Tuomas Vase

INTEGRATING DOCKER TO A CONTINUOUS DELIVERY PIPELINE – A PRAGMATIC APPROACH

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2016

ABSTRACT

Vase, Tuomas
Integrating Docker to a Continuous Delivery Pipeline—A Pragmatic Approach
Jyväskylä: University of Jyväskylä, 2016, 66 p.
Information Systems Science, Master's Thesis
Supervisor: Seppänen, Ville

Docker is a lightweight open-platform product that can package an application and its dependencies inside a virtual container; it is also referred to as the container technology. When they are used correctly, these packages can be game changing for IT professionals, as these packages can be easily built, shipped, and run inside distributed environments. The rise of Docker has been a great success, as it has almost become a standard for containers in only three years of its existence, and it works natively with Windows and Linux. This technology offers possibilities for the future of software development and deployment, in terms of a new kind of portability, scalability, speed, delivery, and maintenance.

This study focuses on Docker and continuous delivery at a pragmatic level. The purpose of the study is to design, implement and execute a working Docker-based architecture for the modern type of Continuous Delivery. Docker has been chosen to be the examined container technology as it is the most used and feature-rich technology available. Differences between virtualization and container technologies are examined at a higher level, and the usability and practical usage of container technologies inside a Continuous Delivery pipeline gained a deeper level of examination. The aim of the study is to investigate whether a working model of Continuous Delivery can benefit from the usage of Docker in different situations and, if so, to determine key situations in which a Docker-based solution can enhance the overall Continuous Delivery process.

The most important observation of this study is that Docker can be used in several positions in Continuous Delivery, and it is recommended for use in future systems. However, as the technology is still developing, a better analysis of its usage is still needed in the near future, as many of the mentioned technologies or models are still evolving or even in beta phase. The research was conducted as a Design Science Research Model type of study, including all its phases.

Keywords: Docker, continuous delivery, virtualization, container, container technology, container-based architecture

TIIVISTELMÄ

Vase, Tuomas
Integrating Docker to a Continuous Delivery Pipeline—A Pragmatic Approach
Jyväskylä: Jyväskylän yliopisto, 2016, 66 s.
Tietojärjestelmätiede, Pro gradu -tutkielma
Ohjaaja: Seppänen, Ville

Docker on kevyt avoimen alustan sovellus, joka pystyy pakkaamaan sovelluksen kaikkien tarvittavien riippuvuuksien kanssa yhteen konttiin, ja tätä teknologiaa kutsutaan konttiteknologiaksi. Oikein käytettynä IT-ammattilaiset voivat saada konttiteknologiasta merkittäviä hyötyjä, sillä näitä paketteja voidaan helposti rakentaa, lähettää ja ajaa hajautetuissa järjestelmissä. Dockerin nousu on hämmästyttävää, sillä siitä on tullut konttiteknologian standardi vain kolmessa vuodessa ja se toimii jo nativisti Windowsilla sekä Linuxilla. Tämä teknologia tarjoaa suuria mahdollisuuksia tulevaisuuden ohjelmistokehitykselle sekä käyttöönotolle tarjoamalla uudenlaisia tapoja siirrettävyyden, skaalautuvuuden, nopeuden, jakamisen ja ylläpidon muodossa.

Tämä tutkielma keskittyi käytännöllisellä tasolla Dockeriin ja jatkuvaan toimitukseen. Tutkimuksen tarkoituksena oli suunnitella, toteuttaa ja ottaa käyttöön toimiva Dockeriin pohjautuva arkkitehtuuri, joka mahdollistaa modernin jatkuvan toimituksen. Docker valittiin tutkittavaksi konttiteknologiaksi sillä perusteella, että se on käytetyin konttiteknologia ja sillä on eniten haluttuja ominaisuuksia. Tavanomaisen virtualisoinnin ja konttiteknologian eroja tutkittiin ylätasolla ja perusteellisemmin tutkittiin Dockerin käytettävyyttä sekä käytännöllisyyttä. Pyrkimyksenä oli tutkia, voiko olemassa oleva sekä toimiva jatkuvan toimituksen putkimalli saada hyötyjä Dockeriin pohjautuvista ratkaisuista ja jos voi, niin mitkä ovat avaintekijät tämän prosessin parantamiseksi.

Tärkein löytö tutkimuksessa oli, että Dockeria voidaan käyttää moneen eri vaiheeseen jatkuvan toimituksen putkimallissa ja sitä on suositeltavaa käyttää moderneissa arkkitehtuureissa. Kuitenkin tulee huomioida, että teknologia kehittyy edelleen, joten tarkemmat analyysit ja tutkimukset ovat tarpeellisia lähitulevaisuudessa. Tutkimus tehtiin suunnittelutieteellisellä tutkimusmetodologialla sisältäen kaikki sen vaiheet.

Avainsanat: Docker, jatkuva toimitus, virtualisointi, kontti, konttiteknologia, konttiarkkitehtuuri

FIGURES

TABLES

TABLE OF CONTENTS

1 INTRODUCTION

Virtualization technologies have grown steadily over the last few years to bridge the gap and meet the needs of individuals and enterprises. The existing and steadily increasing demand for services in the cloud needs new solutions; therefore, new methods and types of virtualization have been created to satisfy the needs. (Bui, 2014.)

Virtualization refers to the abstraction of physical computer resources that is aimed at enhancing resource utilization between virtual hosts and providing a unified platform that is integratable for users and applications (Luo, Lin, Chen, Yang, & Chen, 2011). Container technology is a new type of virtualization that essentially delivers software inside containers. Containers are packages that have applications inside of which all the needed dependencies are combined. The use of containers is increasing drastically across the IT industry, from large enterprises to the small start-up firms. (Mouat, 2015b.) In other words, containers have either a platform-as-a-service (PaaS) or software-as-a-service (SaaS) focus with great capabilities for portability and scalability. This type of architecture provides advanced interoperability while it is still utilizing the basic operating system (OS) virtualization principles. (Pahl, 2015.) Container technologies are essentially changing how enterprises develop, run, and deploy software. Software can be built locally, as developers know that the containers will run identically regardless of the host environment. (Mouat, 2015b.)

Although virtual machines (VMs) and containers are virtualization techniques, they are used for different purposes (Pahl, 2015). Containers are not a new concept, as for decades, there has been a simple container type of encapsulation in a form of the *chroot* command, which provides all the same principles as containers (Mouat, 2015b). Although virtualization was introduced in the 1960s with virtual machines in IBM System/360 machines, it was only in 2001 that VMWare introduced its x86 virtualization software to expand the virtualization usage of Linux environments (Fink, 2014).

This study is based on Continuous Delivery and focuses on Docker container technology because it is the most used type of containerization, it is the most feature-rich container technology, and most of the published scientific ar-

ticles focus on it ("Docker," 2016). In addition, the technology provides native support for Windows- and Linux-based operating systems and other available integration tools.

Docker is an open-source software, which is an extension of the existing Linux-container technology in multiple ways. This technology mostly aims for user-friendliness in creating and publishing containers. (Jaramillo, Nguyen, & Smart, 2016; Mouat, 2015b.) It enables a consistent method of automating the faster deployment of software inside highly portable containers (Bernstein, 2014). With containers, applications share the same operating system and, whenever possible, libraries and binaries. This combination allows that these deployments are minimal in size compared to the traditional virtualization alternatives, thus making it possible to store hundreds to thousands of containers on a single physical host (Bernstein, 2014).

This study is motivated by the observation that many large organizations have already transferred their solutions to container technologies; thus, an investigation of selected software company's current Continuous Delivery (CD) pipeline versus a Docker-based CD solution is an interesting topic, especially when stability, performance, usability, and maintenance are measured against each other.

The aim of the current study is to investigate whether Docker can solve problems of the current CD pipeline in a software firm with designed architecture and to determine specific CD areas that need Docker the most to simplify the overall CD process. These areas are examined through a literature review, design and development of a working solution, hands-on experimentation, and propositions for best practices. This study has had some challenges in terms of finding enough relevant academic papers on Docker, as this technology is only three years old. Another problem has been that even the practical side of Docker is incomplete because many features or technologies involved are still in the alpha or beta phase and are thus not yet recommended for production use.

This study revealed that when properly used, Docker can enhance and simplify the overall CD process in several fields. Therefore, the use of Docker or alternative technologies is recommended when making a new and modern CD pipeline process.

1.1 Client and task of the thesis

This study is conducted as a full-time contract for Landis+Gyr, which is multinational company that operates in more than 30 countries. The task is to investigate and execute a working solution or proof of concept for CD purposes with Docker. Landis+Gyr has already been working with the CD pipeline but is willing to examine a more modern and enhanced version. The proposed solution can deliver enchantments for the CD pipeline and its overall process in multiple ways.

1.2 Research problem, research questions and limitations

Docker is a virtualization framework that runs applications and is not for emulating hardware, which underlines the main difference between OS-level virtualization software such as Docker and machine-level virtualization (Fink, 2014). However, using Docker may be complex because Docker runs applications by default in the foreground, which necessitates the conversion of common programs (Fink, 2014). Docker's focus of one service per container might also be problematic, although there are many advantages in this same principle. As Dua, Raja and Kakadia (2014) stated that containers have some congenital benefits over virtual machines due to refinements in performance and reduced start-up time. Docker is a solution that is not only lightweight, but it can be launched in a sub-second on top of an operating system with a hypervisor, which allows for much scalability (Anderson, 2015).

As Docker delivers many attributes, e.g., performance, stability, and scalability that fits extremely well in CD, these were also desired attributes for Landis+Gyr. The problem was that the old CD pipeline had problems in builds reliability, stability, automation, and speed; consequently, a faster-paced and more trustworthy solution was needed. Since Docker can also be used for other purposes than building and managing software, the overall process enhancement with Docker also became a requirement. Regarding the aforementioned possible benefits and requirements, the research first focuses on the following questions:

- How can Docker enhance a working CD pipeline?
- What benefits are there in Docker-based CD?

Software nowadays mainly controls critical infrastructure; thus, securing the chain of software production is an evolving and rising concern (Bradley, Fehnker, & Huuck, 2011). All stages of software deployment can be attacked or corrupted along the software deployment pipeline, and other vulnerabilities may occur when software is being integrated with other infrastructures (Bass, Holz, Rimba, Tran, & Zhu, 2015). Attackers are interested in finding new ways to exploit any detected weaknesses or other vulnerabilities in software systems to get financial benefits or to only cause harm (Axelrod, 2014). However, secured container technologies may deliver a solution for these concerns. Therefore, the third research question is:

- What security problems are associated with using Docker, and how can these be mitigated or avoided?

1.3 Research method and data acquisition

This study is conducted using a Design Science Research Method (DSRM). Scientific literature for the study was mostly acquired from ACM Digital Library, Google Scholar, IEEE Xplore, ProQuest and Springer Link databases and other reliable sources. The relevance of the used references and articles was weighted by the number of citations, and newer articles were weighted more valuable than older ones. This study mostly uses articles that were published from 2014 to 2016. References were searched in the aforementioned databases using relevant keywords. These keyword combinations were executed from the following words: Docker, virtualization, container, container-technology, continuous delivery, and security. Interviews and user opinions were used for sub-chapter Evaluation of the implementation to get more reliable metrics and feedback than those from only comparing build times or calculating resource allocation.

The DSRM (Figure 1) is a research method that attempts to create artifacts that serve human needs instead of trying to understand reality. It is divided into six iterative sections: problem identification, objective definition, design and development of artifact, artifact demonstration, result evaluation and communication. (Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007). As the DSRM can have different research entry points, this study focuses on the design- and development-centered approach because Docker based CD needs much practical design and implementation to find a working solution for production purposes.

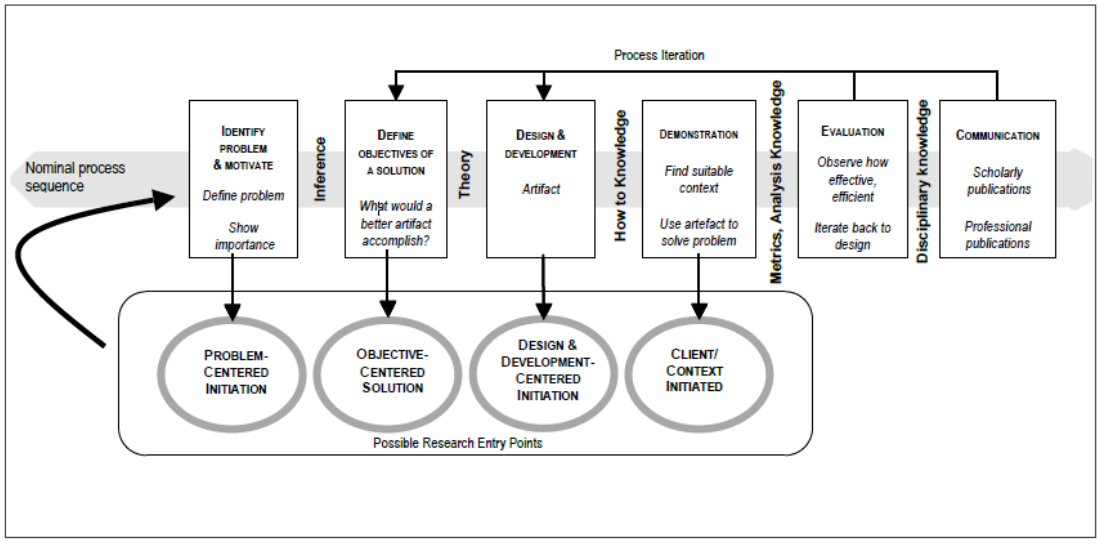

Figure 1 - Design Science Research Method (Peffers et al., 2007)

1.4 Research structure

The structure of this study follows the stages of DSRM. Whereas DSRM has six iterative process steps, this study is divided slightly differently, because some sections fit together, and some are better when divided. The DSRM steps are presented in following order with brief explanations:

- **Identify problem and motivate**
  The first chapter is the introduction to the study, in which the main points of container technology and a basic explanation of virtualization with a motivation are provided, followed by the research problem, the method with data acquisition and conventions used in this study.

- **Define objectives of a solution**
  The second chapter explains the main concepts of container-based virtualization and how CD and Docker fit together; it describes the main principles of how Docker containers work and how Docker should be used. This chapter also contains the definitions of used terminology, other container technologies and the security aspect of container technologies.

- **Design and development**
  The third chapter is the first of two main chapters, and it describes the process of decision making when designing a modern CD pipeline that is based on Docker. The main architectural decisions, charts and argumentation behind the decisions are demonstrated and explained in this chapter.

  The fourth chapter is the second main chapter of this study, and it demonstrates the overall process of developing a working CD pipeline with Docker. This chapter contains several examples and used code and configuration files.

- **Demonstration and evaluation**
  The fifth chapter describes how the implemented artifact has been demonstrated for Landis+Gyr, how it has been tested and how the Docker-based CD
  System can be used.

  The evaluation will focus on how the implemented solution has enhanced the existing system. This chapter has a tabular comparison of the old and new systems and the overall evaluation of how the solution fulfills system requirements. The researcher critically evaluates the solution and justifies decisions that have been made when artifact was designed and built.

- **Communication**
  The sixth chapter discusses previous studies on the current topic, findings of the current study and possible future studies. This chapter also concludes on the entire study.

1.5 Backbone theories of DSRM

Since this study utilizes the DSRM process, it involves to some extent the theories that are the backbone of the methodology creation, as is uses the same or evolved principles of DSRM. The evolution has come from far as the first mention of Design Science was in 1969: "Whereas natural sciences and social sciences try to understand reality, design science attempts to create things that serve human purposes" (Simon, 1969, p. 55). Another mentionable theory is defined information systems design theory by Walls, Widmeyer, and El Sawy (2004), which was used for creating the DSRM process (Peffers et al., 2007). However, Hevner, March, Park, and Ram (2004) provided the last piece of the puzzle, in Design Research in Information Systems Research, by providing seven guidelines that describe the characteristics of credible and well-performed research. From these seven guidelines, the most important part is that research must produce an artifact that focuses on solving a problem. (Peffers et al., 2007). Thus, as this study designs and develops an artifact in an iterative manner, these guidelines and iterative steps greatly assist in the process of making this study and making an artifact for human purposes.

1.6 Conventions used in this study

The following typographical conventions are used in this study:

Italic for describing a command or configuration variable,

`Courier` for Docker based commands and

```
OpenDocument text objects with real syntax
```

for long commands and Dockerfiles.

2 DOCKER, CONTINUOUS DELIVERY, AND CONTAINERS

The final artifact of Docker-based CD system is based on Docker and its features; thus, the literature must be examined in such detail that the objectives for the artifact are clear. As DSRM involves designing part in the process and iterates until the artifact is satisfactory, a literature review is also needed to gather existing information that the new solution will use for, e.g., the best practices, architectural design, or other mandatory knowledge to avoid pitfalls when the artifact is being developed. (Peffers et al., 2007).

2.1 Virtualization technologies

The two main categories of virtualization, namely hypervisor-based virtualization, and container-based virtualization, contain almost every virtualization technology available (Boettiger, 2015). While containers provide OS-level virtualization, hypervisor-based virtualization is more at the hardware level. In virtual-machine-based virtualization (Figure 2), host machines and their resources are controlled by a hypervisor, e.g., Virtualbox and VMware Workstation ("Oracle VirtualBox," 2016; "VMware Workstation," 2016). This type of virtualization can share, e.g., memory resources across memory limits as most processes do not consume all their allocated memory. Operation-level virtualization, such as Docker and container-based virtualization, can do this even better because it has a refined method for resource sharing. Container technology brings multiple isolated instances available with desired properties for the user and makes the managing and generating software processes more user-friendly. Thus, container-based virtualization is more modern than traditional virtualization as it has better usability and diversified the resource utilization; therefore, it minimizes the overhead of creating new virtual processes. (Adufu, Jieun, & Yoonhee, 2015)

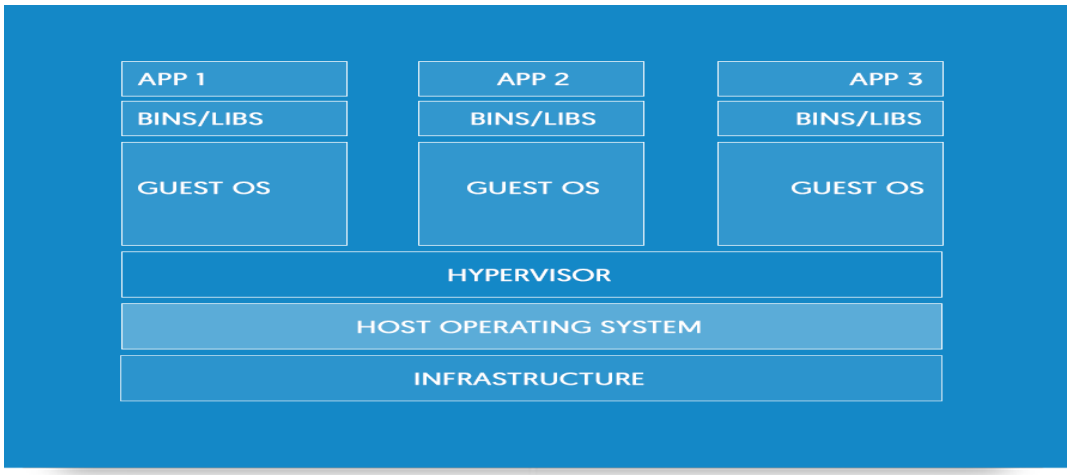

Figure 2 – Traditional virtualization ("What is Docker?," 2015)

Figure 2 illustrates three separate applications that each run a VM on a specified host. The hypervisor in this virtualization method is needed for access control and for system calls whenever necessary. Each of these virtualized hosts needs a full copy of the OS, the application or applications being run, and the needed libraries. In Figure 3, the same three applications are running inside a container-based system. The main difference is that the kernel of the host is shared among the running containers, which basically means that containers always have a constraint against a used kernel as it must be same as the host. In a container-based system, data and libraries can also be shared, as containers do not use identical copies of these data sets. The Docker engine stops and starts containers quite similarly to how a hypervisor does. However, the processes inside containers are identical to the host's native processes. Thus, Docker will not generate any overhead. (Mouat, 2015b).

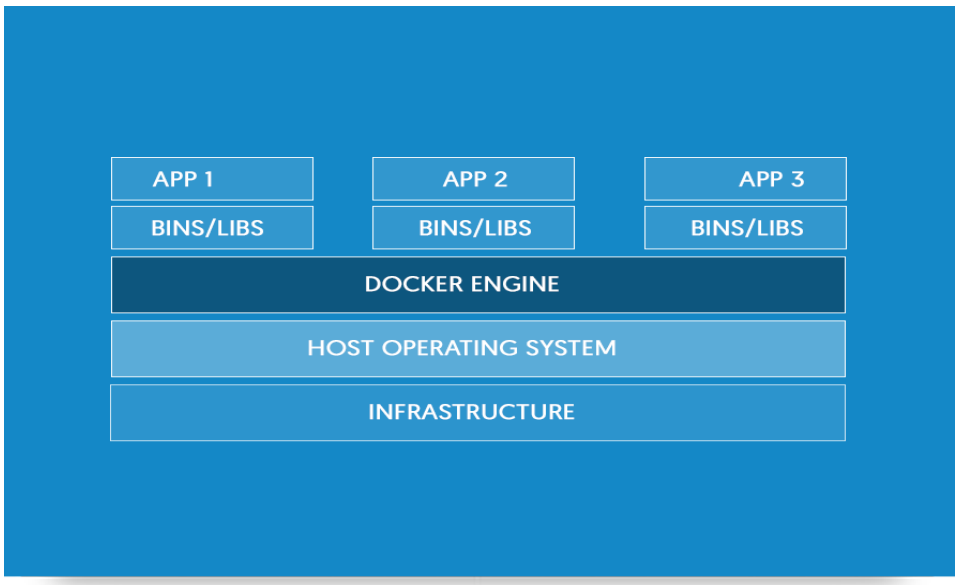

Figure 3 - Docker Containers ("What is Docker?," 2015)

Containers and hypervisor-based virtualization can be used for the isolation of applications from each other that are on the same host (Mouat, 2015b). Virtual machines have a little more isolation from the hypervisor, and by being a more mature technology, hypervisor-based virtualization is identified as more trusted and battle-hardened technology (Manu, Patel, Akhtar, Agrawal, & Murthy, 2016). Containers are somewhat new, and many organizations doubt completely trusting the isolation of containers before they have a proven track record. Thus, it is still common to find hybrid environments with containers that run inside virtual machines to take advantage of both technologies. (Matthias & Kane, 2015).

2.2 History of Docker and containerization

One might presume that containers are new, which is not the case. Containers have been around already for many years in Linux distributions but only rarely used because of the complexity of building a container that would work. Everything started when Solomon Hykes, the CEO of dotCloud, had a talk at Python Developers Conference in California on March 15, 2013. Around that time, only 40 people had tried the first version of Docker. After the surprising amount of press coverage, the project was open sourced to GitHub to provide wide access for anyone who would like to contribute. (Matthias & Kane, 2015). Docker had reached popularity earlier on as, for example, Amazon Web Services announced in 2014 that they were widely supporting applications that were containerized by Docker (Linthicum, 2014).

Docker has created a kind of Renaissance for Linux containers by enabling waves of interest and possibilities, which have led to rapid technology adaptation. Docker started the container revolution, as containers were previously considered to be complex constructs. Containers are now seen as a solution for almost every software design. However, Docker is not the perfect answer for every design as some of the most eager people might say, but it delivers the concept of tools that get the job done while allowing their parts to be changed to provide customized solutions. (Matthias & Kane, 2015).

Nowadays, "containerized" or "containerization" is a common term in the software industry. The term is borrowed from shipping containers, which use the same principle as software containers: ship and store all kinds of cargo in standardized units. Docker containers provide the same generic method for isolating processes from the host and from each other. (Dua et al., 2014). However, even when Docker is referred to as the "de facto standard of containers," the standardization of containerization still lacks some features. According to Dua et al. (2014), the features are as follows:

- Standardization: A real standard for container file format is missing, which is a necessity for full interoperability.

- Security: Containers need a secure method of isolating networking and memory.
- Independence of the operating system: A level of abstraction is still needed as there should be no restrictions to using a specific kernel or user space.

2.3 Continuous delivery and continuous integration

Software firms' highest priority is to satisfy customer needs with continuous delivery of software (Fowler & Highsmith, 2001). Another main principle in companies is to stay ahead of competition. Thus, companies that can deliver their products faster and more reliably remain in the market (Schermann, Cito, Leitner, & Gall, 2016). Continuous delivery is the capability to make quick, safe, and sustainable software deliveries to production from all kinds of software changes, whether it is a bug fix, experiment, new feature, or configuration change. Studies have demonstrated that deploying software more frequently makes the overall performance of software delivery better, faster, and reliable. This is possible when a code is always maintained at a deployable state. (Humble & Farley, 2010).

Humble (2010) states that the reason behind continuous delivery is to gain several important benefits for a software-based business, as continuous delivery enables for several factors:

- Low-risk releases: It is possible to achieve almost zero downtime deployments by demand when, e.g., blue–green deployments are used.
- Higher quality: By building an automated deployment pipeline with continuous regression, performance, security testing and other delivery-process activities can be performed to ensure quality.
- Better products: Continuous delivery is an enabler for quality because working in small quantities is economical. Thus, small parts can get a greater amount of user feedback from multiple deliveries, and features that do not deliver value can be easily avoided.
- Faster time to market: When build and deployment are automated with all quality assurance, spending weeks or months in the test/fix phase is not necessary anymore. Consequently, deliveries are quicker to market.
- Lower costs: Software products tend to evolve during development. An automated continuous delivery pipeline drastically reduces incremental changes and fixed costs.
- Happier teams: Continuous delivery makes less painful releases and thus reduces software teams' burnout. Frequent releases enable a team to communicate more actively with users; thus, it is possible to see beforehand ideas that work.

The basic idea of continuous integration is that developers integrate their code to a trunk, master, or mainline branch multiple times a day to enable constant testing and, thus, seeing how a change works or affects other changes. This allows the "fail fast" principle to see the possible issues early on as builds will fail fast. In continuous integration, most of the testing is done automatically with some unit test framework. Generally, this is done in a build server that performs all the merged tests, so developers do other work while the tests are running. (Fowler & Foemmel, 2006).

According to Wolf and Yoon (2016), modern continuous delivery is a combination of many available tools and technologies that ensure high-quality products for production. However, to achieve a desired level of quality and to remain efficient, the testing must be automated. The mentioned testing phases include sets of user interface tests, integration tests, load tests and unit tests. Thus, Wolf and Yoon (2016) stated that they used modern technologies, e.g., Gerrit, Jenkins and Docker, to perform automated testing for every release to ensure rapid iterations and to ease complex cluster configurations. Dhakate and Godbole (2015) also stated that Docker is suitable for continuous integration and continuous delivery, since production environment replicas can be easily made in local computers. Thus, developers can test their changes in a matter of seconds. Cloning is also faster with containers because an exact replica of a container takes few seconds, whereas the full clone of VM takes minutes. Changes to images of the containers can be made rapidly as only needed sections are updated after a desired change.

A CD pipeline that uses containerized software can also lead to new testing environments, as, for example, containers can be disabled or disconnected quickly to determine surviving capabilities of the system. The container grid does not at all increase the build times; instead, it delivers efficacy, resilience and stability to the production. (Holub, 2015).

2.4 Alternative container technologies

Nowadays, multiples of different container solutions are available. As this study purely focuses on Docker containers due to the needed features and native support in both Linux and Windows, only two alternatives are briefly described: Rocket (rkt) and Linux containers (LXC). The rkt alternative has a status as a competitor for Docker, and LXC is the antecedent of Docker. Windows containers are also demonstrated briefly in a separate section, because they differ slightly from the Linux version of Docker.

2.4.1 Rocket – project rkt

Project rkt, which is now called Rocket, started soon after Docker Inc. had adjusted its orientation and roadmap to not only make standardized containers

but to also focus on building tools and features as full platform products ("CoreOS rkt," 2016). In 2013, Docker even deleted their published container manifesto; this is exactly when Rocket was born to develop its own standardized container system (Polvi, 2014). The main purpose of Rocket is to create a specified container model that also supports different images. The Rocket technology itself is an option for Docker containers, because it has advanced security than that of Docker and necessary production features. Another reason is that from the security point of view, Docker does not deliver as much certainty as Rocket, as every Docker process runs via a daemon, which is bad as daemon failures can lead to broken containers. Rocket's container runtime pursues the application container specification, which sets up a decent number of different formats that can be easily transferred. (Kozhirbayev & Sinnott, 2017).

Docker and Rocket both introduce a model of automated application deployment, which can start virtual containers independently based on the server type. However, Rocket focuses on an application-container specification that allows more diverse migration of containers. Thus, Rocket is more suitable for performing simple functions with advanced security, whereas Docker delivers a more diverse environment to support large-scale requirements and operations. Rocket can be more difficult to use because it stays in command-line-based environments, whereas Docker simplifies the whole process of building containers through its descriptive interface. (Kozhirbayev & Sinnott, 2017). Rocket describes and solves important problems in the container field, but, as previously described, Docker and Rocket solves different problems. Thus, Rocket is excluded from this study because it is not designed to solve the problems that are required by the requirements.

2.4.2 Linux containers

The LXC is an antecedent of Docker ("LXC," 2016). The LXC is a container-based virtualization technology that is compatible with flexible or common API. The LXC and Docker share multiple features but also have differences. (Kozhirbayev & Sinnott, 2017). The LXC uses namespaces and control groups (cgroups) the same way Docker does to guarantee the isolation of containers. Linux containers initially used the process identification (PID) and network namespacing. The LXC also developed the method of resource sharing and management via cgroups. (Xavier, Neves, & Rose, 2014). From the different features' perspective, LXC containers have high flexibility and performance, and they can be snapshot, cloned and backed up, which can lead to the illusion that there is a virtual machine or another server in the background (Kozhirbayev & Sinnott, 2017). Dua et al. (2014) compared Docker and the LXC, and the only difference was in container lifecycle. Docker uses its daemon and a client to manage containers, whereas the LXC uses different tools, e.g., *lxc-create* or *lxc-stop* to create or stop a container.

2.4.3 Windows containers

At end of September 2016, Microsoft released the new Windows Server 2016 edition that has, for the first time, native support for Docker containers. The most interesting part is that Windows containers are not Linux-based, but instead based on Windows kernel that is running Windows inside Windows, which is an entirely new solution. (Sheldon, 2016). Currently, Microsoft provides two types of images for Windows containers that are Server Core and Nano Server. As expected, the latter one is a smaller version of Windows, whereas Server Core provides an "everything you would need" solution.

Windows containers share many common principles with the Linux version of Docker. Windows containers are designed to isolate environments and to run applications without harming the rest of the system while delivering needed dependencies to make the container environment fully functional. The techniques that are used to do this are also similar, as they can be seen in Figure 4. (Sheldon, 2016).

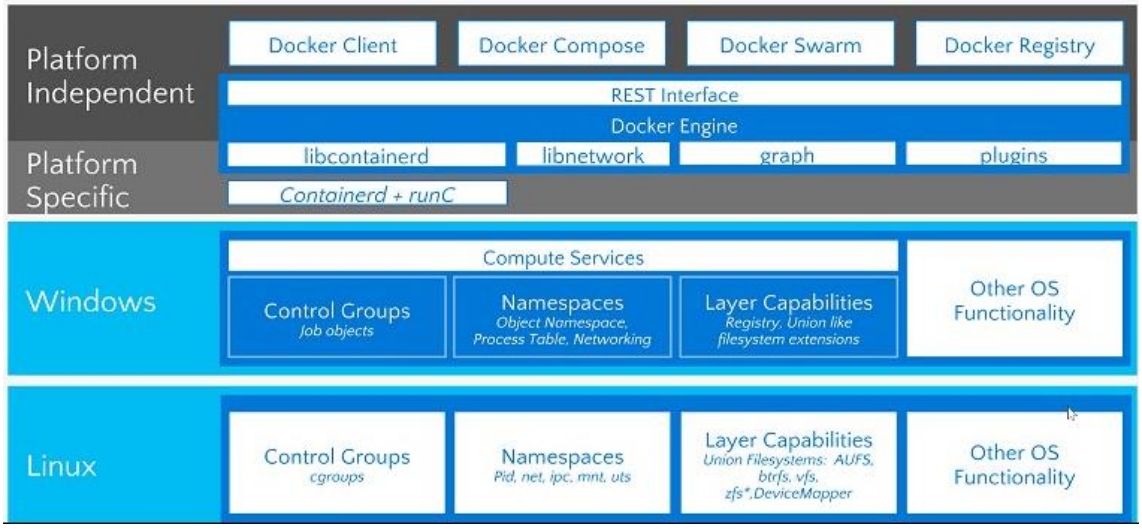

Figure 4 - Linux and Windows comparison ("Docker Windows Containers," 2015)

Microsoft currently delivers two types of containers in the Windows Server 2016 edition: Hyper-V and Windows Server containers. These containers work basically the same way, except that they provide different levels of isolation. Windows Server containers share the kernel with the underlying operating system the same way Docker does on the Linux side. Each of the containers has its own view of the operating system, IP address, file system and other needed components, and all the listed attributes have similarities in isolation in terms of namespaces, process, or other resource-controlling utilities. However, as Windows Server containers depend on the kernel, the operating system's level patches and their dependencies also affect the containers, thus it can be hard to notice and can cause complicated issues. (Sheldon, 2016).

By contrast, Hyper-V containers are slightly different, as they provide a minimal virtual machine between the host and a container. Thus, the aforemen-

tioned patching or other operating system dependencies do not affect a Hyper-V container as it is more isolated. However, such isolation slightly affects negatively the performance and efficiency of the container. The solution regarding performance and efficiency is providing more secure and stable containers. (Sheldon, 2016). The Hyper-V version of Docker is capable of running both Windows containers on top of the aforementioned small virtual machine, but this version is also capable of running LXCs, as demonstrated in Figure 5.

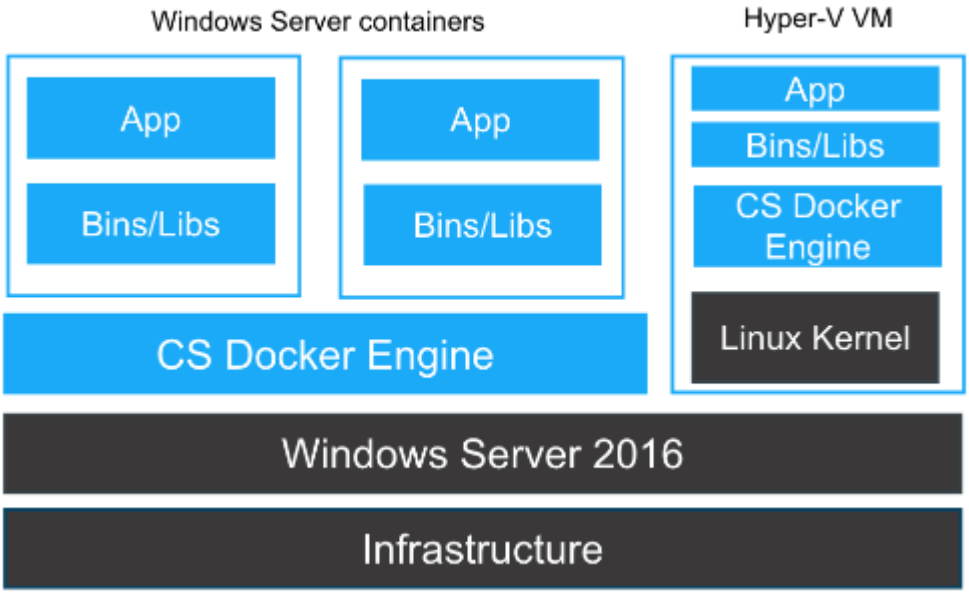

Figure 5 - Windows Containers ("Docker Windows Containers," 2015)

Since Windows Server 2016 can run both types of containers natively on Windows or a small overhead on top of a virtual machine, it is now possible to run the containers with Windows technologies such as PowerShell. Windows Server 2016 also has nested virtualization that was not available in previous server editions, as now Hyper-V containers can run even if the host is running inside a virtual machine. However, as Windows Server 2016 natively supports Docker, Windows Server 2016 does not contain Docker when installing a new server. Thus, to get Docker up and running, the Docker engine must be downloaded and configured separately from Windows. Subsequently, the Docker engine runs as a Windows service. (Sheldon, 2016).

Windows Server 2016 provides something new to the container world; however, there is still much to be done before Docker and Windows are fully compatible. (Sheldon, 2016). Problematic areas are that there is no cross-platform containerization, as different kernels are not suitable for each other and need a hypervisor between them. Although there is a desire to make the integration as smooth as possible, a common kernel for both types of containerization is anticipated to be in the distant future, if it will ever happen. This also applies to Dockerfiles and their creation syntax because containers created by PowerShell cannot be used in a Linux kernel, and vice versa. However, it is now

possible to get the same benefits from containers on top of Windows, which is an improvement from Microsoft's side (Sheldon, 2016).

2.4.4 Summary of alternative containers

As Docker's main rival, Rocket focuses on application-container specification that allows more diverse migration of containers, whereas Docker delivers a more diverse environment to support large-scale requirements and operations. Docker and Rocket solve different problems, but Rocket has excluded from this study because it does not have the required tools or features. The antecedent of Docker, the LXC, is similar to Docker, but it lacks the needed features and is not as simple to use. Therefore, the LXC has also been left out of this study. Windows-based Docker is new in the container world, but it is still a new technology integration and thus needs some time to develop. The Windows-based Docker solution was searched in all available materials, but none of the available academic articles concern Docker. Therefore, Windows containers are only mentioned as a future possibility. Consequently, the Windows-based Docker solution was partly excluded because of the lack of reference materials. Based on these limitations and constraints, the only fully examined container technology is Docker itself.

2.5 Docker containers

Docker is open-source software that is an extension of the existing technology of Linux containers in multiply ways, mostly by delivering a complete solution with user-friendliness in mind to create and publish containers (Mouat, 2015b). Docker has features such as Git type of versioning, makefile syntax in Dockerfiles, and it also delivers performance by sharing binaries and libraries whenever possible (Boettiger, 2015). As containers consist of all the needed dependencies, containers enable a consistent way to automate the faster deployment of software inside highly portable containers (Bernstein, 2014). With containers, applications share an operating system's kernel and thus allow these deployments to be minimal in size compared to traditional virtualization alternatives to make storing hundreds to thousands of containers on a single physical host possible (Bernstein, 2014). Docker is widely used, since the majority of large cloud providers provide support for Docker or use it themselves. Example of large cloud providers are Amazon AWS, IBM Cloud, Microsoft Azure and many more. Many users of the most popular applications, such as Netflix or Spotify, also use Docker because of its scalability options. In 2015, Google also announced that it would be using Docker as the primary container format. (Matthias & Kane, 2015).

2.5.1 Docker in practice

Docker basically extends the LXC application and kernel-level API, which both provide process isolation (Bernstein, 2014). The two main parts of isolation that Docker uses are namespaces and control cgroups, which are part of the Linux kernel technology (Anderson, 2015). The third vital part is the layer-based Union File System (UFS), in which layers are on top of each other and only the last layer is read-writeable. Docker uses cgroups to manage the host's resources by controlling CPU, memory, block I/O and network usage. Namespaces are used to isolate a container from the host and from other containers. Namespaces control processes, networks, file systems, hostnames, user IDs and the underlying operating system. (Anderson, 2015). All Docker-based actions, such as running or building containers, are tied to a Docker engine, which is the main controller of all processes. However, Docker containers do not run on top of the engine but on top of Docker's daemon (Mouat, 2015b).

Docker containers are created from images; in other words, an image is created from a Dockerfile, and a running image is a container. A Docker image can consist of only the minimum requirements of an application, a fully functioning operating system or it can have a pre-built application stack for production purposes. (Bernstein, 2014). Docker by default uses a build cache, which ensures that there are no unnecessary builds if there are no changes (Anderson, 2015). In a Docker build process, each command or action in the Dockerfile, for example *"apt-get install"*, creates a new read-only layer on top of the previous one, which is made possible by the UFS. All commands in a Dockerfile are automatically executed, and the commands can also be used manually (Bernstein, 2014).

The difference between containers and virtual machines is that containers share the host kernel; thus, Docker images also share the kernel, and the image cannot be used in different kernels. Using the kernel natively allows Docker to use fewer resources than virtual machines. This enables that instead of unnecessary overhead generated by virtual machines, multiple containers run in the same host with a higher performance that is attractive for the software industry, which has resulted in container technology's popularity. (Boettiger, 2015). However, the popularity would have been much less if there had not been available repositories for the images. The main repository is called a Docker hub, and it can be used as a storage for public and private Docker images. The hub can be used from a browser by searching for desired images or by using the Docker search command. (Bui, 2014). It is also possible for one to run private image repositories inside Docker hosts with one's own frontends for better security.

The automation tool for building images is Dockerfiles (Anderson, 2015). Dockerfiles execute Docker commands in a desired order for creating images, and they have similarities with the Makefile syntax. Currently, there are 18 different instruction commands available in Dockerfiles. Each usage of such instruction creates a new layer on top of another, e.g., using

```
RUN apt-get update && apt-get install application
```

creates only one layer, whereas

```
RUN apt-get update
RUN apt-get install application
```

creates two layers with one on top of the other. As Docker uses build cache by default, only changed parts are built if Dockerfile is changed. However, if a previous command has changed, all commands after it are also built. (Karle, 2015).

2.5.2 What Docker is not

Docker is suitable for multiple different scenarios and can be used to solve various problems; however, its feature-centric focus means that Docker lacks specific functionalities in features. Companies used to think that they could remove all configuration management tools when migrating to Docker; however, they have now realized that the power of Docker is not in pure functionality but in being a segment or part of the final solution. (Matthias & Kane, 2015). To fully realize the "dos and don'ts", Table 1 presents results by Matthias and Kane (2015) that indicates what Docker does not deliver.

Table 1 – What Docker is not

Function	**Reason**
Virtualization Platform	A container is not a virtual machine, as it does not contain a complete OS running on top of the host OS.
Cloud Platform	A container workflow has similarities to cloud platform, as it responds to demand by scaling horizontally. Docker can only deploy, run or manage containers on existing Docker hosts but cannot create new host systems.
Configuration Management	Dockerfiles manage containers at build time, but they cannot be used for containers' ongoing state or to manage the host system.

2.5.3 The architecture of Docker

The basic architecture of Docker is that it is a simple client/server model that runs inside one executable (Jaramillo et al., 2016). Under the simple exterior, many complex processes function, such as various file system drivers, virtual bridging, cgroups, namespaces and other kernel mechanisms. (Matthias & Kane, 2015). As mentioned earlier, namespaces and cgroups are the core isolation of container technology, and the third vital building block is the UFS.

Namespaces provide the desired isolation that containers only see by default containers own environment, and if linked or exposed, wanted environments. This ensures that Docker containers do not affect other containers or the

host environment. Namespaces also guarantee that containers have restricted access to the file system and do not give containers any rights that are above the container level. Networking for containers are also handled by namespaces, by giving own virtual network adapters for containers to enable a unique IP address and hostname for each container. (Joy, 2015)

Almost every container technology uses cgroups to secure the balanced utilization of resources. Control groups are vital for containers to work because they ensure the allocation of resources to each container and prevent the overusage of given resources. (Dua et al., 2014).

The last key element that enables Docker is the UFS, which allows multiple file systems to be overlaid and thus appear as a single file system. Docker images are made from multiple layers. Every layer is a read-only file system, and each of these layers is included in each instruction that is given in Dockerfiles. Each new layer sits on top of previous layers, and the last layer appears when a Docker image is started as a container when the Docker engine adds a read/write file system on top of all previous layers and adds other settings such as name, ID, resource limits and IP address. (Mouat, 2015b). In addition, it is possible to share the common files and libraries when necessary in the UFS. Images can use pre-existing binaries and libraries when image is being built, which essentially lowers the build time and need for space while still utilizing portability. (Haydel et al., 2015).

Docker has multiple available storage drivers for UFS, which are mostly system dependent. The file system can also be changed if desired, which must be done with care to prevent file corruption. (Mouat, 2015b). One of the most used file system layers is the Advanced multi-layered unification filesystem (AUFS), but its downside is a hard limit of 127 layers. Most Dockerfiles try to minimize the number of used layers when making Dockerfiles. Thus, this can be seen when multiple commands appear in one single line such as the *RUN* or similar instruction. (Mouat, 2015b).

2.5.4 The usability of Docker

One of the effects of usability of containers is the ability to outperform and exclude traditional virtualization. In most cases, containers provide a better performing solution without unnecessary overhead for scaling; consequently, Docker and its alternatives are now gaining popularity. Many of cloud providers have changed to using Docker, as Docker can be launched quickly and saves resources. (Joy, 2015). Another main problem that can be solved with Docker is the so-called "dependency hell," which is one of the largest problems in software development and deployment (Merkel, 2014). By providing an environment of all needed software dependencies that go together with the software, Docker provides a solution that does not get broken elsewhere but works in every Docker host provided. Dependencies can be controlled by using different build images in the deployment pipeline, which ensures that there are no serious outages from an otherwise working code. (Boettiger, 2015). Software firms

can also fight against the so-called "code-rot" using Docker's feature of image versioning.

Joy (2015) described five top values of Docker in terms of usability:

- Portable deployments
  Containers have applications that are built inside a container, thus making a container extremely portable. Moving the unit or bundles does not affect containers in any means.
- Rapid delivery
  Since Docker containers always work the same way, software team's members can focus only on their own tasks: Developers can focus on the containers and the built code, and operations can focus on fully functioning container deployments.
- Scalability
  As Docker containers run natively in Linux and Windows, they can also be deployed to various cloud services or other hosts. Docker containers can be moved from cloud to a desktop and back to cloud in seconds because they have a unified container format. These containers can also be scaled from one to hundreds and back to one using orchestration software.
- Faster build times
  As containers are by default designed to be small, their build times are short, and they affect the speed to get feedback from testing, development and deployment. An automated build pipeline allows the same container to be used for testing and later be moved to production environments.
- Better performance and higher density
  Docker containers do not use a hypervisor; thus, all left-over resources from less overhead can be used more efficiently. This means that a single host can have multiple containers instead of few virtual machines, thus gaining better performance.

In other words, Docker is a bridge between operations and developers. Docker enables developers to use whatever new technologies they want, as a technology's final format is always the same as that of a container, which results in much more trustworthy deployments. (Joy, 2015).

2.5.5 Performance of Docker

Docker and virtualization technologies are here to stay. All the major cloud providers, e.g., Amazon AWS, Microsoft Azure, and Google Compute Engine, use virtualization to drive their powerful and scalable solutions. However, as mentioned previously, virtual-machine-based virtualization has its own disadvantages in terms of overhead; thus, better performing solutions are required. (Joy, 2015). Using containers in production is not anything new as, e.g., Google, IBM and Joyent have all successfully used containers as backbones of their clouds, instead of using traditional virtualization. (Bernstein, 2014).

In Table 2, Dua et al. (2014, p. 610) compare containers and virtual machines.

Table 2 – Container and VM comparison (Dua et al., 2014)

Parameter	**Containers**	**Virtual Machines**
Guest OS	Containers an OS and kernel. The image of the kernel is loaded into the physical memory.	All VMs run on virtual hardware, and a kernel is loaded into its own memory region.
Communication	Standard IPC mechanism such as signals, pipes, sockets, etc.	Through Ethernet devices
Security	Mandatory access control can be leveraged.	Depends on the implementation of a hypervisor
Performance	Containers provide near native performance	Virtual Machines suffer from a small overhead.
Isolation	Sub-directories can be transparently mounted and shared.	Sharing libraries, files between guests and between hosts is not possible.
Startup time	Containers can be booted up in a few seconds	Virtual Machines take a few minutes to boot up.
Storage	Containers take less amount of storage.	Much more storage usage is required as the whole OS kernel and its associated programs must be installed and run.

Docker performs well in higher data volumes. In Many Task Computing (MTC) and High Throughput Computing (HTC), billions of ongoing tasks depend on computing power and management of data. Both MTC and HTC require a solution to manage high data volumes and fast and reliable access to memory. Container-based solutions, such as Docker, substantially outperformed traditional virtualization in a recent study on described requirements of execution times and memory management. (Adufu et al., 2015). In another study, in which Docker was compared with a bare metal server, there were roughly no overheads on CPU or memory utilization, whereas I/O and OS interaction caused some overhead. In these cases, the mentioned overhead appeared as additional cycles for every I/O operation. Thus, applications inside containers that have an increased amount of I/O operations can perform more poorly than applications that do not need so much I/O to operate. (Kozhirbayev & Sinnott, 2017). Furthermore, when Docker containers and virtual machines were compared against a bare metal, Docker containers were significantly less affected in terms of performance loss (Gerlach, Tang, Wilke, Olson & Meyer, 2015).

In an apples-to-apples benchmark test, a container-based IBM Softlayer was five times better performing than the virtual-machine-based Amazon AWS (Bernstein, 2014). Finally, when kernel-based VMs (KVMs), which are a mature technology and have had time to evolve, and Docker were both tuned to maximum performance settings: Docker exceeded or equaled KVMs in every aspect. (Felter, Ferreira, Rajamony, & Rubio, 2015; Higgins, Holmes, & Venters, 2015).

The container-based approach is suitable when the simplest solution for application deployment is desired. This is one of the main reasons why cloud vendors have moved to using containers instead of virtual machines, as simplified solutions are more appealing, they use less resources, and they are also cheaper for the customer. (Bernstein, 2014). Docker performs well in these areas, and it can be compared against an operating system that is running on a bare metal (Preeth, Mulerickal, Paul, & Sastri, 2015).

Although all the aforementioned tests give an impression of container superiority in every imaginable use case, there are still use cases in which virtual machines do well. Hypervisor-based virtualization is better when applications require different operating systems with multiple versions in the same cloud and when building a software monolith makes sense. (Bernstein, 2014; Chung, Quang-Hung, Nguyen, & Thoai, 2016).

2.5.6 Security of Docker

Software security is a challenge whenever services are running in virtual environments. A common statement is that container-based virtualization, such as Docker, is less secure than the hypervisor type of virtualization. Thus, using Docker raises security concerns. (Bui, 2014). However, there are solutions and principles for providing secure Docker, for example, using the architectural segregation, such as namespaces and cgroups, and new innovations correctly. According to Combe, Martin and Di Pietro (2016), Docker security basically relies on three factors: network operations security, user space managed by the Docker daemon and the vigil of this isolation by the kernel. Docker has also stated as a company that its mission is to provide highest level of security without losing any usability. (Diogo, 2015). Thus, using Docker safely is a certain possibility.

Security of the software raises concerns, for example, when a question was posed to IT professionals about their greatest concern, the response was "someone subverting our deployment pipeline" (Bass et al., 2015). Nowadays, everything works at a faster pace, so does software development. At such a rapid pace of development, more errors are bound to occur even with the most cautious development personnel. Thus, software security must be included at all levels that are possible. (Bradley et al., 2011). Docker containers use the same kernel as the host; therefore, it is possible for one to gain unauthorized access to a container if security is not at the required level or users are not aware of possible and potential risks. Thus, securing a Docker container is mostly about monitoring and limiting the possible attack surface. (Mouat, 2015a).

Namespaces provide the first level of security through container isolation, which prevents processes from seeing or connecting to another container. The second type of segregation is cgroups, which limit the available resources for containers, e.g., that a Distributed Denial-of-Service attack cannot deplete other resources (Petazzoni, 2013).

Docker is fairly secure and has default configurations (Chelladhurai, Chelliah, & Kumar, 2016). However, there are still multiple problems, as Bui (2014) and Chelladhurai et al. (2016) stated that Address resolution protocol (ARP) spoofing and Media access control (MAC) flooding attacks can be done against a default networking model of Docker because of its vulnerability. However, the aforementioned attacks can be mitigated or totally avoided if a networking administrator is aware of the vulnerability and has correct filtering and configurations on the host.

In autumn 2015, Docker introduced the Docker Content Trust (DCT), which provides public key infrastructure (PKI) for Docker images. This allows more secure remote repositories for Docker images because PKI provides a root key (which is offline) and a tagging key (which is per repository) that are generated when an image is created for the first time. The DCT adds a specified signature on all data that are sent, i.e. it constantly verifies the originality of the desired image when downloading. All the keys are unique and per repository, which means that security is not breached if one key is revealed (Vaughan-Nichols, 2015).

To better grasp examples of possible exploits, Mouat (2015a) described several possible exploits of Docker that are combined in Table 3.

Table 3 – Docker security exploits (Mouat, 2015a)

Exploit	**Explanation**
Kernel exploit	Kernel is shared with all containers. Therefore, one malicious container can crash a kernel and all the containers connected to it.
Denial-of-service attacks	All kernel resources are shared among containers. If configuration of cgroups is poorly executed, one noxious container can starve all other containers.
Escapes from a container	Gaining access from one container to another or to the host itself should not be possible.
Poisoned images	Without trusted repositories, it is impossible to know what image containers and thus to know if the image is safe.
Secret compromises	Containers that are connected to, e.g., a database will most likely require a secret to access it, such as a password.

The aforementioned exploits or given online articles about Docker security can lead to security concerns. However, while working with Docker containers needs special care and knowledge to achieve security, it is not too complicated

to do so. Nowadays, all fields of software development need to know the potential security issues; thus, these issues are not limited to Docker only. Docker can also be used more efficiently and has better security than bare-metal servers or virtual machines alone (Mouat, 2015a). A correct configuration and awareness can drastically reduce the issues of Docker attack surface. In addition, one potential security method is to install Docker inside a virtual machine, but it is not the right solution as it only creates a more complicated system. Thus, the complexity will add more attack surface (Hemphill, 2015).

The most important thing to achieving security in software development is to know and understand the fact that breaches happen. In such cases, it is necessary to understand how the attack or breach has been made and how it can be avoided in the future (Hemphill, 2015). As Docker is rapid in multiple ways, security patches can be applied quickly and precisely on the targeted place (Mouat, 2015a).

3 DESIGNING A ROBUST SOLUTION

This chapter contains the reasoning and investigation behind the created Docker-based CD system. As Landis+Gyr already has a working CD pipeline, the proposed architectural decisions are based on that pipeline. This means that there are limitations, constraints, and requirements for the architectural design, because a new solution must integrate flawlessly into an existing one. The design is based on resilient principles as it delivers a robust solution that does not fail if unexpected events occur but is instead forgiving and adaptable (Matthias & Kane, 2015).

3.1 Requirements and constraints for design

As mentioned earlier, other alternative container technologies than Docker were ruled out due to several reasons: First, Docker is already used in Landis+Gyr at some level; thus, continuing to make use of the existing container technology makes sense. Second, as Docker is the only container technology that natively supports Windows and Linux, it is a mandatory requirement to stay with Docker, because there are existing Windows- and Linux-based environments. Third, Docker is the reference for containers and the DevOps ecosystem; for example, in a recent survey, 92% of people are planning to use Docker or are already using it (Combe et al., 2016). Fourth, as Docker is the most used container-based and feature-rich technology, it is justified to only use Docker as the first requirement.

The second major requirement by the Landis+Gyr was to use the existing continuous-integration platform called Jenkins, which is one of the largest open-source automation servers available ("Jenkins," 2016). As the current Jenkins master server is responsible for the current production and controls all the "to be containerized" build environments, it is a mandatory requirement to keep the existing system and integrate the new solution based into it. As com-

panies want to keep their code only on premises, cloud-based solutions are also left out.

The third major requirement is to deliver a real deliverable or artifact in terms of proof of concept or a production-viable solution. Thus, this study is significantly contributing on the practical level of study and by demonstrating the created code whenever possible. Sub-requirements for the solution are to follow the available best practices, to keep the solution reproducible, to achieve all the mentioned benefits from Docker and to provide a secure solution. Thus, the following sections justify the reasoning behind the decisions made to achieve desired stability, performance, usability, maintenance, and security.

The fourth requirement is to make the solution as light as possible to save valuable computing resources. Each virtual machine consumes much disk space, and licensing costs tend to be per used CPU or per used megabyte on a disk. Thus, a lightweight solution that consumes fewer resources brings monthly savings.

The fifth requirement is to examine "out of the box" concepts that are not part of the proposed CD solution but can be achieved with Docker to enhance the overall usability and development practices in the company. Lastly, all used components must be open source. Thus, the proposed solution does not use anything else.

3.2 Current continuous delivery system

The current solution follows basic principles of continuous integration and CD because every code change is merged to the master branch after verification, and the changes are built and tested multiple times per day before continuous release of software. This basic functionality can be seen in Figure 6, in which a code change is first pushed to Gerrit, which is a code-reviewing software, for peer review; if the code is valid, it is submitted to the master Git branch, which is a version-control system ("Gerrit Code Review," 2016, "Git," 2016). Jenkins polls the master branch for changes and launches a dedicated job if changes occur. The build itself and all iterative steps of continuous integration are then executed on a dedicated build server, which is a dedicated virtual machine in this case. After execution, the results appear in Jenkins logs, and the cycle can either continue on the same server or be pushed to other servers.

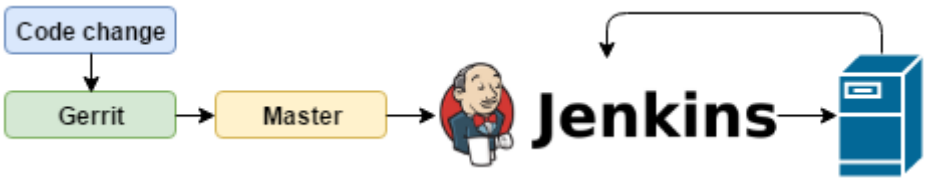

Figure 6 - Basic Build Job

This basic solution is simple to use but not efficient. As every build has its own dedicated hypervisor-based virtualization servers, they are only used when builds or tests are being executed. Thus, servers are frequently run idle and reserve valuable computing resources. As build servers are pre-installed with needed dependencies for builds, they have a large number of dependencies of different software projects; consequently, there are many overlaps of such dependencies. Thus, when a dependency update is needed, it is tricky to conduct as the update can create unexpected behavior for the build environment and the builds. Another problem is that some builds have extremely long build times, and if a build fails, it cannot recover fully to the exact spot where it failed. Thus, when these overnight builds fail, their failures get noticed from the morning office hours.

As the literature has demonstrated, Docker provides a perfect answer to these problems: Docker uses fewer resources that it uses only when they are needed, dependencies can be isolated inside containers, and build environments can be easily replicated if unexpected behavior occurs. Thus, building a Docker-based CD pipeline is justified.

3.3 Proposed solution for implementation

Docker reduces system deployment time and enables system automation deployment; hence, focus can be on system planning, design, and practice (Wang, Cheng, Chen & Chien, 2015). The proposed solution contains multiple components, which are described later in this section. The requirements and limitations strongly affect the design, as, e.g., cloud-based solutions could not be used and the existing software is a dependency that dictated suitable technologies. Basically, the proposed solution avoids or mitigates all the disadvantages of the old CD solution in terms of robustness, performance, and maintainability within the frame of requirements and constraints. The basic Job Workflow of the designed Docker-based CD system can be seen in Figure 7.

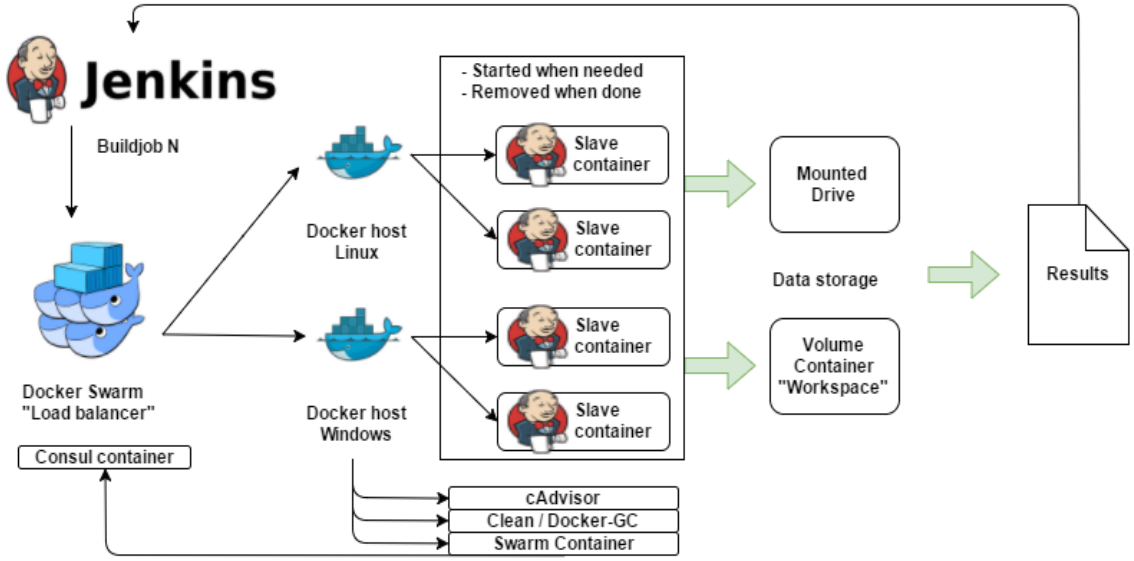

Figure 7 - Basic Job Workflow

The build process starts the same way as that of the old solution, as a dedicated build job starts when Jenkins notices a new change in the code base. Instead of direct connection from Jenkins to host, Jenkins communicates through Docker Swarm, which divides the build workload ("Docker Swarm," 2016). By doing this, Swarm can divide the workload between multiple different Docker hosts depending on what is being built, e.g., Linux- or Windows-based software. When the host has been selected for workload, the Docker host checks if there are any available Jenkins Slave containers that can run the build. By default, there is none available as the system is configured such that these build containers are only started when they are needed. Slave containers are in this case pre-made Docker images that have all the needed dependencies to build the desired software. The build itself normally runs inside the container, but the advantage is that the container has all the required dependencies and nothing more or less. By having all needed dependencies, the builds are more robust and maintainable because each build environment can be specifically maintained, and the changes cannot at all affect other build containers. When the build has finished the tasks, the results or builds are either pushed to a volume container or a mounted drive, depending on the build environment, and then pushed back to the Jenkins master server.

Figure 7 illustrates that each Docker host has three other containers and the Swarm container is connected to the Consul container. Google's cAdvisor container is used for monitoring, Spotify's Docker-GC is used for cleaning the host, and the Swarm container is responsible for maintaining communication to the Docker Swarm master container ("Google cAdvisor," 2016, "Spotify Docker-GC," 2016). The Consul container between Swarm containers is for service discovery; it keeps the IP-addresses and health-status data of Docker hosts, which are sent from Swarm containers ("Consul.io," 2016).

3.4 Overall architecture

To put things into perspective, Figure 8 demonstrates the overall architecture. Each of the described components, except Jenkins, runs inside Docker containers. The basic workflow is the same as that in Figure 7, but the other parts are discussed thoroughly in this section.

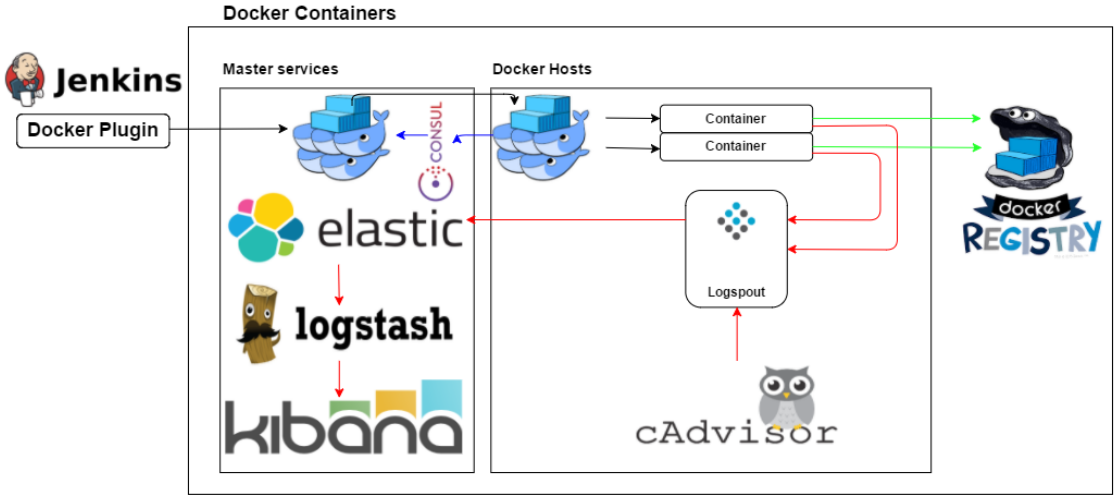

Figure 8 - Proposed overall architecture

First, to make the integration simple, a Jenkins Docker Plugin is used for easy configuration and connection to the Swarm ("Docker Plugin," 2016). The architecture of the solution is divided into master services and slave services, both of which are Docker hosts. The master services are Swarm manager, Consul, and Elastic Stack. This means that in principle, only Docker hosts that are used for building are scaling, but master services are located only in master Docker servers. However, every master server also has the same contents as the Docker host, as master servers are also Docker hosts but are only responsible for running the master services. The Elastic Stack is responsible for keeping precious logs in Elastic Search and Logstash and showing them in the Kibana user interface ("Elastic Stack," 2016). Every host has a Logspout container, which is responsible for fetching all logs from running containers and putting them to the Elastic Stacks safe remote location ("Glider Labs Logspout," 2016). Google's cAdvisor container also runs per host to deliver the needed monitoring of each host. Finally, there is a Docker Private Registry for storing created Docker images in the company's premises ("Docker Registry," 2016).

3.4.1 Docker plugin

The first step of choosing the correct plugin is to examine all plugins that could be suitable for Docker + Jenkins synergy. The requirement is to find a suitable candidate that can be used for production and launch Jenkins slaves inside a

container at a remote host. Currently, multiple Docker-based Jenkins plugins are available:

- CloudBees Docker pipeline
- Docker Commons plugin
- Yet Another Docker plugin
- CloudBees Docker Build and Publish plugin
- CloudBees Docker Custom Build Environment plugin
- Docker plugin
- Docker Slave plugin

Each of these plugins is suitable for different tasks. However, the only suitable candidate from the preceding list is the Docker plugin, as it is the only one that fulfills all the requirements. The Docker plugin can create a "Docker cloud", which is a cluster of Docker hosts that are managed by Docker Swarm. Each Docker host can have N number of containers, which can be configured in Jenkins. Basically, the plugin asks for, e.g., a name, host URL, timeouts, and container capacity.

The Docker plugin simplifies the configuration of new Jenkins jobs and the work of the build team when setting up new Docker host servers. The aim of the Docker plugin is to provide balanced provisioning of build slaves, which makes the build environment scalable. Docker can be used without the plugin, but it is not possible to use a clustered solution through Docker Swarm; instead, Docker without the plugin works as the old method per build environment server. Basically, the only changing part for the user is the build job's label inside each build job, which was previously used to dictate the build environment. Now, the label dictates the used Docker image, in which a Jenkins slave is being run. For example, label "Java-Slave" could contain Java JDK and other needed dependencies to build Java-specified code, whereas "Python-Slave" could contain python alternatives.

3.4.2 Docker Swarm

To achieve a robust build environment, an orchestration or clustering solution is needed to handle the build load. Thus, investigation and experimentation on clustering solutions are required for building this ecosystem. Clusters are also referred to as Containers-as-a-Service (CaaS), as in them, e.g., Docker Swarm builds, ships and distributes containers without interoperability issues on multiple hosts in multiple clouds (Naik, 2016).

As there are multiple orchestration tools and clustering solutions, only few candidates have been selected for further investigation. For this purpose, the following candidates have been selected because of their popularity: Docker Swarm, Docker Swarm mode, Kontena, Kubernetes and Mesos ("Apache Mesos," 2016; "Google Kubernetes," 2016).

Kubernetes and Mesos have ruled out quite quickly because Kubernetes are too complex, and Mesos are more suitable for larger server clusters and do

not deliver real advantages after the release of Docker 1.9 (Farcic, 2015; Hall, 2016). To demonstrate dimensions of such clustering solutions, Google's Kubernetes can be configured to also orchestrate Docker containers that run on Mesos (Pahl & Lee, 2015; Pahl, 2015). Kontena is highly promising as it delivers many of attributes that are desired for this type of build environment ("Kontena," 2016). However, as Kontena is still in the beta phase and is not compatible with the Docker plugin, it has also been left out. The two candidates that have remained are the native solutions of Docker, as Docker Swarm and the Swarm mode have same API as the Docker engine itself. Thus, to make the solution as simple and reproducible as possible, only the native candidates were considered to be suitable. With Docker engine version 1.12, it is possible to use the engine's own Swarm mode to natively manage a cluster of Docker engines. Swarm mode has all the features and functions of the standalone Swarm, but it is simpler to use. However, the Swarm mode is so new that the Docker plugin unfortunately does not support it yet. Thus, the Swarm mode has an "on hold" status until a new version of Docker plugin is released. Until then, the standalone Swarm will be used for the solution.

The first beta of Docker Swarm was released in early 2015. The idea of the Swarm is to present a single interface for a Docker client, although there can be a whole cluster of Docker hosts instead of a single daemon behind the interface. Swarm is used for clustering computation resources in more complex systems (Matthias & Kane, 2015). Thus, it can be used to provide desired properties for the build cluster. Docker Swarm delivers workload balancing for containers as it automatically chooses the best host for building by checking the resource usage while maintaining the required performance levels (Naik, 2016). Therefore, using Swarm meets the requirement of saving resources as build loads are distributed evenly based on available resources.

After each code change or period, Jenkins starts a build job based on the change. This triggers a request for a new slave from Docker Swarm, which manages all available Docker hosts. The aforementioned label determines the host, e.g., Windows-based containers can only be run with a Windows host. If there are not any dependency issues with hosts, Docker Swarm provides the host with the least amount of work. Each Docker host can contain as many containers as a user chooses or based on resources available, e.g., 100 containers per host.

3.4.3 Consul container

Standalone Docker Swarm does not have service discovery or a key value store, whereas Swarm mode has it natively. A new solution needs to have this component. The three most popular candidates were etcd, ZooKeeper and the previously mentioned Consul ("Apache ZooKeeper," 2016; "Consul.io," 2016; "CoreOS etcd," 2016). Although these candidates are suitable for the current task, Consul has been chosen for simplicity over etcd and ZooKeeper that only have a key-value store in primitive stage and require their own service discovery

methods to be implemented. Another reason for choosing Consul is that this candidate is more popular and has more than 15 million downloads in the Docker hub compared to 1 million of etcd and Zookeeper ("Docker Hub," 2016). If Swarm mode becomes usable in the future, the Consul can be removed as it will not be needed then. Until then, Consul is needed especially for maintaining hosts' IP addresses and for ensuring service discovery.

Consul makes the solution highly available by delivering a distributed service-discovery tool for enabling rapid deployments for massive scales. Consul is also used for health checking, as it constantly checks the Docker host's availability status. Consul also works as a key-value store in which multiple types of information, e.g., configurations or coordination, can be saved. Consul itself has a high availability system by the gossip protocol, which is a communication protocol from computer to computer. This means that if a Consul master becomes unavailable, the existing Consul followers automatically elect which node becomes a new master to maintain the availability.

3.4.4 Elastic Stack, Logspout and cAdvisor

Working with containers brings completely new problems in terms of monitoring and logging. As containers are meant to be ephemeral, they go up and down extremely quickly and thus make it much harder to monitor what is going on in containers than in other hosts. Logspout is used to get the logs to live elsewhere rather than getting deleted inside a container. Logspout is a log router that runs inside Docker hosts and sends valuable logs to a remote location. Tagging the containers makes it possible to monitor the logs as sets instead of individual containers, which is more reasonable in the container world (Young, 2016). However, Logspout is not mandatory to use nowadays because Docker provides native solutions for logging drivers that can send logs to a remote location ("Docker logging," 2016). Although the logging drivers can be used for simplicity, they are not part of the solution, as then every container should be started with the following syntax and by longer runtime commands the solution would be much more complex.

```
docker run --log-driver=syslog \
--log-opt syslog-address=tcp://0.0.0.0:123
```

The selected application for the remote logging system is the Elastic Stack (previously ELK-Stack) that, which combines three technologies: Elasticsearch, Logstash and Kibana. Elasticsearch is mainly a distributed search and analytics engine that can be easily managed; it is reliable and horizontally scalable. Logstash is a dynamic data store that has great synergy with Elasticsearch. Finally, Kibana is a highly customizable visualization tool for the container data. Altogether, the Elastic stack delivers desired attributes for saving logs to a remote location and for visualizing them to a browser's user interface. Other logging tools can also be used instead of the Elastic Stack, but using the Elastic

Stack is desirable because it is already used in Landis+Gyr, is scalable and easy to set up. Logspout can send container logs in multiple forms, thus making it is easy to change the monitoring tool if necessary.

The Elastic Stack has solved the problem of logging longevity, but real-time monitoring is still required. For this purpose, Google's cAdvisor is a perfect fit because it is extremely easy to set up and is a lightweight monitoring tool. Currently, Google's cAdvisor is one of the most popular tools, as indicated by more than 300 million downloads from the Docker Hub. Google's cAdvisor's user interface illustrates multiple types graphs and indicators of current resource distribution and usage, such as CPU or RAM usage and running containers. However, cAdvisor lacks some necessary features of a robust system because it does not alert if, e.g., resource usage is too high. Thus, in the future, cAdvisor should be changed to a preferable real-time monitoring system, such as Prometheus, to have such alerts in place ("Prometheus," 2016). Since cAdvisor is more than enough at this point, it will be used until a change is needed.

3.4.5 Docker private registry

Docker images are ordinarily stored in repositories that are called Docker registries (Jaramillo et al., 2016). For better security and by obeying the requirements, the created containers must remain on the company's premises. Thus, pushing company containers to public registries must be prohibited, and pulling the containers from public hubs must be done with care to avoid malicious images. To meet these conditions, the company's own private registry is required for distributing the created images inside the company. Such a registry allows controlling the location of these images and the full possession of the distribution pipeline. The candidates for image repositories are only the Docker Hub, Docker Private Registry and Docker Trusted Registry (DTR) ("Docker Trusted Registry," 2016). As the Docker Hub has been rejected due to the on-premises requirement, only the private registry and DTR is examined. However, although one might presume that the DTR is more secure than the private registry, this is not the case. The only big difference between these two is that the DTR is commercially supported; thus, to set up a registry and maintain it, real-time support can be achieved from Docker's employees. In the proposed solution, Docker Private Registry has been chosen because setting up the registry is not too hard. In addition, the private registry is well secured with Rivest-Shamir-Adleman cryptosystem (RSA) keypairs. Thus, it can be safely used for image repository inside the company.

3.5 Provisioning and secure connections

To make the solution more secure, each connection in the system uses either RSA keypairs or Transport Layer Security (TLS) connection. To demonstrate

this, Figure 9 illustrates how the provision of workload is made between Docker hosts. First, when Docker Swarm provisions a suitable host, the connection is made through TLS. When the Slave container has been started, the Swarm does not carry the messages between Jenkins and Slave container, but instead the container is talking to Jenkins host directly through Secure Shell (SSH) connection with RSA key pairing. In RSA key pairs, there are two keys: the private key, which must be kept secret, and the public key, which can be used more freely.

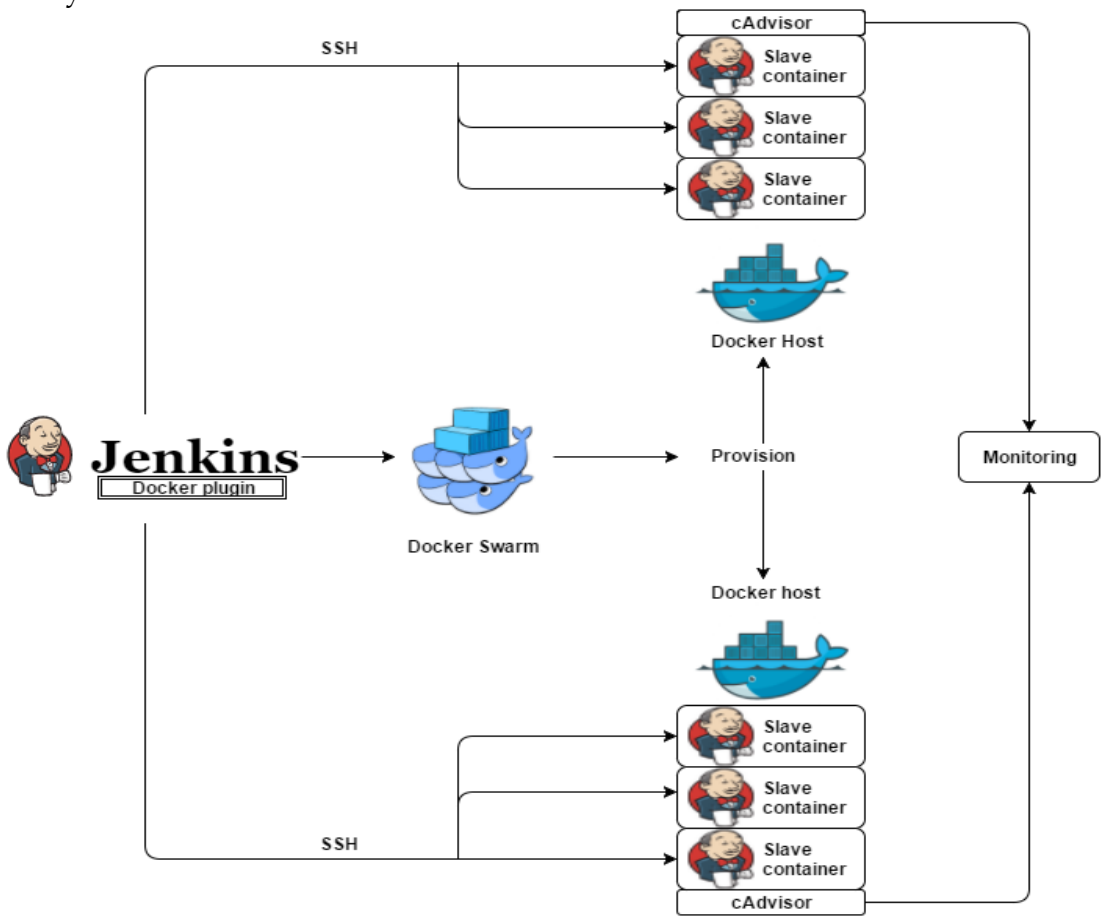

Figure 9 - Provisioning and SSH connection

To make RSA keypairs as secure as possible, they are generated with strong cryptography, and each keypair is used only for one task. This means that when connecting from the host to a Docker registry, from Jenkins to a Slave or from the Slave to the Git master branch, each connection has different keypairs, as this can be seen in Figure 10. However, the Slave container's private key is not put in place automatically, but it is injected from Jenkins credentials plugin whenever needed.

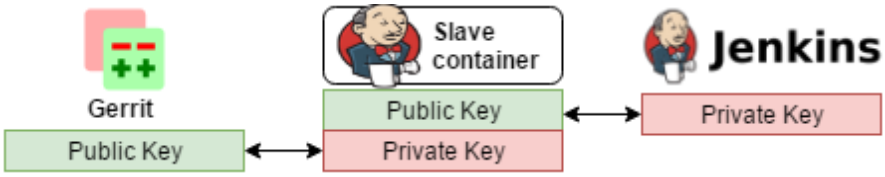

Figure 10 - RSA key locations

3.6 Designing a secure architecture for containers

To build a secure solution, the best practices for security and available tools have been examined. One of the tools is Docker Bench security, which is a script that checks common security best practices that are based on Center of Internet Security (CIS) benchmarks ("Docker Security Bench," 2016). The final version of the solution should be evaluated with the aforementioned script to ensure the Docker environment's security. Furthermore, the following are high-level picks for better security (CIS Docker 1.12 benchmark, 2016) that should be used when implementation of the system is started:

- The Kernel should always be updated and, if possible, run together with grsecurity and PaX.
- The used host should only run Docker-based features, and Docker should always be up to date.
- Every Docker host must be audited at a required level, and only trusted users should have access to the system.
- Network between containers should be restricted to avoid unwanted disclosure to other containers.
- Insecure or public registries should be avoided to prevent malicious images.
- Docker daemon should have TLS authentication to avoid unwanted usage.
- Logs should be centralized in a remote location.
- Live restore should be used to achieve greater availability if Docker daemon terminates.
- Use appropriate number of Docker Swarm managers to avoid possible compromises.
- Verify that used files have appropriate file permission levels.
- Verify that used certificates are in a safe location.
- Create users in Dockerfiles to exclude root-level usage in containers.
- Use trusted base images in Dockerfiles to avoid malicious images.
- Slim down the containers by using only needed dependencies.
- Use *HEALTHCHECK* instruction inside containers to get container's reported health status.
- Use combined commands in a single instruction whenever possible to minimize the number of layers.
- Dockerfiles should only use *COPY* instead of *ADD* to avoid downloading malicious files from URLs.

- Dockerfiles should not contain secrets, as they can be compromised.
- Containers should not use the privileged mode, as containers can then do almost everything that the Docker host can do.
- Containers should not use SSH if possible, as it makes security management complex.
- Expose only the ports that are needed to isolate a container as much as possible.
- Use memory and CPU limit tools to avoid over-usage of given resources.
- Containers should use only *--restart=on-failure:5* to avoid restart loops.
- Basic "docker0" bridge network should be avoided, as it is vulnerable to ARP spoofing and MAC flooding.

3.7 Best practices for creating containers

Docker containers are based on Docker images, which are created from Dockerfiles (Matthias & Kane, 2015). Thus, to make robust containers, it is crucial to master the Dockerfile syntax. As containers are meant to be transient, the Dockerfile should be made as ephemeral as possible. This means that a container can be stopped and destroyed without problems, and a new one can be built and started with minimal configuration. Dockerfiles should also have separations, such as one for building and one for distribution, that are based on the build. Thus, it is possible to create as light and clean containers as possible (Holub, 2015).

Writing a Dockerfile is easy; however, writing a robust and transient Dockerfile is much trickier, as a lot of decisions must be made. The first decision is, e.g., whether the new image is based on an old one; if so, what kind of an old image? This leads to other questions about the old image, such as whether is it safe or it remains the same forever, and what binaries and libraries the image has by default. Thus, mastering a Dockerfile can be hard, even with easy instructions and commands.

Each used Dockerfile instruction creates a new layer, and all the layers are images. Thus, each created image is a set of previously made images. To demonstrate this, when using following instructions separately:

```
RUN apt-get update && apt-get install -y curl
```

and

```
RUN  rm -rf /var/lib/apt/lists/*,
```

"*rm -rf*" does not shrink the image size but uses an unnecessary layer to bloat the image, as the packages are in the previous layer, and the deletion does not affect them. Thus, many Dockerfiles use only a single *RUN* instruction when installing packages and cleaning the cache immediately thereafter to make the image as small as possible. It is crucial to always remember to combine *apt-get update* and *apt-get install* in same *RUN* statement. If this has not been done, it is

possible that the next time the same image is built, the *apt-get update* is cached and does not run, which can potentially cause harm. Furthermore, only running *apt-get* for packages without the version tag always uses the latest versions, which can also cause unintended upgrades and thus unnecessary problems. Therefore,

```
RUN apt-get update && apt-get install -y package-foo=1.0.*
```

can be forced to use a specific version and remove the dependency issue. This can also force the builds to retrieve a specific version, regardless of the status in the cache. To iterate and highlight the commonly used best practices, they are written below based on Benevides (2016) and Docker ("Dockerfile Best Practices," 2016).

- Write transient Dockerfiles
  As containers are meant to be transient, the Dockerfile should be made as ephemeral as possible. This means that a container can be stopped and destroyed without problems, and a new one can be built and started with minimal configuration.
- Use a .dockerignore file if needed
  The best practice is to put the Dockerfile to an empty folder and then add the only needed files for building. Excluding is possible by using .dockerignorefile in the same folder and uses the same syntax for excluding as .gitignore files.
- Add only one process per container
  It is much easier to scale horizontally and reuse the same containers if it is written as a single process. If a service is dependent on another, it can be linked to another container.
- Do not install unnecessary packages
  To reduce build time, dependencies and complexity, all unnecessary packages should be avoided, e.g., installing a text editor to a production container to minimize the image size.
- Minimize the number of layers and sort them
  As AUFS storage driver, for example, has the hard limit of 127 layers, and each layer makes a bit of an overhead, minimizing the use of new layers in the *RUN* statement or similar should be done. Multiple lines should be written alphanumerically to spot duplicates, and use of the backslash (\) increases readability.
- Use a build cache if possible
  Each instruction in Dockerfiles has its own cache; thus, Docker examines if it can reuse the existing cache rather than creating a duplicate image. However, if one section of the cache has changes, all the later instructions are built again. Another notable observation is that files that have been added with the *COPY* or *ADD* command are calculated by checksum if the file has changed. With the *--no-cache=true* parameter, this option can be disabled.
- Avoid storing data in containers

As containers are meant to be ephemeral, data should not be stored inside containers. However, if data must be stored, it should be stored inside a volume container or mounted directly to disk.

- Avoid shipping in multiple pieces
  To achieve CD, Quality Assurance and production best practices, an application should be built in the shipped image instead of deploying it to running ones.
- Do not use commit to creating images
  Docker command *"docker commit"* can be used for creating images from running images, so it should not be used. The traceability and version control of such images is nonexistent, and the images can contain malicious software.
- Avoid using the latest tag for images
  By default, the images are created with the latest tag if not described otherwise. Thus, a update for such an image can remove a working version of the same image with the latest tag, so always use the *-tag imagename:version* syntax.
- Credentials do not belong to the image
  Use environmental variables or key-value stores instead of credentials to achieve this. Another solution is to use a secret inside the image when necessary.

3.8 The future of Docker containers

One of the latest breaking news about containers is that official Docker images switch from Ubuntu to Alpine Linux (Christner, 2016). The reason behind this is that Alpine Linux is a security-oriented, extremely lightweight distribution. In addition, when compared to other distributions, the Alpine Linux image is only 4.8 MB in size, whereas the Ubuntu official Docker image is 188 megabytes ("Alpine Linux," 2016). This basically means that the image download time decreases dramatically, the security of official Docker images increases, and Docker hosts use less space, which is always a benefit (Christner, 2016).

The Alpine Linux is designed with simplicity, security, and efficacy in mind. It is based on musl, libc and busybox, which make it smaller than, e.g., GNU/Linux distributions. Alpine is a fully functional minimal distribution that has left over all unnecessary packages and contains its own package management system, which is called apk. Alpine also has grsecurity/PaX for increased security that prevents entire classes of zero-day and other vulnerabilities. Grsecurity is a security enchantment that defends against a large variety of security threats of system hardening, and PaX is a patch for a kernel that adds the least privilege for memory pages ("Grsecurity," 2016; "PaX Security," 2016). Both these security technologies are over 15 years old; thus, they are quite battle-hardened. (Thompson, Latchman, Angelacos, & Pareek, 2013).

As the official support moves or has already moved on top of Alpine Linux, the solution should also use Alpine-based images whenever possible. The caveat here is that Alpine is based on musl and libc, whereas some software requires glibc and alternative libraries. Thus, some software runs into issues inside Alpine containers if not aware of the library differences because of the dependency. Alpine Linux-based containers also increase the security of the system, as the minimal images have small attack surfaces and the aforementioned security hardening technologies. Alpine is also minimal in size and does not contain anything unnecessary; thus, it is possible to create extremely stable and maintainable Docker images.

4 DOCKER-BASED CONTINUOUS DELIVERY SYSTEM

In this chapter, the architecture that has been mentioned in Chapter 3 is going to be implemented and executed. The implementation is demonstrated with a real Dockerfile syntax code and Docker commands in Appendix A. The implementation also follows the best practices and tries to mitigate or avoid possible concerns that are described in Chapters 2 and 3. The newest version of Docker (Docker 1.12.1) has been used at the time of implementation. The section headings' order describes the recommended implementation workflow, i.e. how the architecture should be systematically built.

4.1 Creating a Docker host

There are multiple ways to get Docker up and running. For example, to get Docker running on a newer Windows version, there are two alternative ways: Windows 10 can use "Docker for Windows" that is capable of running Hyper-V containers, and the Windows 2016 server can run both native Windows containers and Hyper-V containers. For older systems, there are Docker Toolbox, which runs Docker inside Oracle Virtualbox ("Install Docker on Windows," 2016). However, it is recommended to have the core components of the proposed architecture on top of Linux, as Linux-based Docker is much more stable than the relatively new Windows-based Docker. Thus, this section covers the installation of Docker to Linux-based systems. Since there are multiples of possible Linux distributions available for Docker, this study only demonstrates the installation on Ubuntu, as it is the only fully supported distribution by the company. In Ubuntu, the minimum kernel version is 3.11 for Docker to work. Thus, the kernel must be upgraded if it is not 3.11 or newer.

The easiest method to install Docker on Ubuntu is to use a pre-made script that checks all dependencies of the system and configures the system based on

that. The command itself is a bash script that is located on Docker premises and is executed with curl:

```
curl https://get.docker.com | sh
```

However, the previous script would only install Docker and not do anything else. Thus, to install and configure Docker hosts in the future that support the created architecture, a script will be used that will also install the other needed configurations (see Appendix A).

The aforementioned script needs only the IP address of the host and a downloaded certificate for the Docker Private Registry. The script basically installs Docker, configures needed environmental settings for the solution and adds the new host to be part of the existing Docker host cluster. The demonstrated configurations are mandatory as without them, the Docker host cannot be part of the designed cluster.

4.2 Configuring version control and Jenkins Master

As this architecture uses an existing Jenkins Master due to requirement, there is no need for Jenkins installation. However, for reproducibility of this study, a Jenkins Master container installation is described in Appendix A. In addition, as there is a requirement to use existing Git and Gerrit services available for version control and reviewing by the firm, the implementation does not cover these.

Installing plugins in Jenkins is done via a plugin manager. Different plugins can be searched using a search bar, and when a wanted plugin is found, it can be selected and installed with the press of a button. In this case, the Jenkins Docker Plugins are searched and installed, and when the Jenkins service has restarted, it is available to use. After the installation of a Docker plugin, the needed configurations can be done (see appendix A).

Automated tasks in Jenkins are called jobs. Jenkins jobs are basically only a clear presentation of execution order of, e.g., software builds or software tests, in which each step inside the build is executed after the previous one. If the job can do all the given tasks, such as the written shell scripts, it announces that the job has been successful. If it is a partly successful or failed build, it also announces that.

After the Jenkins Master, the Docker plugin and Jenkins jobs are configured correctly; it is possible to connect to the Docker hosts to start wanted builds just the same way as in the old system. Thus, the solution provides usability as users do not need to know anything new inside Jenkins to achieve the desired performance. However, the build environments, i.e. build slaves, must be designed before execution to achieve the designed scalable build cluster.

4.3 Creating a Jenkins slave

The Docker hub contains multiple versions of Jenkins slaves as Docker images. However, as there has not been Alpine Linux-based images that also have an SSH connection available, an Alpine Linux-based image with SSH was made to create a as minimal and secure as possible Jenkins slave to accomplish the requirements. Although the CIS benchmark states that SSH should not be used, it is mandatory to do so as a requirement because there are currently no other available methods to make a Jenkins-based build cluster work. The SSH connection has been made as secure as possible, as the connection can be done only with a unique RSA keypair. Thus, the connection is secure and ensures that the private key for connecting containers is only available for Jenkins.

The created Docker image is based on the official Alpine Linux image, which is the future of containers and is downloaded from a trusted source (Christner, 2016). All the aforementioned best practices from Benevides (2016) and Dockerfile Best Practices (2016) are used to create a robust image that can be a base image for all other images. This means that if a Dockerfile is built with

```
docker build -t alpinebase:1.0 .,
```

all subsequently created images could start with *FROM alpinebase:1.0.* The created image contains basic build tools such as Git, curl, bash and other mandatory dependencies. This image will also help the subsequently created images to start building from a robust and securely created image that already covers all the basic dependencies. Thus, this image gives better usability and maintainability that enable developers focus only on editing their own Dockerfile, which is based on the created base image.

The most important value of this base image is its size, which is only 160 megabytes compared to the alternative Jenkins slave running on Centos that is over 700 megabytes. Therefore, using the Alpine-based Jenkins slave image decreases the overall performance, e.g., the download times will be much shorter.

4.4 Creating a Docker Private Registry

The created images and containers must be kept on premises as a requirement. Therefore, the Docker private registry is needed. To make the registry secure, RSA keypairs are used for connecting the Private registry. Since the command for starting the Docker Private Registry is rather long, Docker Compose is used, which is a tool for running multi-container applications (“Docker Compose,” 2016).

Docker Private registry is the backend of the solution. To make the solution more usable, the available images can also be seen in the Docker Registry Frontend, where created and pushed images can be viewed in the browser. The registry frontend also allows viewing the versions of the images by the used

tags. Thus, it is possible to see the full history in the browser of a certain image and achieve the requirement of greater usability from easier readability.

The created registry can be connected from Linux- and Windows-based workstations by adding the registry RSA certificate to the right place (see Appendix A). Thereafter, the Docker Private registry can be used. To push images to the registry, the image that is going to be pushed must be tagged with the registry's domain name. The used naming syntax informs Docker about the location of registry. If the previously created folder has the same syntax and the certificate is valid, the image is pushed to the registry. An IP address can also be used, but domains are recommended to simplify the usage of the registry (Mouat, 2015b). If the push has been successful, the image can be seen in the frontend of the registry and be downloaded to all eligible hosts with the *PULL* command using the same syntax.

4.5 Starting Docker cluster and the containers

The preceding sections have made it possible to run Docker builds on a single host. However, for a more robust, scalable, and production-ready system, the following containers are required in the Docker-based build cluster.

As mentioned before, the Logspout container needs to be running in every host when the logs are being sent to a remote location to ensure safety (see Appendix A). Therefore, for larger clusters, the Logspout container or alternatives are mandatory as the logs are required for error handling. However, the Logspout container only sends logs, so a monitoring container is also needed for real-time information. Matthias and Kane (2015) stated that the native statistic feature of Docker, called Docker Stats, is a helpful tool but does not deliver any graphical stats; thus, Google's cAdvisor or similar tools are recommended. The cAdvisor can be run natively on the host, but the simplest method is to run it inside a container. After a successful execution, the container delivers a user interface to the host's IP address and given port and provides a graphical view, e.g., CPU, RAM, running processes, and running images and graphs about network load.

The aforementioned remote location is the Elastic Stack. It can be installed either natively on the machine, each component on its own container, or be launched with the previously mentioned Docker Compose. After the correct configurations, the Elastic Stack is up and running. Thus, all required logs from the Docker hosts, which are delivered by Logspout containers, can be seen in the Kibana interface. The Kibana provides a search bar with multiple filters, so the wanted logs can be found in a required quantity.

The cluster needs service discovery for robustness because the standalone Docker Swarm does not have it, and the engine-based Swarm cannot yet be used due to limitations. Therefore, Consul containers are started for maintaining Docker hosts IP addresses for the Docker Swarm. A Consul container must be started before the Swarm to avoid problems, as the command to start Swarm

includes the Consul host's IP-address. It has been recommended to have at least three master Swarm hosts and three Consul masters for a robust build environment. However, it has also been stated that there should not be more than seven master Swarm hosts and seven Consul masters to ensure that the cluster is not too complex and unpredictable when errors occur (Mouat, 2015b). By default, the Swarm masters are not part of the build pool, but they can be included by adding a worker node to each of them (see Appendix A).

Docker volumes are used for the workspace of build containers. Volumes themselves are files or directories that are not part of a certain image. Instead, these directories are mounted to running containers from the host. According to Mouat (2015), there are three types of volumes. Runtime volume makes the /data folder inside the container into a volume. This means that all files inside the directory are copied to the host. The second method is to use the *VOLUME* instruction inside a Dockerfile, which works the same as runtime volume. The third method is to run a definite directory on the host to be bound in the container, which mounts the directory inside the container. However, another common practice is to use data containers in Docker to share data with other containers (see Appendix A). The main benefit of this is that it enables data containers to be easily moved together with other containers, and it provides a useful namespace for volumes. As UFS allows that new running containers do not consume more space, it makes sense to use the existing images. Thus, the same Jenkins slave image can be used for the base of a data container, where runtime logs and other data are saved.

The build cluster creates many images. Thus, a Docker host's disk space can get full somewhat quickly. In the following example, Docker images' creation has failed a couple of times, and thus a *docker images -a* can provide unwanted or not needed layers of these images to the host.

```
<none><none>          6ed163f9ee9f          6 days ago    14.68 GB
<none><none>          8726553d3838          6 days ago    14.68 GB
<none><none>          5e7778ee20bb          6 days ago    12.3 GB
alpine   latest       ee4603260daa          9 days ago    4.803 MB
```

These images could be deleted manually; fortunately, there is no need for that because of Spotify's creation. The Docker-GC container automatically deletes all containers that do not belong to any existing containers and have exited more than an hour ago. Thus, it is safe to use in this type of a build cluster environment.

5 DEMONSTRATION AND EVALUATION OF THE SOLUTION

As this study describes the design and implementation in depth, it in itself demonstrates the created artifact as a proof of concept. After the solution had been successfully installed, it worked as intended, and this is recapped in Figure 11.

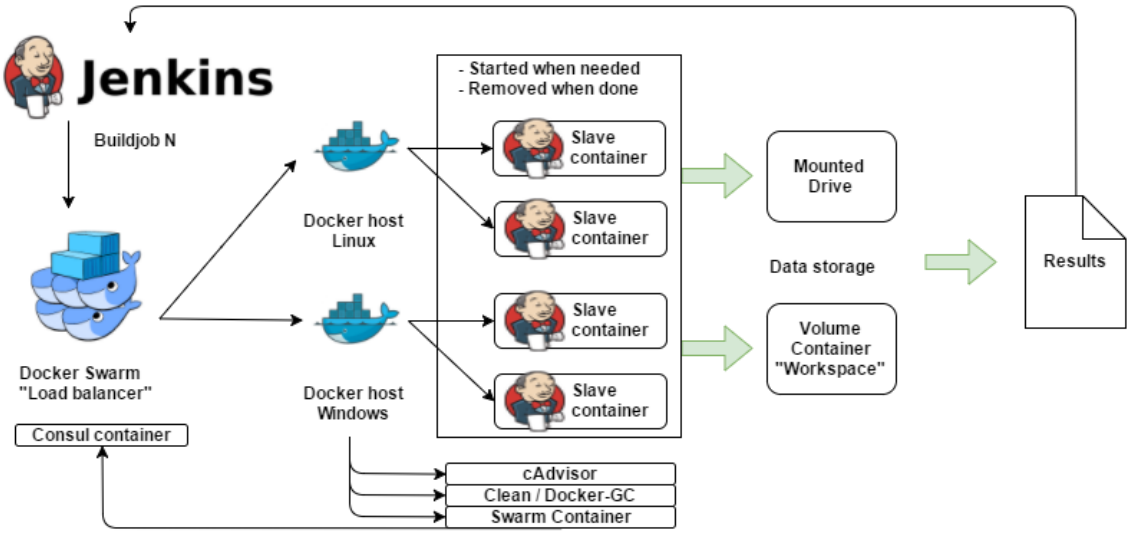

Figure 11 - Working Solution

Whenever Jenkins notices a change in the code base, it starts a new build through Docker Swarm on the correct dependencies containing an image. The containers start from nothing, build the code, and deliver the results to a dedicated workspace. After the build, there are available results, and the container can be pushed to the Docker private registry. Thus, the solution accomplishes all the requirements, as it uses Jenkins and Docker on premises only. The aforementioned benefits of Docker are achieved as the solution uses less resources more efficiently with security and usability in mind. The solution is

lightweight, and it works; thus, it saves computing resources and delivers a product-viable solution.

5.1 Testing the solution

The solution was tested with multiple different test cases. The most important tests were to integrate Docker to the existing infrastructure, which was the major requirement, and key pieces to all problems. When the Docker host had been installed and configured and the Docker plugin was inside Jenkins master, it was possible to connect the Docker host from Jenkins. Thus, the solution passed the first requirement-based test.

Other requirement-based tests were to check if all the used Dockerfiles and Docker images were on premises. As the Docker Private Registry is hosted inside the company's network and the Dockerfiles are in version control, the second major test was also passed. In addition, it can be seen from the Dockerfiles that the best practices are in use, e.g., the Dockerfiles are written in a robust style, and maintenance and scalability are considered. Moreover, the security and light weight requirements of used images were revealed in the tests, as the images are based on the Alpine Linux, which is the most secure base container (Christner, 2016).

The first additional test was to check if one Docker host contained the required dependency image, but another Docker host had a smaller amount of workload. The solution did not start the job on the host that already had the image but on the second host, which was intended by the design. Thus, the requirement of flexibility and scalability of build containers was demonstrated as the builds could be built on multiple hosts.

The second additional test was if the image has a registry syntax in front of it as a label, will the solution figure out by the label to download the image instead from the registry. The solution downloaded the image from the registry; thus, it also passed the requirement of usability, as the label automatically alerts informs Docker host about an image needs to be downloaded in place of any manual file transfers.

The third successful additional test was to check the RSA-keypair injection inside the image, as can Jenkins pull source code from Git to inside the container, without the container having the private key. This was also achieved; therefore, there is no need to copy the keys inside the containers. This meets the security requirement.

The solution was also demonstrated to the company through a couple of two-hour presentations in terms of live demonstration and the overall discussion of the implementation. As the audience was only IT professionals and their feedback was positive, it is justified to say that the solution works as intended.

An example of the live demo was to run company's software product that is using Docker, to run inside of a Docker container. As this kind of demonstration is purely experimental and only demonstrates the possibilities of Docker, it

is not discussed in detail. However, this particular experiment fulfilled the "out of the box" requirement. The Docker-in-Docker Alpine Linux Dockerfile can be seen in Appendix A.

5.2 Evaluation of implementation

This study started due to problems in the CD system, as the system had troubles in reliability, stability, and automation, and the overall performance was slow; therefore, a faster-paced and more trustworthy solution was needed. Other requirements were to flawlessly integrate the Docker-based solution to existing infrastructure and to get the possible benefits of Docker in use.

The proposed solution is up and running, is integrated to be part of the production CD pipeline, meets its requirements and uses the benefits of Docker securely. Thus, the implementation of the artifact was successful and delivers value in a somewhat new technology, which does not have plenty of similar studies available. Thus, the implementation and the background research contribute to this field of study. However, this study lacks scientific material for the same reasons, which could make a solid foundation of the study. Another constraint of this study is that the fact that many of the used technologies are still evolving may affect the solution either positively or negatively in the future. However, the selected technologies were justified to be the best candidates, and there were no other suitable candidates when this research was executed.

The design and implementation parts have been described in detail that the findings of this study are somewhat reproducible. All the used technologies were open source, which also will allow for easy replication in the future without worrying about licensing. However, the researcher also experimented licensed software, e.g., how Oracle products work inside the containers. Since there are many ambiguities in these cases, such as how containerized products differ from basic installs, it is unfortunate that these experiments cannot be described.

One of the implementation sections was previously designed to be about Windows containers. As Windows containers are an extremely new concept, this type of research would have further extended the quality of the implementation. However, Windows Server 2016 was made publicly available later than expected, and the Windows container part itself did not make its way to the implementation. Another reason was that unlike the Linux counterpart of Docker, the Windows Docker is still relatively unstable because it still lacks many features. Nevertheless, Windows containers would have been a great addition to the implementation, and they will be added to the system in the future.

The implementation was also compared to the existing CD pipeline, which can be seen in Table 4.

Table 4 – Comparison of CD solutions

Parameter and requirement	**Old system**	**New system**
Disk usage and build environment [Lightweight solution]	Approx. 50 GB per Hyper-V server. Maximum of 4 slave executors per server.	160 MB base container; UFS allows disk space sharing. Maximum number of containers 100–1000 per host.
Dependency management [Usability]	Needed dependencies installed manually per server; code must be built on a dedicated host.	Dependencies inside container; changes to dependencies will change automatically from version control. Container can be built on every host that has same kernel
Use of computing resources [Performance]	Server remains idle when build or testing is not running; resources cannot be shared.	Containers start whenever needed and are removed after a dedicated task, as resources are only used when needed. Containers can also share the resources dynamically inside the Docker hosts.
Build environment scaling [Scalability]	Manual installation of needed dependencies, takes 1–5 days based on the build environment	Automated, using a script that will do all needed installations for needed dependencies. Takes 1–2 minutes based on the download speed.
Host maintenance [Usability]	Servers need manual restarts for services and build slaves when down	Restarts are automated; if a container malfunctions, a new container is started as a replacement
Recovery of the solution [Robustness]	Needs manual restarts if a build host is unavailable. Build job cannot be launched elsewhere	Host may need a manual restart. Build jobs can be launched elsewhere on the build cluster.
Build performance [Performance]	Overhead from extra OS layer; builds start almost immediately	No overhead; builds based on used kernel. Builds start after 10 seconds, as containers are started from nothing.
Monitoring of build environment [Usability, Security]	Build logs are per server; no dedicated log location	Build logs are per container; build logs are sent to a centralized remote location for safety
Cloning environment [Usability]	Full clone of VM takes 5–15 minutes.	Full clone of Docker container takes 1-10 seconds
Security of solution [Security]	Connections possible with administrator rights. Slaves connected with username and password. Isolation provided by the hypervisor.	Connections possible for only dedicated set of people. Slaves connected with RSA keypairs. Isolation provided per container. However, root-level access is possible to the host when poorly used.

Although VMs and containers are both virtualization techniques, they are used for different purposes (Pahl, 2015). Thus, VM-based environments can be used when, e.g., making a monolith makes sense. However, in every other comparison and especially when code is built, the Docker-based solution is stronger, as it can be seen in the Table 4 (Adufu et al., 2015). The first comparable variable is the size of the needed build environments. The old system was using Hyper-V-based virtualization servers; thus, the size of each server is larger when com-

pared to Docker containers. With containers, applications share the same operating system's kernel and thus these deployments are minimal in size compared to the traditional virtualization alternatives, thus making it possible to store hundreds to thousands of containers on a single physical host (Bernstein, 2014). A typical server contained approximately 50 gigabytes of disk space, whereas one Jenkins slave uses only 160 megabytes when started. Therefore, using the new system accomplishes the requirement of a lightweight solution, which uses less disk space and is thus cheaper.

Another factor that should be compared is dependency management. The Hyper-V build servers are quite limited because they have pre-installed dependencies for each build. The implementation allows builds to run on every host, as the dependencies are inside the containers. Thus, the computing resources can be allocated more precisely and equally, which fulfills the requirement of performance utilization. The dependencies can also be maintained more easily as all the used dependencies are described in the Dockerfiles. As containers contain all the needed dependencies, they enable a consistent way of automating the faster deployment of software inside highly portable containers (Bernstein, 2014).

The Docker-based CD system can also be scaled up by adding new hosts to the build cluster, which is not possible in the old system. The new solution sends the monitoring and build logs to a remote location, which makes the system safer. Thus, the new system meets the requirements of scalability and security. The new system also recovers quicker, as build locations can be changed and restarts are mostly automated. Thus, the requirement of a robust solution is also fulfilled. Finally, when build times were compared, the Docker-based solution had the same build times to those of the Hyper-V servers. However, containers start from nothing with fresh environments, as it takes few seconds to start a container. Thus, the new system has either equal build speed or a bit worse depending on the perspective.

The implementation also followed the described best practices in terms of syntax and security. Furthermore, this study provides these practices in clear lists; therefore, the used practices can be evaluated and also used elsewhere. To make sure that the implementation runs safely, the previously described security benchmark test was run to achieve the desired level of security for each host. However, there are still improvements to be made in the future, as all tests did not pass, e.g., to make the build cluster work, the build slaves use SSH connections. Nevertheless, at least the lacking security properties are now known and can be fixed when needed.

As Bui (2014) and Chelladhurai et al. (2016) stated that other remaining security issues include ARP spoofing and MAC flooding attacks, which can be done against a default networking model of Docker because of its vulnerability. However, these attacks can be avoided with awareness and by creating own networking stacks, which is quite easy to implement. Other mentionable exploits are by Mouat (2015a), namely kernel exploit, Denial-of-service attacks, escapes from container, poisoned images, and secret compromises. As the new

solution uses only on-premises images and does not add any secrets to containers, the last two exploits can be totally avoided. Mouat (2015) stated that the security of Docker is mostly about limiting the attack surface through monitoring. The new solution mitigates the last three exploits through the incorporated monitoring solution.

Azab and Domanska (2016) stated that Docker was still maturing and had operational and some security issues. However, Docker still has advantages over VMs in terms of flexibility and security, e.g., each container has a separate network stack. If the host does not allow the interaction between containers, there cannot be any connections. Furthermore, the usage of Dockerfiles implements a new kind of security, as it is not trivial to install a whole VM with a script, whereas script usage is a standard use case in Docker. As *docker commit* based building should not be used due to version control, the security vulnerabilities can be easily discovered from the Dockerfiles. Thus, as the solution's Dockerfiles obey the mentioned security best practices and the files are in version control, the new solution fulfills the additional security requirements.

The container-based solution is mostly an enabler to achieve wanted outcomes that are mandatory for CD. However, the container technology still lacks few features that are limitations. Dua et al. (2014) described that containers still need a real standard for container file format, i.e. a secure way of isolating memory and networking and independence of an operating system to achieve the finalized form of robust technology. Therefore, container technology still needs improvements in the future. Wolf and Yoon (2016) mentioned that modern CD is a combination of many available tools and technologies that ensure high-quality products for production. They also stated that Gerrit, Jenkins and Docker work flawlessly together. Thus, Wolf and Yoon (2016) state that they use modern technologies, e.g., Gerrit, Jenkins and Docker are used to perform automated testing for every release for rapid iterations and to ease complex cluster configurations. Hence, since the new solution uses the same technologies, it is also modern.

One bottleneck of the design is the old Jenkins master, as when it goes down, all the build jobs and other dedicated tasks cannot be used. The requirement to keep the current master for production and maintenance reasons, so this master could not be changed. However, in the future, the Jenkins master automation servers should be based on per software project in the company and run inside of containers so that only one team suffers when the master build server fails and the Jenkins master containers are quickly launched back to online.

The feedback of the solution was positive in the company. As all the users of this solution are IT professionals whose positive feedback is valuable for the implementation. As Fowler and Highsmith (2001) stated, software firm's highest priority is to satisfy the customer needs with CD of software. Thus, the working solution fixed a high-priority problem while obeying the requirements and constraints of the existing architecture. The build environments can now be more robust, and the new solution performs better than the old system. The

solution also delivers usability with clear user interfaces, e.g., the registry frontend and monitoring tools deliver graphs and other mandatory information of the implementation. Thus, the implementation delivers a solution for human purposes, which is mandatory in DSRM (Peffers et al., 2007).

6 DISCUSSION

This study has examined how the Docker container technology can solve problems of current CD pipeline with designed architecture. The study was executed as DSRM through a qualitative type of research and produced a working artifact to solve a problem in a software firm. The literature was gathered from the most cited, most valuable, and reliable academic sources, and it was supplemented with professional texts about technologies. The most important contribution of the study is the working implementation itself, the comparison table of the old and new systems and the undercoating research material that made the implementation possible. The most important observation from the implementation is that a Docker-based CD pipeline can improve an already working CD system. Thus, new systems should be based on container technologies if possible.

This study has had some challenges in terms of finding enough relevant scientific material, as Docker is only a three-year-old technology and thorough examinations or academic papers of this technology do not exist yet in a large quantity. Thus, the academic papers were supplemented with professional technology texts to obtain a sufficient quantity for this study.

The literature has revealed that container technology has better performance, scalability, and usability than hypervisor-based virtualization (Dua et al., 2014; Felter et al., 2015; Joy, 2015; Kozhirbayev & Sinnott, 2017). The performance of containers affects server workloads because in many cases, VMs' performance tends to be inferior, whereas Docker containers perform equally to native performance of bare metal servers (Slominski et al., 2015).

Docker containers can also deliver better security when used correctly (Bui, 2014; Chelladhurai et al., 2016). Docker security basically relies on three factors: network operations security, user space managed by the Docker daemon and the vigil of this isolation by the kernel (Combe et al., 2016). Thus, securing the Docker container is mostly about monitoring and limiting the possible attack surface by doing correct configurations (Mouat, 2015a).

There are still use cases left for hypervisor virtualization, such as when a monolith type of software makes more sense (Bernstein, 2014; Chung et al., 2016; Joy,

2015). However, Docker can also be seen as a bridge between operations and developers. Therefore, Docker enables developers to use whatever new technologies they want, as the final format is always the same as that of the container, which results in more trustworthy deployments (Joy, 2015).

Docker containers deliver the same advantages, such as scalability, delivery, speed, density, and portability, as those that are appreciated in CD. Another great impact of Docker is the capability to remove "dependency hell," which is one of the largest problems of the software industry (Boettiger, 2015). As software firms' highest priority is to satisfy the customer needs with continuously delivered software, Docker-based CD can seriously enhance the competitiveness of a company (Fowler & Highsmith, 2001).

The created artifact was integrated to be a part of an existing CD system. Thus, the requirements, limitations and constraints were kept in mind when the artifact was implemented. As there were both Linux- and Windows-based software, both methods of building software natively on top of the host kernel were examined. As in DSRM type of research, the requirement was to deliver a real deliverable. This study has provided a proof of concept in the chapters 3 and 4. As this study use open-source software and best practices of making Docker-based software securely in deep detail, it is highly reproducible for future studies. The implementation mitigated and avoided the concerned areas of Docker by following the aforementioned practices and by providing its own solutions, e.g., an Alpine-based Jenkins slave, to the field of study. The new solution uses images on premises, but this could be changed in the future. The current Docker private registry delivers all the mandatory parts of a desired repository, but, e.g., image deletion from the registry is rather hard and the limitation of who can use the registry can be tricky; these fields should be examined in the future.

The implementation has been tested with multiple different test cases, and it has been demonstrated to the software company's IT professionals. Thus, the implementation is of high quality, as it is reviewed and tested in multiple iterations. The created artifact delivers value, as it is in the field in which similar studies do not exist in a sufficient quantity. Thus, the implementation and the background research has contributed to this field of study. In comparison to the existing system, the implementation outperformed the existing system at various levels and allowed new use cases for the employees. Therefore, the implementation has also contributed at a practical level. The solution is currently in use; thus, the implementation itself has delivered an artifact for human purposes, which is mandatory in DSRM process (Peffers et al., 2007).

Overall, Docker is an interesting technology that can be used in multiple different settings, particularly in the CD. However, the Docker-based solution is not an answer to everything, as it is not a virtualization platform, cloud platform or configuration management but a method to achieve desired outcomes (Matthias & Kane, 2015). Hence, before Docker is used, the use case of Docker must be considered, such as how Docker is going to be used, for what purpose and to solve what problem. After validation of these fields, the use of Docker

can be justified. Therefore, when Docker is used, it is guaranteed that given features can deliver great performance and usability in a secure way, when used correctly.

As this study has only focused in detail on the Docker container technology, and there were limitations and constraints in the study, deeper examinations of the Docker-based CD solution should be conducted. These in-depth analyses should be started from nothing without any limitations, so that the best of the best technologies and their practices could be used. One research path could also be to make a quantitative research based on the findings of the evaluations chapters' comparison table. Another research path could investigate how Docker logging and monitoring could be simplified. Currently logging and monitoring are complex systems that generate 1,700 rows of raw logs per hour in one single host. Thus, an examination of which logs are valuable would enhance the overall usability.

The CPU and memory testing were part of performance of the Docker chapter. However, these tests are mainly done with HTC, MTC or with simple test containers against different virtualization technologies. Thus, they do not frequently measure use cases of software companies, who are interested in different types of measurement. Consequently, a quantitative study of CPU and memory usage from the software company's perspective with a closer-to-actual use case of containers would make for an interesting study.

Unfortunately, this research did not have resources to examine all the available container technologies in practice; therefore, the examined solution could contain practical usage of different container technologies. Nevertheless, Docker itself should also receive more in-depth examinations, as the usage of virtualization technologies will only rise in the future. Thus, Docker and Docker-based research will most certainly be part of the software research and development (Joy, 2015; Mouat, 2015b).

REFERENCES

Adufu, T., Jieun, C., & Yoonhee, K. (2015). Is container-based technology a winner for high performance scientific applications? In *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific* (pp. 507–510). https://doi.org/10.1109/APNOMS.2015.7275379

Axelrod, C. W. (2014). Reducing software assurance risks for security-critical and safety-critical systems. In *Systems, Applications and Technology Conference (LISAT), 2014 IEEE Long Island* (pp. 1–6). https://doi.org/10.1109/LISAT.2014.6845212

Azab, A. & Domanska, D. "Software Provisioning Inside a Secure Environment as Docker Containers Using Stroll File-System," *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, 2016, pp. 674-683.

Bass, L., Holz, R., Rimba, P., Tran, A. B., & Zhu, L. (2015). Securing a Deployment Pipeline. In *Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop on* (pp. 4–7). https://doi.org/10.1109/RELENG.2015.11

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *Cloud Computing, IEEE*, *1*(3), 81–84. https://doi.org/10.1109/MCC.2014.51

Boettiger, C. (2015). An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, *49*(1), 71–79.

Bui, T. (2014). Analysis of Docker Security, *Aalto University School of Science*(Aalto University T-110.5291 Seminar on Network Security).

Chelladhurai, J., Chelliah, P. R., & Kumar, S. A. (2016). Securing Docker Containers from Denial of Service (DoS) Attacks (pp. 856–859). Presented at the Services Computing (SCC), 2016 IEEE International Conference on, IEEE.

Chung, M. T., Quang-Hung, N., Nguyen, M.-T., & Thoai, N. (2016). Using Docker in high performance computing applications (pp. 52–57). Presented at the Communications and Electronics (ICCE), 2016 IEEE Sixth International Conference on, IEEE.

Combe, T., Martin, A., & Di Pietro, R. (2016). To Docker or Not to Docker: A Security Perspective. IEEE Cloud Computing, 3(5), 54-62.

Dhakate, S., & Godbole, A. (2015). Distributed cloud monitoring using Docker as next generation container virtualization technology. In 2015 Annual IEEE India Conference (INDICON) (pp. 1-5). IEEE.

Dua, R., Raja, A. R., & Kakadia, D. (2014). Virtualization vs Containerization to Support PaaS. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on* (pp. 610–614). https://doi.org/10.1109/IC2E.2014.41

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. In

Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on (pp. 171–172). https://doi.org/10.1109/ISPASS.2015.7095802

Fink, J. (2014). Docker: a Software as a Service, Operating System-Level Virtualization Framework. *Code4Lib Journal*, *25*(Journal Article).

Fowler, M., & Foemmel, M. (2006). Continuous integration. *Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf*, (Journal Article), 122.

Fowler, M., & Highsmith, J. (2001). The agile manifesto. *Software Development*, *9*(8), 28–35.

Gerlach, W., Tang, W., Wilke, A., Olson, D., & Meyer, F. (2015). Container orchestration for scientific workflows. In Cloud Engineering (IC2E), 2015 IEEE International Conference on (pp. 377-378). IEEE.

Haydel, N., Gesing, S., Taylor, I., Madey, G., Dakkak, A., de Gonzalo, S. G., & Hwu, W. M. W. (2015). Enhancing the Usability and Utilization of Accelerated Architectures via Docker. In 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC) (pp. 361-367). IEEE.

Hevner, March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75–105.

Higgins, J., Holmes, V., & Venters, C. (2015). Orchestrating Docker Containers in the HPC Environment, (Journal Article).

Humble, J., & Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

Jaramillo, D., Nguyen, D. V., & Smart, R. (2016). Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016* (pp. 1–5). https://doi.org/10.1109/SECON.2016.7506647

Joy, A. M. (2015). Performance comparison between Linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in* (pp. 342–346). https://doi.org/10.1109/ICACEA.2015.7164727

Kozhirbayev, Z., & Sinnott, R. O. (2017). A performance comparison of container-based technologies for the Cloud. *Future Generation Computer Systems*, *68*(Journal Article), 175–182.

Luo, S., Lin, Z., Chen, X., Yang, Z., & Chen, J. (2011). Virtualization security for cloud computing service. In *Cloud and Service Computing (CSC), 2011 International Conference on* (pp. 174–179). https://doi.org/10.1109/CSC.2011.6138516

Manu, A. R., Patel, J. K., Akhtar, S., Agrawal, V. K., & Murthy, K. (2016). Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. In *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)* (pp. 1–14). https://doi.org/10.1109/ICCPCT.2016.7530217

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

Naik, N. (2016). "Building a virtual system of systems using docker swarm in multiple clouds," *2016 IEEE International Symposium on Systems Engineering (ISSE)*, Edinburgh, United Kingdom, 2016, pp. 1-3.

Pahl, C. (2015). Containerization and the PaaS Cloud. *Cloud Computing, IEEE*, *2*(3), 24–31. https://doi.org/10.1109/MCC.2015.51

Pahl, C., & Lee, B. (2015). Containers and Clusters for Edge Cloud Architectures -- A Technology Review. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on* (pp. 379–386). https://doi.org/10.1109/FiCloud.2015.35

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77.

Preeth, E., Mulerickal, F. J. P., Paul, B., & Sastri, Y. (2015). Evaluation of Docker containers based on hardware utilization (pp. 697–700). Presented at the 2015 International Conference on Control Communication & Computing India (ICCC), IEEE.

Schermann, G., Cito, J., Leitner, P., & Gall, H. C. (2016). Towards quality gates in continuous delivery and deployment (pp. 1–4). Presented at the Program Comprehension (ICPC), 2016 IEEE 24th International Conference on, IEEE.

Simon, H. A. (1969). The sciences of the artificial. *Cambridge, MA*, (Journal Article).

Slominski, A., Muthusamy, V., & Khalaf, R. (2015). Building a Multi-tenant Cloud Service from Legacy Code with Docker Containers. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on* (pp. 394–396). https://doi.org/10.1109/IC2E.2015.66

Thompson, C. A., Latchman, H. A., Angelacos, N., & Pareek, B. K. (2013). A Distributed IP-Based Telecommunication System Using SIP. *arXiv Preprint arXiv:1312.2625*, (Journal Article).

Walls, J. G., Widmeyer, G. R., & El Sawy, O. A. (2004). Assessing information system design theory in perspective: how useful was our 1992 initial rendition? *JITTA: Journal of Information Technology Theory and Application*, *6*(2), 43.

Wang, J. C., Cheng, W. F., Chen, H. C., & Chien, H. L. (2015). Benefit of construct information security environment based on lightweight virtualization technology. In Security Technology (ICCST), 2015 International Carnahan Conference on (pp. 1-4). IEEE.

Wolf, J., & Yoon, S. (2016). Automated Testing for Continuous Delivery Pipelines.

Xavier, M. G., De Oliveira, I. C., Rossi, F. D., Dos Passos, R. D., Matteussi, K. J., & De Rose, C. A. F. (2015). A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on* (pp. 253–260). https://doi.org/10.1109/PDP.2015.67

Xiangyang Luo, Lin Yang, Linru Ma, Shanming Chu, & Hao Dai. (2011). Virtualization Security Risks and Solutions of Cloud Computing via

Divide-Conquer Strategy. In *Multimedia Information Networking and Security (MINES), 2011 Third International Conference on* (pp. 637–641). https://doi.org/10.1109/MINES.2011.54

Zeng Shu-Qing, & Xu Jie-Bin. (2010). The Improvement of PaaS Platform. In *Networking and Distributed Computing (ICNDC), 2010 First International Conference on* (pp. 156–159). https://doi.org/10.1109/ICNDC.2010.40

Zhang Qiang, Wu Yunlong, Cui Dong, & Dang Zhuang. (2010). Research on the security of storage virtualization based on trusted computing. In *Networking and Digital Society (ICNDS), 2010 2nd International Conference on* (Vol. 2, pp. 237–240). https://doi.org/10.1109/ICNDS.2010.5479355

COMMERCIAL REFERENCES

Alpine Linux. (2016). Retrieved November 19, 2016, from https://alpinelinux.org/

Apache Mesos. (2016). Retrieved November 19, 2016, from http://mesos.apache.org/

Apache ZooKeeper. (2016). Retrieved November 19, 2016, from https://zookeeper.apache.org/

Benevides, R. (2016). 10 things to avoid in docker containers. Retrieved November 19, 2016, from http://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/

Christner, B. (2016). Docker official images are moving to alpine linux. Retrieved November 19, 2016, from https://www.brianchristner.io/docker-is-moving-to-alpine-linux/

CIS Docker 1.12 Benchmark. (2016). Retrieved November 19, 2016, from https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.12.0_Benchmark_v1.0.0.pdf

Consul.io. (2016). Retrieved November 19, 2016, from https://www.consul.io/

CoreOS etcd. (2016). Retrieved November 19, 2016, from https://coreos.com/etcd/

CoreOS rkt. (2016). Retrieved November 19, 2016, from https://coreos.com/rkt/

Diogo, M. (2015). Introducing Docker Content Trust. Retrieved November 19, 2016, from http://blog.docker.com/2015/08/content-trust-docker-1-8/

Docker. (2016). Retrieved November 19, 2016, from https://www.docker.com/

Docker Compose. (2016). Retrieved November 20, 2016, from https://docs.docker.com/compose/

Docker Hub. (2016). Retrieved November 19, 2016, from https://hub.docker.com/

Docker logging. (2016). Retrieved November 19, 2016, from https://docs.docker.com/engine/admin/logging/overview/

Docker Plugin. (2016). Retrieved November 19, 2016, from https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin

Docker Registry. (2016). Retrieved November 19, 2016, from https://docs.docker.com/registry/

Docker Security Bench. (2016). Retrieved November 19, 2016, from https://github.com/docker/docker-bench-security

Docker Swarm. (2016). Retrieved November 19, 2016, from https://docs.docker.com/swarm/

Docker Trusted Registry. (2016). Retrieved November 19, 2016, from https://docs.docker.com/datacenter/dtr/2.0/

Docker Windows Containers. (2015). Retrieved November 19, 2016, from https://www.docker.com/microsoft

Dockerfile Best Practices. (2016). Retrieved November 19, 2016, from https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

Elastic Stack. (2016). Retrieved November 19, 2016, from http://www.elastic.co/products

Farcic, V. (2015). Docker Clustering tools compared: Kubernetes vs Docker Swarm. Retrieved November 19, 2016, from https://technologyconversations.com/2015/11/04/docker-clustering-tools-compared-kubernetes-vs-docker-swarm/

Gerrit Code Review. (2016). Retrieved November 19, 2016, from https://www.gerritcodereview.com/

Git. (2016). Retrieved November 19, 2016, from https://git-scm.com/

Glider Labs Logspout. (2016). Retrieved November 19, 2016, from https://github.com/gliderlabs/logspout

Google cAdvisor. (2016). Retrieved November 19, 2016, from https://github.com/google/cadvisor

Google Kubernetes. (2016). Retrieved November 19, 2016, from http://kubernetes.io/

Grsecurity. (2016). Retrieved November 19, 2016, from https://grsecurity.net/

Hall, S. (2016). Kontena Offers a Kubernetes Alternative That Promises to be Easier to Deploy. Retrieved November 19, 2016, from http://thenewstack.io/kontena-offers-kubernetes-alternative-promises-easier-deploy/

Holub, J. (2015). Continuous Integration and Delivery With Docker. Retrieved November 19, 2016, from https://blog.codeship.com/continuous-integration-and-delivery-with-docker/

Install Docker on Windows. (2016). Retrieved November 20, 2016, from https://docs.docker.com/engine/installation/windows/

Jenkins. (2016). Retrieved November 19, 2016, from https://jenkins.io/

Karle, A. (2015). Operating System Containers vs. Application Containers. Retrieved November 19, 2016, from https://blog.risingstack.com/operating-system-containers-vs-application-containers/

Kontena. (2016). Retrieved November 19, 2016, from https://www.kontena.io/

Linthicum, D. (2014). Cloud app containerization all the rage -- but is it anything new? Retrieved November 19, 2016, from http://searchcloudcomputing.techtarget.com/podcast/Cloud-app-containerization-all-the-rage-but-is-it-anything-new

LXC. (2016). Retrieved November 19, 2016, from https://linuxcontainers.org/

Matthias, K., & Kane, S. P. (2015). *Docker: Up & Running*. O'Reilly Media, Inc.

Mouat, A. (2015a). *Docker Security: Using Containers Safely in Production* (Vol. First Edition). Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Mouat, A. (2015b). *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Inc.

Oracle VirtualBox. (2016). Retrieved November 19, 2016, from https://www.virtualbox.org/

PaX Security. (2016). Retrieved November 19, 2016, from https://pax.grsecurity.net/

Petazzoni, J. (2013). Containers and Docker: How secure are they? Retrieved November 19, 2016, from http://blog.docker.com/2013/08/containers-docker-how-secure-are-they/

Polvi, A. (2014). CoreOS is building a container runtime, rkt. Retrieved November 19, 2016, from https://coreos.com/blog/rocket/

Prometheus. (2016). Retrieved November 19, 2016, from https://prometheus.io/

Sheldon, R. (2016). Windows Containers and Docker. Retrieved November 19, 2016, from https://www.simple-talk.com/cloud/platform-as-a-service/windows-containers-and-docker/

Spotify Docker-gc. (2016). Retrieved November 19, 2016, from https://github.com/spotify/docker-gc

Vaughan-Nichols, S. (2015). Docker 1.8 adds serious container security. Retrieved November 19, 2016, from http://www.zdnet.com/article/docker-1-8-adds-serious-container-security/

VMware Workstation. (2016). Retrieved November 19, 2016, from http://www.vmware.com/

What is Docker? (2015). Retrieved November 19, 2016, from https://www.docker.com/what-docker

Young, K. (2016). The Docker monitoring problem. Retrieved November 19, 2016, from https://www.datadoghq.com/blog/the-docker-monitoring-problem/