.3, pp. 24–31.

Popek, Gerald J and Robert P Goldberg (1974). "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7, pp. 412–421.

Räbinä, Henry (2019). "Creating a reusable FPGA programming model architecture for 5G layer 1; Uudelleenkäytettävä FPGA:n ohjelmointimalli 5G verkon 1. kerrokselle". en. G2 Pro gradu, diplomityö, pp. 67 + 8. URL: http://urn.fi/URN:NBN:fi:aalto-201908254921.

Smart, John Ferguson (2011). *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. " O'Reilly Media, Inc."

Soltesz, Stephen et al. (2007). "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors". In: *SIGOPS Oper. Syst. Rev.* 41.3, pp. 275–287. ISSN: 0163-5980. DOI: 10.1145/1272998.1273025. URL: http://doi.acm.org/10.1145/1272998.1273025.

The Set Builtin (Bash Reference Manual) (2019). URL: https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html (visited on 11/14/2019).

Use bind mounts | Docker Documentation (2019). URL: https://docs.docker.com/storage/bind-mounts/ (visited on 11/05/2019).

Use volumes | Docker Documentation (2019). URL: https://docs.docker.com/storage/volumes/ (visited on 11/05/2019).

Using Docker Containers As Development Machines - Rate Engineering - Medium (2019). URL: https://medium.com/rate-engineering/using-docker-containers-as-development-machines-4de8199fc662 (visited on 10/31/2019).

Valgrind Home (2019). URL: http://www.valgrind.org/ (visited on 08/22/2019).

WSL 2 is now available in Windows Insiders | Windows Command Line (2019). URL: https://devblogs.microsoft.com/commandline/wsl-2-is-now-available-in-windows-insiders/ (visited on 12/03/2019).

Xavier, Miguel G et al. (2013). "Performance evaluation of container-based virtualization for high performance computing environments". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, pp. 233–240.

Xiang, Wei, Kan Zheng, and Xuemin Sherman Shen (2016). *5G mobile communications*. Springer.

yum - Trac (2019). URL: http://yum.baseurl.org/ (visited on 11/04/2019).

Zuul Concepts — Zuul documentation (2019). URL: https://zuul-ci.org/docs/zuul/user/concepts.html (visited on 09/17/2019).

A Interview outline

1. Interviewee
   (a) What is your role on the project?
   (b) How much experience do you have on:
      i. Mobile networks
      ii. Project (Layer 1 software)
      iii. Project build environment
      iv. Docker and containers
      v. Linux
2. Build environment
   (a) What do you think about the process?
   (b) What is good?
   (c) What is bad?
   (d) How easy?
   (e) How to improve?
   (f) Other thoughts?
3. How do you build the software?
4. Have you manually configured build environment?
   (a) Yes
      i. How was the experience?
      ii. Where/how did you do it?
      iii. Are you using it?
   (b) No
      i. Why not?
5. Explanation of idea for containerized build environment
   (a) What do you think about the idea?
   (b) What is good?
   (c) What is bad?
   (d) How to improve?
6. Free feedback