Alina Aleshkina

MIGRATING CONFIGURATION MANAGEMENT SYSTEMS TO CONTAINERS

Bachelor's thesis
Information technology

2017

South-Eastern Finland University of Applied Sciences

Author	**Degree**	**Time**
Alina Aleshkina	Bachelor of Engineering	December 2017

Title

Migrating configuration management systems to containers

61 pages

Commissioned by

Oy L M Ericsson Ab

Supervisors

Matti Juutilainen (supervising lecturer)
Antti Tolonen (company's representative)

Abstract

Configuration Management (CM) systems are widely used in modern IT infrastructures. These systems are utilized for configuring servers according to predefined instructions in an automated manner. One of the most popular CM software is Ansible.

At the timeframe of this bachelor's thesis, Ansible was used in one of the Ericsson's products as a main CM tool. Due to the nature of the product, this CM software required frequent changes. These changes related to both the predefined instructions and the version of the CM tool. However, introducing these changes required the upgrade of the whole product's infrastructure. Therefore, updating Ansible took excessive time and effort.

The objective of this thesis was to accelerate and abstract the current update process for Ansible. To achieve these goals, it was proposed to migrate Ansible into a container with Docker. In essence, this tool would bound the CM software together with related code, runtime and configurations as a separate package. Thus, the Ansible would be isolated from the underlying OS as well as from the rest of the architecture.

The solution was implemented according to the researched best practices related to the development of Docker containers. As a result, the final container succeeded to significantly increase the speed of the Ansible updates and isolate these updates from the general product's upgrade process. However, the study showed that the containerization would still require additional effort in order to gain a production-ready solution. The most crucial suggested improvements would be related to the container's portability, version control and user permissions.

Keywords

Ansible, configuration management systems, containerization, containers, Docker

CONTENTS

1 INTRODUCTION 4
2 BACKGROUND 5
2.1 From physical servers to containers 6
2.2 Containers 8
2.2.1 Basic terms and definitions 8
2.2.2 Advantages and challenges 11
2.3 Developing containers 12
2.3.1 Developing applications in containers: best practices 12
2.3.2 Migrating existing applications to containers 17
2.4 Configuration management systems 18
2.4.1 Overview 18
2.4.2 Ansible 19
2.5 Running a configurator in a container 20
3 PRODUCT ARCHITECTURE AND EVOLUTION 21
4 PRACTICAL IMPLEMENTATION 23
4.1 Docker images 24
4.1.1 Ansible base image 24
4.1.2 Ansible release image 26
4.2 Building the Docker images 30
4.3 Running the container 32
4.3.1 Networking 34
4.3.2 Secrets management 36
4.3.3 Identifying deployment issues 39
4.3.4 Replacing the existing solution with the container 46
5 RESULTS 49
6 CONCLUSION 53
REFERENCES 57

1 INTRODUCTION

These days, there is a constantly growing demand for highly available applications and rapid deployment cycles, which leads to the adaptation of new tools such as containers. According to Docker Inc. (2017), containers are "lightweight, stand-alone, executable packages of a piece of software that include everything needed to run them: code, runtime, system tools, system libraries, settings". They have become popular among IT enterprises due to several benefits, including quick delivery, low space consumption, significant isolation and high portability, especially compared to virtual machines (Docker Inc. 2017). Although many alternative container frameworks exist, Docker is commonly considered as a de facto standard (Babcock 2015).

Another popular trend among modern IT operations is the concept of Configuration Management (CM) systems. In general, these systems aim to improve infrastructure orchestration by automating repetitive tasks from a centralized point (Red Hat 2017). They also embrace an idea of Infrastructure as Code (IaC). This principle implies that infrastructures can be handled in the similar way as traditional software, for instance, tested, version controlled, continuously integrated and reviewed by other developers (Puppet 2017). For instance, a markup language can be used to explicitly represent how a group of hosts should be configured. There are many CM tools available, for example, Ansible, Chef and Puppet (UpGuard 2017).

One of the existing Ericsson's products uses a CM system, namely, Ansible to configure the rest of the solution. It runs from a dedicated virtual machine, which is delivered to customers with other components of the product as a single VM image. Ansible automatically runs all the provided steps, called playbooks, to set up the environment and requires a minimum of interaction.

However, an update to the playbooks is challenging in the product. If it is needed, the whole product has to be rebuilt and redeployed. This process takes considerable amount of time and is sometimes not even feasible. The alternative could be the upload of new playbooks to hosts independently. However, the problem with this method is that environments could have

different versions of Ansible pre-installed. Therefore, uploaded playbooks could be incompatible with certain versions of the tool.

One of the solutions to these problems is to migrate the existing Ansible configurator to a Docker container. In theory, this approach makes it possible to modify the playbooks without complete upgrades of the product. Instead, the container is rebuilt and redeployed independently. As for possible incompatibilities between Ansible versions and playbooks, containers would solve this issue as well. The reason is that containers tightly couple versions of the tools with a source code.

This bachelor’s thesis focuses on the implementation of an Ansible container in order to show whether this solution is feasible in the product and is able to solve efficiently the problems stated above. The given case is not trivial, since this Ericsson software already has the mature architecture and the organized way to set up and execute Ansible. Therefore, the part of the existing product is needed to be transferred to a container rather than that a container-based product is created from scratch. An additional requirement is that the transition should not have impact on the remaining product functionality.

The structure of the thesis is as follows. Chapter 2 presents the background information on containers and configurators. Additionally, this chapter explains how different configuration management tools suit to be migrated to a container platform. Then, Chapter 3 briefly describes the company case and its evolution via the utilization of containers. Later, Chapter 4 documents the steps taken to implement the solution, including the contents of containers, how they are built and how they are deployed. Lastly, the results and the conclusion are given in Chapters 5 and 6.

2 BACKGROUND

This chapter provides the theoretical background to justify the steps taken in the implementation of the solution. First of all, it describes the timeline of how physical servers have evolved into containers. After that, the information about containers is presented from a technical perspective. Then, the practices of containers’ development are studied. The next subsection describes the

concept of configuration management systems. Lastly, configurators are analyzed to be run in a container environment.

2.1 From physical servers to containers

IT Infrastructures have evolved greatly over time and barely resemble the ones used in the past. For example, a few decades ago a set of physical hosts connected with physical networks and accompanied by physical storage was enough to provide a digital service to a customer. However, this way of delivering applications usually do not suffice in the modern IT world. First of all, software is getting more dynamic and its demand for resources can change at any point of time. This agility is not compatible with the static nature of physical infrastructures. Second, the IT market is becoming extremely competitive, and therefore, the enterprises need to rapidly deliver their products in order to beat their competitors. Again, physical resources usually are not able to support such a fast pace. These and many other limitations led to the point when some another way to handle infrastructures was required. And this was the point when *virtualization* came as a well-suited solution.

Virtualization is a broad term in the IT field. In short, it can be explained as an “abstraction of some physical component into a logical object”, according to Portnoy (2012, 2). In theory, any physical resource can be transformed into a virtual one. In practice, hosts, networks and storage are the most common infrastructure parts to be virtualized. As for this thesis, it focuses only on virtualized servers.

Virtual machines are the essential virtualization units which are able to replace physical servers in infrastructures. Essentially, a virtual machine (VM) is similar to a physical one in the way that it also contains an operating system. However, it is possible to run multiple VMs with different operating systems and applications at the same time on one physical machine. (Portnoy 2012, 35.) Physical components are fairly distributed and scheduled between virtual machines by a *hypervisor*. Thus, resources are more dynamic and a certain share can be provided to a VM quickly on demand unlike in physical hosts.

Once introduced into the market, server virtualization became the dominant technology to deliver applications. However, the increasing need for even more efficient use of resources and even shorter deployment cycles led to the point when VMs were not sufficient anymore. Therefore, the *containerization* (migration to containers) technology, which in fact existed since the early 2000s, started gaining wide popularity (Red Hat enterprise Linux blog 2015). In 2008 The Linux Containers Project (LXC) was presented as a tool around such Linux concepts as *control groups* (cgroups), which allowed grouping the processes, and *namespaces,* which enabled identifying an independent set of users per container (Red Hat enterprise Linux blog 2015). LXC is considered to be a cornerstone in the history of containers. However, the server virtualization still prevailed in infrastructures.

The containerization technology started substantial competition with virtual servers, when the tool called Docker came into the market. dotCloud released it in 2013 as an open source product (Gallagher 2016, 1). First, it was presented as an easy-to-use tool wrapped around LXC (Red Hat enterprise Linux blog 2015). Later on, it matured into its own unique container runtime environment. This tool is targeted for single-process containers, does not support persistent storage and creates stronger isolation from OS in comparison to LXC multi-process, storage-enabled and less isolated containers (Wang 2017). Starting from its first release in 2013, Docker has invaded the market rapidly having 100 million images downloaded already by the end of 2014 (Wootton 2017).

This thesis uses the term *containers* as a synonym for Docker containers. The reason is that this tool has been commonly used for containerization for the past few years. Virtually, Docker has gained such a widespread use that currently it is a de facto standard in the world of containers (Babcock 2015). For instance, among popular adopters of Docker containerization are Spotify, eBay, PayPal, Uber, Business Insider and The New York Times (Wootton 2017).

2.2 Containers

In order to investigate why and how a configurator should be placed into a container, containers themselves should be researched from different angles. First of all, the basic technical concepts and principles of containers are presented. After that, the advantages as well as possible challenges related to the technology are described.

2.2.1 Basic terms and definitions

Containerization technology includes many building blocks. Clear understanding of the key concepts is crucial for the proper utilization of the Docker framework. Among these concepts are containers, images, Dockerfiles, volumes and networking. Also, this chapter compares containers and virtual machines.

In order to define containers, Docker images should be explained first. According to Docker Inc. (2017), an *image* is a “lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files”. In turn, a *container* is a “runtime instance of an image” (Docker Inc. 2017). One Docker image can be used to run multiple containers.

Compared to VMs, containers provide a different level of abstraction. Figure 1 below shows the underlying structures of these two technologies. Virtual machines usually contain their own entire operating system in addition to the necessary applications. As for containers, they include only applications and share a host operating system’s kernel between each other. (Gallagher 2016, 3.) As a result, a Docker container creates an abstraction on an application level as a contrast to an operating system level abstraction of virtual machines (Docker Inc. 2017).

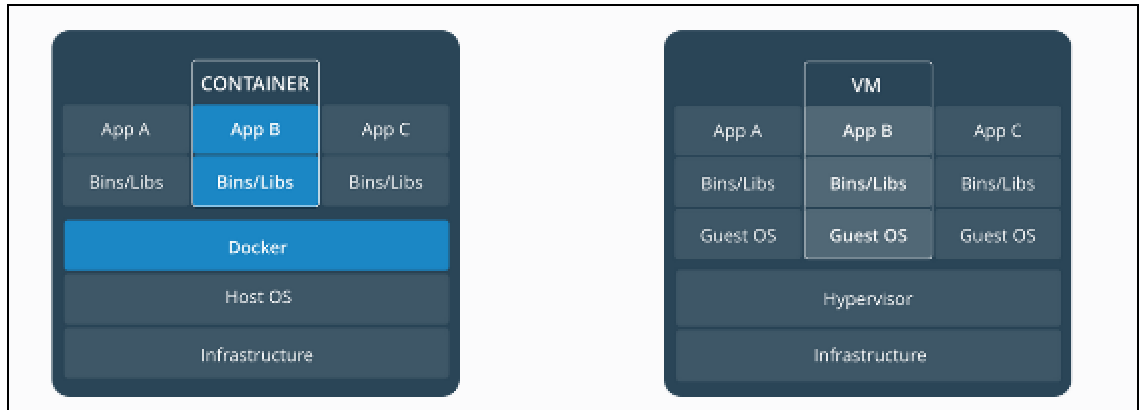

Figure 1. The architectures of containers and virtual machines (Docker Inc. 2017)

Docker images are commonly sourced from Docker Hub. According to Docker Inc. (2017), Docker Hub is a "centralized resource for container image discovery, distribution and change management". At the moment, it contains more than 100,000 images publicly available (Docker Hub 2017). As an alternative, a private *registry* can be used. *Registry* is a server application which allows storing and distributing Docker images (Docker Inc. 2017). This tool can be used by companies for internal Docker images, if Docker Hub cannot be used.

Docker images are usually built from *Dockerfiles* which contain a set of commands to be called (Gienow 2017). Example 1 shows how a simple Dockerfile could look like. Its statements denote that a standard Python image is used as a base, then the working directory is defined. Afterwards, and the Python script is copied to the image, and finally, the copied script is run. This Dockerfile can be used to build a Docker image, and consequently, to run a container from that image. As a result, this container includes everything needed to run this Python application.

Example 1. A simple Dockerfile

```
FROM python:3.7.0a2
WORKDIR /tmp
COPY ./hello.py .
CMD ["python", "hello.py"]
```

Referring to Docker Inc. (2017), a Docker image is composed of *layers*. Each *layer* corresponds to a statement in a Dockerfile. For instance, Example 1 includes three *read-only* layers (`FROM`, `WORKDIR` and `COPY`). As for the last

fourth statement (`CMD`), this layer can be changed at runtime, namely, another command can be specified for the container. When a container is run from this image, every change is written to an automatically added top fifth layer called *container layer*. When the container is removed, that *container layer* is also erased. Therefore, modifications made during the container's runtime are discarded. (Docker Inc. 2017.)

Docker Inc. (2017) state that write operations are not efficient in the context of running containers. If a considerable number of write operations is expected to be performed, it is recommended to use *volumes* rather than to write the changes to a container directly. A *volume* is a "directory or file in the Docker host's filesystem that is mounted directly into a container". Changes done inside containers automatically synch with volumes on the host and vice versa. A volume performs read and write operations with a native speed of a host. Moreover, the data written to a volume persists even after containers are stopped. In addition, one volume can be shared between multiple containers. (Docker Inc. 2017.) Figure 2 shows an example how containers can utilize and share volumes.

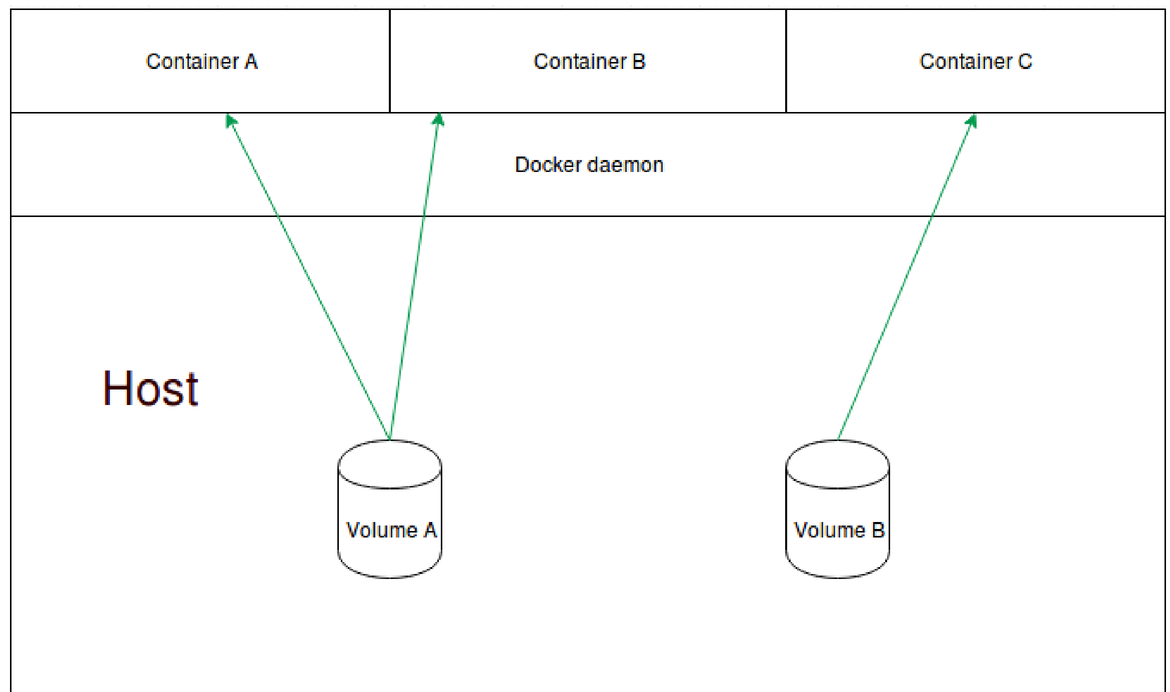

Figure 2. Docker containers' use of volumes

In addition to specific storage setup, Docker identifies its own unique set of rules for networking. By default, containers are connected to the bridge

docker0 network unless other networking options are specified at the runtime. Unfortunately, this connection method has several limitations and practices that are not recommended to be used (Docker Inc. 2017). Therefore, other alternatives can be used to achieve a reliable and efficient connectivity between containers and the outside world. For instance, custom bridge Docker networks can be created. They possess additional features compared to the default *docker0* bridge. (Slash Docker 2017.) Another option is to use host networking. It completely removes network isolation between the container and the host OS. (Hausenblas 2016.) Therefore, a container runs directly on the network stack of the host machine (Docker Inc. 2017).

2.2.2 Advantages and challenges

Nowadays, software products can gain many benefits by adopting Docker containers technology. The list of the most significant ones is the following:

- According to Docker Inc. (2017), containers are lightweight and fast. The reason behind it is that a kernel is shared. As a result, compute overhead and RAM usage are reduced. In addition, image layers are organized to be shared, cached and reutilized. Therefore, disk utilization is getting more efficient as well. (Docker Inc. 2017.)
- Another advantage from Gienow (2017) is that Docker provides strong isolation for containers. A host machine does not need to engage into processes inside containers as well as containers do not need to rely on operations and configurations of the host (Geinow 2017).
- As Gallagher claims (2016, 3), containers are highly portable and can be run in various environments. This is achieved on the grounds that the application is packed to the container with all the required dependencies and configurations at the build time (Docker Inc. 2017).
- Docker speeds up a development cycle by enabling quick roll-backs and encourages experiments inside a team of developers. Such an advantage is caused by the fact that containers can be deployed, run and discarded much faster than virtual machines. Additionally, Docker containers are relatively easy to use. Therefore, developers might focus more on an implementation of a code base rather than being concerned with the infrastructure beneath it.

Although containers appear as an appealing technology to distribute and deliver applications, three considerable challenges exist:

- First of all, Docker containers have multiple security aspects to be taken into account (Medium 2016). From operations point of view, it is crucial to be aware of the possible vulnerabilities and harden containers as well as underlying infrastructure from all the possible angles.
- Setting up and maintaining network connections in the context of containers is quite different from doing that with physical or virtual

machines (Medium 2016). Moreover, additional challenges might come up when containers are plugged into existing infrastructures or hidden behind firewalls.
- There are many considerations to bear in mind when it comes to deploying containers into production. In such cases, it is crucial to build high competence around containers for operations teams. Otherwise, containers could be implemented in an unsustainable way leading to even more complexity and poor functionality rather than efficient and fast container-based infrastructures.

The points above indicate that Docker requires proper learning and preparation. Nevertheless, it is a powerful containerization tool. Docker is able to bring many improvements to infrastructures and applications.

2.3 Developing containers

There are multiple practices and recommendations involved into the process of developing containers for applications. Some of the guidelines are provided in this section. Additionally, the containerization of existing applications is considered separately.

2.3.1 Developing applications in containers: best practices

Containerization provides a lot of freedom in the ways how it can be implemented and utilized. However, common best practices should be taken into account in order to use the technology as efficiently as possible. There are recommendations regarding which applications suit better for containerization. Additionally, there are certain Docker practices which are advised to be avoided. The specific guidelines can also be followed for the content of images and their relations. Lastly, the security of containers can be improved by applying common hardening techniques.

According to Docker Inc. (2017), stateless applications are the most suitable for containerization. For instance, such containers can be safely discarded at any point of time, because they do not store any state information. In addition, it is a common practice to run only one process per a Docker container. However, an exception might be made for tightly coupled components in order to make upgrades and deployments easier by running them in one single container. (Docker Inc. 2017.)

Unfortunately, certain practices can lead to significant difficulties in the context of Docker. One of the reasons is that containers can be considered as another type of virtualization technology. However, Coleman (2016) recommends perceiving it rather as an application delivery technology to avoid misconceptions and misuse. In addition, Benevides (2016) defines several operations that should be avoided for Docker. In the first place, he recommends keeping in mind that containers are disposable. At any point of time a container might be stopped, discarded and another instance can be run as a replacement. Therefore, the images should be built in a way that such a dynamic behavior is possible. For the same reason, he does not advise storing any data inside a container, since it can be lost if this container is stopped. One more recommendation from him is to remember that containers should be immutable. Therefore, an application should not be deployed into a container when the latter is already running. (Benevides 2016.)

Image inheritance is a significant part of Docker and promoted by Docker Inc. (2017). It brings many advantages and three of them are listed below:

- One parent image can be reutilized as a base for many children. As a result, images are simple to maintain and general image structure is clear.
- If a parent image has several children, any change in that parent is automatically applied to all the inheriting images. Therefore, an image hierarchy enables introducing new functionality without a need to change multiple Dockerfiles in parallel.
- Image rebuild is more efficient if there are changes only in one of the top children. For example, if one child contains a source code of an application, highly probable that it will be modified the most. Therefore, each change to the code triggers only the child image’s rebuild instead of one massive image with multiple levels in case inheritance is not applied.

Figure 3 shows an example of Docker image inheritance. On the right there is an image with a Java server application which inherits a Tomcat image which, in turn, inherits a Java image which, finally, inherits an Ubuntu image.

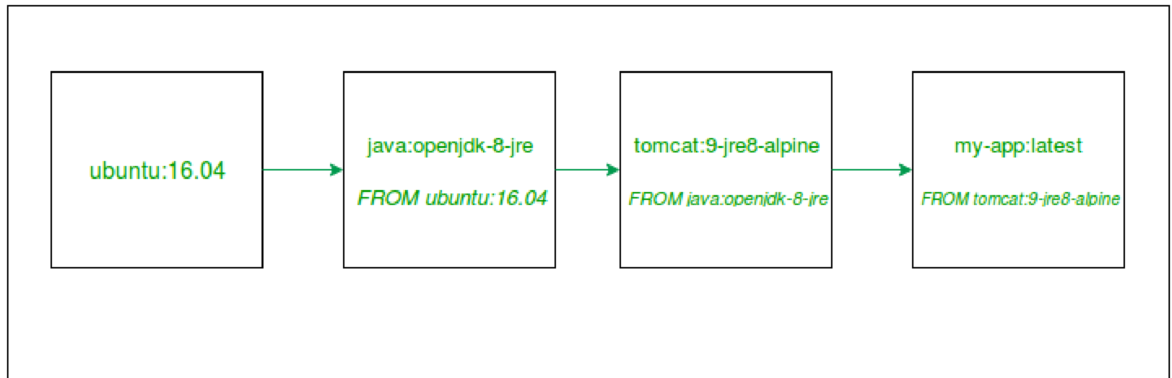

Figure 3. The Docker image inheritance example

Docker Inc. (2017) defines three types of Docker images to help identifying what exactly should be inside a container:

1. *Base image*: This type contains only middleware and necessary dependencies. It provides the highest *flexibility* (ability to modify easily according to an environment) and can be used in majority of cases. However, it does not feature *portability* (ability to be utilized in any environment without any modifications), security and traceability in great extent. An example of such an image could be a Tomcat container.
2. *Release image*: It includes everything from a base image described above in addition to release artifacts and generic configurations applicable to any environment. This type is highly recommended in most of the cases; because it manages to maintain the proper balance between portability and flexibility. Tomcat with a Java application file (*.war* extension) could be one of the examples of a release Docker image.
3. *Environment image*: It contains everything from the release image in addition to configurations specific to a certain environment. This type features the highest portability and security. However, it is not significantly flexible and might require the maintenance of multiple Docker images for each environment. One of the examples of an environment image might be Tomcat, a Java application file and a configuration file (*.xml* extension) altogether packed into one container. (Docker Inc. 2017.)

Figure 4 displays these three types in relation to each other.

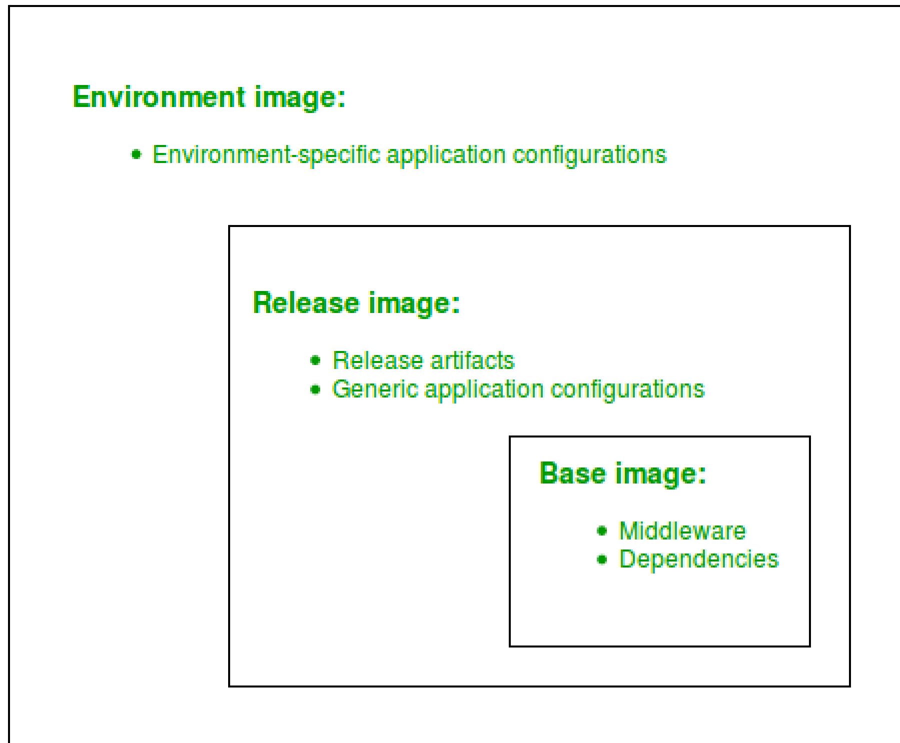

Figure 4. The classification of Docker images

The three types of Docker images described above define which configurations are directly included in an image. According to Tehranian (2015), this practically means that these configurations are copied into an image at build with a `COPY` statement in a Dockerfile (Example 2). This ensures that all the environments have the same configurations. However, if these configurations must be changed, the Dockerfile should be modified, and therefore, the image should be rebuilt. In some cases rebuilding an image might be infeasible.

Example 2. Configuring the application with the `COPY` statement

```
FROM python:3.7
COPY ./app.py /tmp/
COPY ./config.ini /tmp/
CMD [“python”, “/tmp/app.py”]
```

Tehranian (2017) suggests two other alternative ways to configure an application inside a Docker container. The first one is to set environment variables and provide them to a container, when it is run (Example 3). Thus, configurations can be defined at runtime and vary depending on the environment a container is started in. However, if containers can be

configured differently, some undesirable mismatches can appear between environments. Moreover, certain types of configurations are too complex to be set up using environment variables. In such cases the second alternative could be used, namely, plugging configurations via volumes mounted to containers (Example 4). Similarly to environment variables, this way enables configuring containers at runtime. Nevertheless, this method requires that a configuration file is present in a specified volume no matter which environment a container runs in. Otherwise, the Docker container might be non-functional. (Tehranian 2017.)

Example 3. Providing configurations via environment variables

```
> docker run -e HOST=localhost -e SILENT_MODE=true my-python-
app:latest
```

Example 4. Providing configurations via Docker volumes

```
> docker run -v /home/test/config.ini:/tmp/config.ini my-python-
app:latest
```

Apart from configurations of applications, the source code itself should be included in a Docker image. The files with code can be copied to a Dockerfile with two statements. The first one is the `COPY` command. It is able to copy a file from a host to a container's defined directory (Higginbotham 2016). `ADD` is another statement available for Dockerfiles. Similarly to `COPY`, it is able to duplicate files from hosts to containers. Additionally, it features archive extraction and remote URLs fetching. Docker recommends using the `COPY` statement, since it has more explicit functionality compared to the `ADD` command. (Docker Inc. 2017.) Meanwhile, the `ADD` statement suits well for automatic extractions of *tar* files. Example 5 shows how both statements can be applied in Dockerfiles.

Example 5. The `COPY` and `ADD` statements

```
FROM ubuntu:16.04
COPY foo.txt /tmp/
ADD bar.tar.bz2 /tmp/
ADD http://test.com/testfile.txt /tmp/testfile.txt
CMD [“cat”, “/tmp/testfile.txt”]
```

Security is another significant topic to consider in order to successfully build and run Docker containers. There are several attack surfaces, for instance, from kernel and Docker daemon sides. However, Docker claims their containers to be secure by default, if processes inside them are run by non-privileged users. (Docker Inc. 2017.) Additionally, Benevides (2016) recommends not to store credentials inside Docker images.

2.3.2 Migrating existing applications to containers

Creating a container of an existing application introduces several challenges. For instance, legacy applications usually have a heavily restricted stack and regulated set of tools. These components should generally be preserved and cannot be replaced with ones which might be more suitable for a transformation into containers. One more challenge is that it is commonly required to persist current functionality despite containerization. Moreover, the existing application may need kernel patches which could be incompatible with Docker, where the kernel is shared between multiple different containers. These and other possible limitations make the process of migrating existing applications to containers more demanding and elaborate.

There is a set of aspects from Ellithorpe (2016) to analyze before moving the legacy product into a container:

- Ellithorpe recommends to monitor resources consumption for this exact application in order to see how it matches with the Docker environment. Those resources could be, for instance, CPU, memory, disk space and network throughput.
- The current network needs to be analyzed, for instance, which ports are used by the software.
- One should check connections to external services in case certain credentials or certificates are required to be available for a container.
- Ellithorpe suggests tracing where the application writes files and logs in order to match it with future Docker volumes and logging mechanisms.
- The software usually requires certain dependencies and libraries which should be included in a Docker container as well.
- The runtime environment of the product should be analyzed with regard to duplicating it for running containers. (Apcera 2016.)

While distributed applications are containerized more commonly, monolithic existing products can be enhanced by Docker containers as well. However, monolithic applications usually include tightly coupled components, and

therefore, changing one of them could influence the whole architecture. Hence, such a case requires careful planning of the migration's implementation. First approach could be to treat a legacy application as a backend and create containers on top of it (Khan 2017). It reduces risks since an existing stack is retained and continues delivering value (Posta 2017). Consequently, old components can be discarded gradually from the monolith product as soon as the containerization is achieved. As an alternative approach from Khan (2017), the legacy application can be completely rewritten to adopt the Docker platform. However, it will require an extensive analysis to map the existing features to the new system before the old one is shut down (Khan 2017).

2.4 Configuration management systems

Configuration management systems are the target of the containerization in this thesis. Their background and principles are described in this chapter. Ansible is explored in a separate section since it is used in the Ericsson's product.

2.4.1 Overview

Configuration management (CM) is a process which handles the changes inside an IT infrastructure while keeping persistent integrity in the system. In this process, the necessary configurations for hosts are usually predefined and maintained in provisioning scripts. Commonly, they are automatically spread across machines and remotely executed from a centralized host. This process can be defined as a *server orchestration.* (Heidi 2016.)

There are several CM tools on the market. Commonly utilized alternatives include Ansible, Chef, and Puppet (Venezia 2013). All of them are capable of enforcing predefined states of hosts with provisioning scripts (Heidi 2016). Therefore, these tools provide a solution which simplifies infrastructures' configuration and maintenance. Also, they can scale from dozens to thousands of servers (Venezia 2013).

Usually, CM tools rely heavily on the concept of *Infrastructure as Code* (IaC). This conception means that infrastructures are defined in source code, and

therefore, obtain the same properties as software systems (Fowler 2016). These properties can, for example, include ability to version control, continuously integrate, perform tests and get reviews from peer developers (Puppet 2017). All in all, the IaC principle makes states of servers consistent, maintainable and auditable (Fowler 2016).

2.4.2 Ansible

Ansible is used as a main CM tool in the Ericsson's product. Therefore, it is described here in greater details. As many other configurators, Ansible allows defining the states of hosts in easy-to-read text files and automatically enforcing these states throughout the infrastructure. However, the tool stands out by featuring a *clientless architecture*. This design conveys the idea that target hosts do not need any Ansible installed in order to be able to accept instructions from the master node. Hence, the controller host is the only one which is required to contain Ansible software and configuration files (Daniel 2013, 21). However, Ansible clients traditionally need to have Python installed as well as to be accessible over SSH (Zanzane 2016). Other types of connection are possible in Ansible as well, however, SSH is the default one (Red Hat 2016).

Ansible defines configurations of hosts in files called *playbooks.* These files utilize *YAML* (Yet Another Markup Language) which is simple to read without any previous knowledge of it. (Red Hat 2017.) Example 6 shows how a playbook could look like. In this file, `apt` and `user` are *modules*. In principal, *modules* are small bits of codes pushed and executed on target hosts. There are over 750 built-in modules present in Ansible at the moment. (Red Hat 2017.)

Example 6. A simple Ansible playbook

```
- name: Install curl
  apt:
    name: curl
    state: latest
    install_recommends: no

- name: Add test user
  user:
    name: test
    uid: 1042
    group: administrators
```

In addition to playbooks, a master node usually contains an *inventory* file, which contains a list of hosts to be managed by Ansible. The nodes can be unified into *groups*. (Red Hat 2017.) Example 7 shows how an Ansible inventory file can be composed.

Example 7. An Ansible inventory file

```
[frontend]
www.test-frontend.com
www.test-frontend2.com

[backend]
www.test-backend.com
www.test-backend2.com

[database]
www.test-db.com
```

Example 7 displays an inventory file in the *INI* format. Additionally, an inventory can be specified with *YAML* (Red Hat 2016).

2.5 Running a configurator in a container

In theory, any configuration management tool can be run in containers. However, the containerization might cause additional challenges since each software has its own characteristics. For instance, the tools like Puppet and Chef make client nodes initiate pulling of configurations (Pillai & Chef Docs

2017). Therefore, the hosts should be always able to reach the master server. However, such a persistency is challenging with containers, which might be discarded and replaced with new instances multiple times. Another fact hindering the containerization could be that a Chef master host stores the current state of the nodes (Chef Software 2014). As a result, Chef software is stateful which requires additional effort in a containerized environment.

Ansible is a suitable candidate for containerization. First of all, the tool normally consists of one process. This process is commonly an operation of playbooks' execution. Additionally, Ansible is not write-intensive while operating. Usually, configuration results are registered only into a console unless logging file directory is specified (Red Hat 2016). Moreover, Ansible can be claimed to be a stateless tool. Most of its modules feature *idempotence* which is a property of running a task multiple times without changing the final result (Bernardes 2016). Therefore, it is possible to run Ansible playbooks repeatedly without relying on states of the target hosts. As a result, the host containing Ansible software requires no knowledge of the state of the client nodes. Additionally, the Ansible master distributes configurations to the hosts as code executables in a one-way fashion. Hence, the target nodes do not rely on the main Ansible machine and the playbooks can be run from different stateless instances of containers.

3 PRODUCT ARCHITECTURE AND EVOLUTION

The Ericsson product utilizes Ansible as a main configuration management system, and has a comprehensive base of playbooks to define all the needed configurations. These playbooks are located on a dedicated VM along with Ansible and other required dependencies. This virtual machine is connected to the rest of the components to orchestrate the whole deployment of the product. This configuration method has evolved over time into a validated, mature and secure solution for the product. This solution is referred to as the *existing solution* in this thesis.

The product's build and deployment processes are crucial to be analyzed for this thesis. As researched, the build procedure installs all the software's components (including Ansible and all the playbooks) into a single virtual

machine image. At the deployment, this VM image provides the base for several virtual machines. One of these machines is defined as a *Configurator VM* which runs all the Ansible playbooks to set up the whole infrastructure (Figure 5). The initial configuration process involves removing unnecessary components from virtual machines to maintain only role-specific parts. These roles include, for instance, front-end, database and HA proxy. As soon as the roles of the virtual machines are assigned, Ansible continues with other necessary configurations in order to set up the integrated, efficient and reliable deployment of the product.

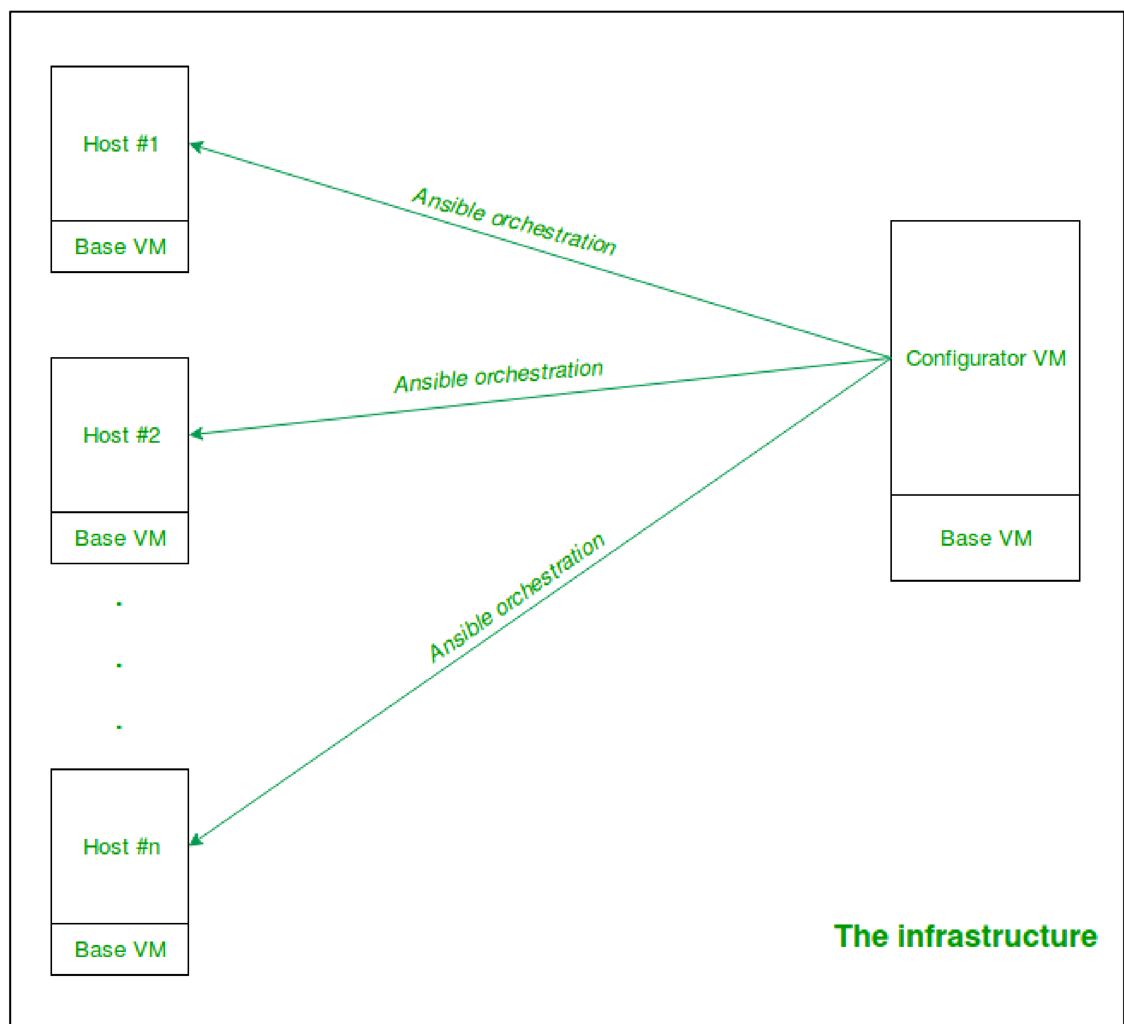

Figure 5. The role of the Configurator VM in the infrastructure

To improve the *existing solution*, this thesis suggests the containerization of Ansible as the *proposed solution*. The first advantage of this proposal is that the updates can be implemented more rapidly for Ansible compared to the current process. At the moment, the product is built within one unified process which takes considerable amount of time. As for the proposed solution, only the Ansible container has to be rebuilt and redeployed instead of the whole

image of the product. The second improvement is the maintainability. It is less straightforward to introduce unwanted changes to the playbooks, if they are inside a Docker image. Otherwise, if the playbooks are located in the virtual machine, they can be modified easily by anyone who has access to the host.

However, there is one risk related to this container migration. All the present Ansible functionality might not be supported, if the configurator is migrated to Docker. This container tool has its own unique ecosystem significantly different from server virtualization. In particular, networking might require extra effort in order to enable the container to reach the rest of the hosts in the similar fashion as it is able to do it now. Additionally, security solutions have to be re-evaluated, if they are compatible with containers, for instance, secrets management, user privileges and required passwords.

4 PRACTICAL IMPLEMENTATION

This chapter documents the process of designing, building, deploying and running the Docker container that contains the Ansible configurator. The solution solely focuses on the company's case. The main implementation method starts with a step-by-step analysis of the current setup of the product. Then, the similar configurations are created with Docker. After that, the solution is tested incrementally using the local environment. As a result, the container is expected to be able to replace the existing Ansible installation. In addition, it should decrease build time and deployment time as well as help avoiding unnecessary full-site upgrades.

The general approach chosen for this work is to implement the container on top of the existing infrastructure. This approach was mentioned in the background part as one of the options to migrate existing applications into containers. It helps keeping the current setup without any changes. Therefore, the container can be implemented as a proof-of-concept task to validate its functionality. In case of errors, no changes are required for the existing stack and only the container-related modifications added on top have to be modified or discarded.

4.1 Docker images

As stated in the background part, Docker images are commonly built from Dockerfiles. It is crucial to carefully plan statements in these files for the sake of controllable behavior and efficiency of containers. This chapter explains design choices for Docker images in scope of the company's case and its configurator.

The first decision is to create two images. Thus, all the advantages of Docker image inheritance are applied. The parent image includes only the Ansible software installed. This Docker image corresponds to the term *base image* from Chapter 2.3.1. As for the child image, it contains the Ansible playbooks and other necessary files which are utilized in the company's case. Environment-specific configurations are not included. This image corresponds to the term *release image* from Chapter 2.3.1.

In order to define the necessary statements for each of the Dockerfiles, the *existing solution* has to be analyzed. Fortunately, there are dedicated Ansible playbooks used at build time. In the scope of this thesis these playbooks are called *configs-for-build*. They define the state of the VM image which is later used as a base for all the hosts in the infrastructure. These playbooks are not the same as the Ansible files required to be inside the Ansible release image. Nevertheless, they simplify the process of defining the content of Docker images.

4.1.1 Ansible base image

Chapter 2.2.1 mentions that Docker Hub is commonly used to source Docker images. It contains multiple options for images with installed Ansible as well. However, there are three reasons to avoid using Docker Hub and to create the internal Docker image for this company's case. First of all, there is not enough control over the content of images from Docker Hub, especially if the *latest* tag is used. Then, Ansible does not officially support any Docker image anymore (GitHub 2014). Lastly, the company policies restrict the usage of public repositories such as Docker Hub.

The final goal of the base Docker image is to have the Ansible software installed. However, a few pre-steps are required to achieve this. In order to define these steps, only the statement with the Ansible installation (`apt-get install ansible`) is included in the Dockerfile in the beginning. Undoubtedly, running such a container causes multiple error messages. These messages help identifying what is missing in order to achieve a successful Ansible installation. Therefore, such a top-down approach helps finding the missing Dockerfile statements. Listing 1 below shows the contents of the final Dockerfile for this Ansible base image created by using this top-down method.

Listing 1. The Dockerfile for the Ansible base image

```
#1 FROM local-ubuntu-base
#2 ENV DEBIAN_FRONTEND noninteractive
#3 RUN echo <local_repository_link> >/etc/apt/sources.list
#4 RUN touch /etc/apt/apt.conf.d/99translations && echo
     'Acquire::Languages "none";' >>
     /etc/apt/apt.conf.d/99translations
#5 RUN apt-get update
     && apt-get install -y apt-transport-https curl
     && echo https://<local_repository_link>
         >>/etc/apt/sources.list
     && curl https://<local_key_link> | apt-key add -
#6 RUN apt-get update && apt-get install -y --no-install-
recommends ansible<+version>
#7 CMD [ "ansible-playbook", "--version" ]
```

The list below explains each statement of the Dockerfile from Listing 1:

1. `FROM` statement sets `local-ubuntu-base` as a parent image for this container.
2. If `DEBIAN_FRONTEND` environment variable is set to `noninteractive`, packages can be installed without any human intervention (Shaw 2017). As a result, console inputs are not required and an image build can be easily automated.
3. This statement copies the link with the internal company's repository to the Linux list of repositories. Later, this repository is used to pull the Ansible software.
4. One of the test's errors identified that the pull from the repository fails because of the absent translation packages. However, the *99translations* file solves the problem, when it contains configurations to cancel any acquisition of these translation packages.

5. One more repository link is needed inside the container (`echo` command). However, this link contains `https`, and therefore, *apt-transport-https* utility has to be present. Also, apt keys are required for the authentication of packages (`apt-key add`). Hence, *curl* is installed to fetch these keys from `local_key_link`. In order to install these two packages, `apt-get install` and `apt-get update` are placed in the same Dockerfile statement. Otherwise, according to Docker Inc. (2017), the repository's update can be cached by Docker and skipped on the next image builds. Therefore, the packages might fail to install (Docker Inc. 2017). As for `-y` tag, it automatically answers *yes* to all input prompts.
6. As soon as all prerequisites are met, Ansible itself can be installed. The version of Ansible is the same as the one installed in the *existing solution*. The Ansible package is archived in the Ericsson's local repository and includes all the required dependencies such as *python* and *pip*. The tag `--no-install-recommends` is used to avoid installing recommended packages.
7. The default runtime command displays the version of Ansible. Hence, it validates that the software is successfully installed.

This image can be built with the following command:

```
docker build -t ansible-base-image:local .
```

As soon as it is built, the container can be tested with the following command:

```
docker run ansible-base-image:local
```

The result of running the container is shown in Figure 6.

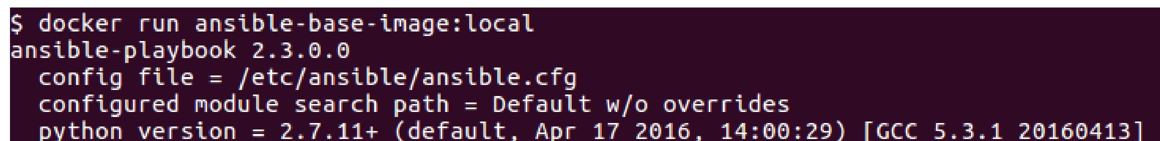

Figure 6. Running the container from the Ansible base image

As Figure 6 displays, the container is run successfully utilizing the implemented image.

4.1.2 Ansible release image

The contents of the second Docker image are selected according to the definition of a *release image* from the background part. Once again, a *release image* should include only the source code and generic configurations. If there are any environment specific configurations, they should be included in the

container at the runtime. As a result, such an image allows creating a flexible container which can be used in multiple environments. (Docker Inc. 2017.)

In order to create a functional Ansible release image for this case, the build time contents of the *existing solution* are duplicated into the Dockerfile. As mentioned before, all the contents of the build are inside the set of Ansible playbooks, called in this thesis *configs-for-build*. This set includes a separate file to specify the settings necessary for the existing configurator. The contents of this playbook are the following:

1. Copy the set of Ansible playbooks which configure the site at the deployment. In the scope of this thesis this set is called *configs-for-site*.
2. Generate one playbook for *configs-for-site* from the *j2* template. The *j2* format is parsed by Jinja2, which is the default Python template library for Ansible (Red Hat 2017).
3. Copy the Ansible inventory file.
4. Copy the *ansible.cfg* file. This file defines the Ansible settings (Red Hat 2016).
5. Copy the Ansible log rotation file. This file configures log rotation for the Ansible service (LinuxConfig 2014).

In addition to the contents defined with *configs-for-build*, the documentation of the product is analyzed. First, it states that the inventory file should be modified after the infrastructure is deployed, but before it is orchestrated with the *existing solution*. As for the inventory copied with *configs-for-build* (step 3 in the list above), it is merely a template for hostnames and IP addresses. Similarly, the documentation requires modifications of the files inside the directory *group_vars* contained in *configs-for-site* according to the deployed environment. Therefore, these two files are the runtime configurations and should be injected to this container when it is run rather than copied to the image at the build time.

Listing 2 displays the contents of the Dockerfile for this image. As mentioned before, the statements are derived from the configurations of the *existing solution*, namely, the list given above. The runtime configurations are taken into account as well in order to be excluded from this release image.

Listing 2. The Dockerfile for the Ansible release image

```
#1  FROM ansible-base-image:local
#2  ARG user
#2  ARG password
#3  RUN <adding-the-user-to-the-system>
#4  WORKDIR /home/$user
#5  RUN mkdir -p /etc/ansible/configs-for-site/roles/<role-
name>/defaults
#6  COPY main.yml /etc/ansible/configs-for-site/roles/<role-
name>/defaults/main.yml
#7  COPY <entrypoint-filename> .
#8  COPY ansible.cfg /etc/ansible
#9  COPY configs-for-site /etc/ansible/configs-for-site
#10 VOLUME ["/etc/logrotate.d/<ansible-logrotate-filename>",
"/etc/ansible/hosts", "/etc/ansible/configs-for-site
/group_vars"]
#11 ENTRYPOINT ["./<entrypoint-filename>"]
```

The statements of the Dockerfile (Listing 2) are numbered and explained as follows:

1. The previously described Ansible base image is used as a parent here.
2. The parameters `user` and `password` are passed as arguments to this image when it is built. This can be done with the `docker build` command via the `--build-arg` tag (Docker Inc. 2017). The parameters are required for the next Dockerfile statement.
3. One of the users which is present in the *existing solution* is included in this image as well. Also, it is configured with the password provided as an argument. Following the recommended security hardening techniques from Docker, this user also lacks *sudo* privileges.
4. The home directory of the previously defined user is set to be a `WORKDIR`. Thus, `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` statements use this directory as a working one (Docker Inc. 2017).
5. The missing directory of *configs-for-site* is created in the image. This directory is necessary for the next Dockerfile statement.
6. Currently, this specific *main.yml* playbook is generated from the template with Ansible at the build time (*configs-for-build*). Obviously, the resulting file could have been merely copied to the image as soon as generated by *configs-for-build*. However, extra dependencies are better to be avoided between the current product’s build and the build process for the container. Therefore, the file is generated for Docker independently. A simple Python script is created to utilize Jinja2 Python library and to generate the file from the source. This Python script is executed outside this Dockerfile.
7. The *entrypoint* script is copied to the Docker image. Its purpose and contents are described at the end of this list.

8. The *ansible.cfg* file is copied to the Docker image. In general, this file specifies the configurations for Ansible, for instance, the inventory location and the default port. Therefore, *ansible.cfg* can be defined as a runtime configuration. However, in this case the file is the same for every environment. Thus, it is copied to the Docker image at the build time.
9. Ansible playbooks (*configs-for-site*) are copied inside the container. The statement is placed in the lowest possible position in the Dockerfile since this layer is expected to be modified most often. Therefore, the layers above this statement do not have to be rebuilt due to this single change if the Docker image caching is used (Docker Inc. 2017).
10. The volume mounting is chosen to be a method to inject the runtime configurations. This method is selected due to the reason that these configurations are stored as files and mounting suits well for files. As mentioned above, these files are the inventory and the group variables. Additionally, */etc/logrotate.d/<ansible-logrotate-filename>* is identified as a runtime configuration, since this file can be different depending on the environment.
11. The last statement of this Dockerfile includes a script copied in the statement #7. This script consists of all the commands required to run the Ansible playbooks for the orchestration of the product's infrastructure. Listing 3 shows these `ansible-playbook` commands. In order to run the script when the container starts, the `ENTRYPOINT` statement is used. According to Docker Inc. (2017), this statement allows running the container as executable. Moreover, the arguments for that script can be provided directly via the `docker run` command (Docker Inc. 2017). Additionally, according to DeHamer (2015), `ENTRYPOINT` layer has less straightforward mechanism to be overwritten compared to `CMD`. Therefore, it is commonly used when exclusive behavior is expected for a container (DeHamer 2015).

Listing 3. Docker's *entrypoint* script for the Ansible release image

```
ansible-playbook -Kk -t <network-tag> /etc/ansible/configs-for-
site/site.yml
ansible-playbook -Kk /etc/ansible/configs-for-site/<ssh-playbook-
name>.yml
ansible-playbook /etc/ansible/configs-for-site/site.yml
```

All in all, such a Docker image includes all the playbooks and build time configurations needed for the orchestration of the hosts in this product. Additionally, it defines runtime configurations as volume points which are mounted when the container is run.

4.2 Building the Docker images

Three aspects have to be determined regarding the arrangement of the container build process in this product. First, it must be defined clearly what exactly is planned to be built and in which order. Then, the build location is required to be selected. Finally, the build tools have to be chosen. All the decisions should follow the *on-top-of-legacy* method chosen in the beginning. Also, the build of the Ansible containers should be aligned with other container-related tasks which are occurring in the development team of the product at the moment.

The objects of the Docker build process are two Docker images described in Chapters 4.1.1 and 4.1.2. The composed Dockerfiles are used as an input. The Ansible base image is a parent of the Ansible release image. Therefore, the former one should be built and present on a host before the latter one is created. As soon as the images are built, they are archived to the *.tar.bz2* files. The reason for archiving is that the VM image is currently delivered to customers as a file. Therefore, the Docker images are planned to be delivered as files as well.

It is decided to use one of the existing VMs as a location for the build of the Ansible images. At the moment, this VM runs the automated common build process in local and CI (Continuous Integration) environments. One of the reasons to choose this VM is that it produces the internal *local-ubuntu-base* image which is required as a parent for the Ansible base image. Also, other product's containers are built at this location by the rest of the development team, and therefore, the Ansible images can be kept aligned.

To control the build of the Docker images, Makefiles are chosen in this case. The reason is that the team already performs the build with the set of Makefiles in order to create all the components of the product including the containers. According to Free software foundation (2017), *Makefiles* are the files which are used for the *make* utility. This utility is able to automatically recompile the parts of large software (Free software foundation 2017).

Listing 4 shows the part of the Makefile for the build of the Ansible release image. In general, the implemented Makefile for the Ansible base image follows the same principles, and therefore, is not presented in the thesis.

Listing 4. Makefile rule for building the Ansible release image

```
$(DOCKER_IMAGE_COMPRESSED): $(PARENT_TIMESTAMP_FILE) Dockerfile
Makefile $(ENTRYPOINT_SCRIPT) $(shell find $(CONFIGS_FOR
SITE_LOCATION))
	cp -r $(CONFIGS_FOR_SITE_LOCATION) .
	cp $(ANS_CNF_SRC_FILEPATH) .
	cp $(LOGROTATE_SRC_FILEPATH) .
	python $(PYTHON_SCRIPT_FILEPATH)
	docker build -t $(IMAGE_NAME):$(IMAGE_TAG) --build-arg
user=$(USER) --build-arg password=$(PASSWORD) .
	docker save $(IMAGE_NAME):$(IMAGE_TAG) | pbzip2 -9 -c > $@
	rm -rf ./$(CONFIGS_FOR_SITE_DIR)
	rm ./$(DEFAULTS_FILEPATH) $(ANS_CNF_FILE) $(LOGROTATE_FILE)
```

The first aspect to consider in Listing 4 is the state when the image is rebuilt. The general intention is to avoid excessive rebuilds in order to save build time and exclude unnecessary images in the Docker cache. Therefore, the image is built only when the parent image, the Dockerfile, the Makefile, the *entrypoint* script or *configs-for-site* playbooks have changed.

This build process needs to have access to all the files included in the Dockerfile with the `COPY` statement (Listing 2, p. 28). Instead of specifying the original location of the files, the Makefile temporarily places these files to the same directory where the Dockerfile is. The reason is that according to Docker Inc. (2017), Docker daemon recursively builds the context at a location specified by `PATH` *or* `URL` in the `docker build` command. Then, this context is entirely transferred to the daemon from the hard drive. If this location includes many files, the context takes considerable time to be built. (Docker Inc. 2017.) Therefore, it is a common practice to set the context at the directory where the Dockerfile is located.

The Makefile is also responsible for the template generation mentioned in Chapter 4.1.2. It runs the implemented Python script to create the playbook

(*main.yml*) needed for the Dockerfile. The output of the script is the file defined with the `DEFAULTS_FILEPATH` variable. When the Docker build is finished, this file is removed.

As soon as all the pre-requisites are met, Docker is able to build the image using the Dockerfile from Listing 2 (p. 28). This Dockerfile includes two `ARG` statements. Therefore, the required arguments are provided with the `docker build` command using the `--build-arg` tag. Then, the built image is archived via the `docker save` command. The resulting file is compressed with the *pbzip2* utility.

4.3 Running the container

This chapter focuses on how the container is handled at the runtime. Similarly to the build process, the implementation requires the definition of what, where and how the container should be deployed. Then, there are four sub-sections focusing on the additional deployment aspects for containers. In the first section, the networking implementation is discussed in order to explain how the container can reach other hosts. After that, secrets management is described, namely, how the necessary SSH keys can be injected into the container in this case. The next section illustrates which issues are identified during the first deployments of the container and how they are solved. Lastly, the step-by-step documentation is presented for the process of replacing the *existing solution* with the implemented container.

The final deployment object is the Ansible release image containing the playbooks and the configurations. It uses the Ansible base image as a parent. However, when `docker save` is used, the archive of the top level child image is enough to run the container. Therefore, only the archive of the child is needed at the deployment's location in this case.

The *Configurator VM* is chosen as a location to run the container. This VM was mentioned in Chapter 3. The reason for using the *Configurator VM* is that it is already a part of the infrastructure and has all the necessary settings to quickly run and test the container on top of it against the whole site. Additionally, if the existing VM is used, there is no need to create a new host

for the container only. Therefore, it helps avoiding excessive integration work and keeps the existing stack untouched.

Unlike in the build environment, the deployment process cannot be aligned with other containers implemented by the development team. It is justified by the fact that all the containers are virtually deployed and run with the Ansible playbooks contained inside the target container of this work. Obviously, it is impossible to deploy a container using the same container which does not exist yet. Therefore, the separate deployment strategy has to be used for this case.

There are multiple tools and advanced frameworks to deploy and orchestrate Docker containers. However, a simple shell script suffices in this case, as shown below in this chapter. It is enough at the moment, since there is only one Docker image and only one instance of a container. Moreover, the script can be easily integrated in the current deployment process.

Currently, there is a sequence of scripts used by the team for the deployment of the product. They simplify and automate the process of software installation and configuration. Moreover, they can be used in local developer machines as well as in CI. Among this set of scripts there is one shell script which can be utilized for the deployment of the container. The reason is that it invokes the Ansible playbooks (*configs-for-site)*. In other words, it runs the commands included to the Ansible release image's *entrypoint* (Listing 3, p. 29). Therefore, this script can be used as a location for the commands to deploy and run this Ansible container. It allows testing the container immediately inside the existing deployment process.

Virtually, the deployment of the container is implemented with two commands in this case. The first one is to load the Docker image from the archive to the Docker daemon. This is done with the `docker load` command. The second step is to run the container. The command should include all the necessary volume tags according to the specifications of the image (Listing 2, p. 28). For the sake of security, files and directories are mounted with read-only access. Therefore, the container is not able to modify the files on the host but only to

read them. (Docker Inc. 2017.) The resulting `docker run` command is shown in Listing 5.

Listing 5. Running the container from the Ansible release image

```
> docker run \
     -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
     -v /etc/ansible/hosts:/etc/ansible/hosts:ro
     -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
     ${IMAGE_NAME}:${IMAGE_TAG}
```

The update mechanism should also be implemented. It is required if a new version of the container has to be deployed to the site. It is decided to take four steps in order to accomplish this:

1. Rebuild the image utilizing the Makefile described above (Listing 4, p. 31). Therefore, the update happens when there is a change in any of the files the container depends on.
2. Remove the old instance of the Docker image from the *Configurator VM*. The command `docker rmi -f` is applied in order to achieve that.
3. Upload the new instance of the image packed into the archive to the running *Configurator VM*. This can be accomplished with the *scp* utility.
4. Load the new version of the image from the new archive at the *Configurator VM* using the `docker load` command.

A script is implemented in order to automate these four steps. This script is able to run, deploy and update the container. Additionally, it can optionally be invoked inside the existing shell script mentioned above. The new script for this Ansible container can also be used separately from the rest of the deployment scripts in case there is a running site containing the *Configurator VM*. Therefore, it is not necessary to rebuild and redeploy the whole site, if only the Ansible part has to be updated, namely, the version of the Ansible or the *configs-for-site* playbooks.

4.3.1 Networking

Currently, the product uses a specific network plan. This plan consists of several networks created for different purposes. The *existing solution* uses one of these networks to orchestrate the rest of the hosts with Ansible.

Initially, the configurator runs the playbooks using the password authentication. After that, Ansible configures the SSH key. Then, the Ansible playbooks use these keys and do not require password input anymore.

In order to make the container able to reach the rest of the nodes, host networking Docker option is utilized for this case. The first reason is that this method can connect the container immediately to the existing network. Additionally, no changes are required to the current network plan nor the Docker image. Another reason is that the product currently uses *firewalld*. The development team has detected that this utility conflicts with Docker default networking significantly. As for the host Docker networking, it helps overcoming this issue related to *firewalld*.

Nevertheless, there are a few disadvantages of connecting the container directly to the host network. First of all, this Docker container is directly exposed to the network interface, which could lead to security implications. Additionally, only one instance of the container can be running at the host at the time. No other containers as well as the *existing solution* are able to perform the playbooks at this point. However, despite these drawbacks, the host Docker networking can be used as a start solution to run and test the container.

The Ansible container can be tested for the connectivity with a simple Ansible ad-hoc command. In this case, the implemented Docker container is run from the *Configurator VM* on the running site with the command of Listing 6.

Listing 6. Running the container with host networking

```
> docker run \
    -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
    -v /etc/ansible/hosts:/etc/ansible/hosts:ro
    -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
    -it \
    --entrypoint=/bin/bash \
    --network=host \
    ${IMAGE_NAME}:${IMAGE_TAG}
```

The command is similar to the one in Listing 5 (p. 34). In addition to that, the –-network tag specifies host networking as initially intended. Also, -it and --entrypoint=/bin/bash are used in order to enter the terminal of the container. From this terminal the ad-hoc Ansible command can be run to check the connectivity as follows:

```
> ansible all -m shell -a 'echo test' -Kk
```

This command's intention is to reach all the hosts from the inventory file. It executes the simple echo command via the *shell* Ansible module. The tag -Kk forces prompting SSH and *sudo* passwords for the authentication and the required privileges respectively. Figure 7 shows the results of this command.

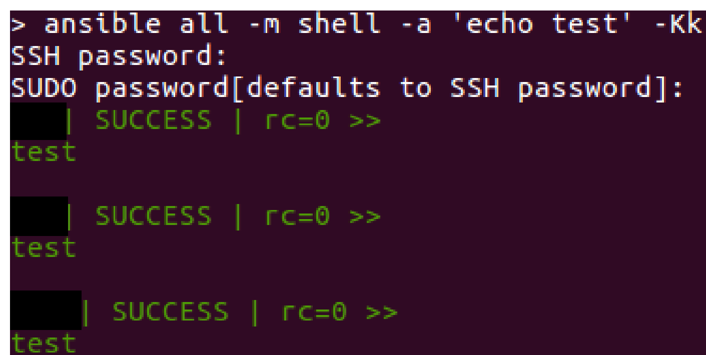

Figure 7. Reaching other hosts at the site with the Ansible container and the ad-hoc command

As Figure 7 displays, Ansible inside the container successfully manages to reach all the hosts from the inventory with the ad-hoc command. Therefore, host networking is an operational connectivity method for containers in this case.

4.3.2 Secrets management

Apart from proper network connectivity, the Ansible container should be configured with correct SSH keys. This requirement arises at the last command of the container's *entrypoint*:

```
ansible-playbook /etc/ansible/configs-for-site/site.yml
```

This command runs the main set of Ansible playbooks for this case (`site.yml`). Also, it does not have the `-k` tag. Therefore, it uses the default authentication method instead of prompted passwords. In case of Ansible, the default authentication is performed with SSH keys (Red Hat 2016). If the keys are absent, the configurator is not able to authenticate into target hosts and, consequently, run the required playbooks. The valid SSH keys are currently present in the *Configurator VM*, however, the container does not have access to them yet.

There are four requirements determined for providing SSH keys for this container. First of all, the provisioning should be executed as securely as possible since SSH keys are the sensitive data. Secondly, the container should be always able to access the required keys. Thus, there is no interruption during the runtime of the playbooks. The third requirement is portability. The container has to function equally in any environment regardless the SSH keys. Lastly, the chosen solution has to be suitable for the existing stack of the product and should not impair its current functionality.

There are multiple methods to inject SSH keys into Docker containers. Three of the most common ones are presented and analyzed in the following list:

1. According to Docker Inc. (2017), it is recommended to configure secrets management with Docker Swarm. In such a case, the secrets are directly available only to this Docker cluster management system. The framework also handles encryption of these keys. Additionally, it distributes the secrets only among containers which have privilege to access this sensitive data. (Docker Inc. 2017.) All in all, the solution provides a robust security and availability of keys as well as portability of containers. However, Docker Swarm requires considerable changes in the existing stack of the product. Additionally, it does not align with other container-related tasks at the moment.
2. The SSH keys can be merely copied to the Docker image at the build time (Stack overflow 2013). Therefore, the container always has the access to the required secrets. However, such an option is not secure enough since the keys are written to one of Docker image layers. Therefore, these keys can be still present in Docker daemon even when a container is removed (Stack overflow 2013). Moreover, the container loses portability due to the keys injected at the build time. The reason is that different sites might require different key pairs. As for this product case, the SSH keys are not generated during the build time but at the time of deployments. Hence, this solution requires considerable changes to the current internal processes for the build and the deployment.

3. Another option is to plug the secrets via the Docker volumes (Stack overflow 2013). In this case, the image does not contain any SSH-related configurations inside and can be used in any environment. In addition, this solution does not require any modification to the current product's stack. However, the container might lack the required keys if the underlying host does not contain them in the expected directory. As for security, the running containers might be exploited to get access to the secrets. However, the volumes can be at least configured with read-only access (Docker Inc. 2017). Thus, it is impossible to change SSH keys at the host via Docker containers.

All in all, among all the solutions mentioned above, the last one suits this case the most at the moment from the security, availability and portability perspectives. Therefore, the new mount point should be added to the Dockerfile's `VOLUME` statement:

```
VOLUME ["/etc/logrotate.d/<ansible-logrotate-filename>",
"/etc/ansible/hosts", "/etc/ansible/configs-for-site/group_vars",
"<SSH-keys-at-container>"]
```

Additionally, the directory with the SSH keys at the host should be mounted to the container at the runtime. Similarly to other volumes, only the read access is granted as Listing 7 shows.

Listing 7. Mounting SSH keys.

```
> docker run \
    -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
    -v /etc/ansible/hosts:/etc/ansible/hosts:ro
    -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
    -v <SSH-keys-at-host>:<SSH-keys-at-container>:ro \
    --network=host \
    ${IMAGE_NAME}:${IMAGE_TAG}
```

Such an update to the Dockerfile and the `docker run` command allows injecting the correct SSH keys to the container. Consequently, the last `ansible-playbook` command of the *entrypoint* (Listing 3, p. 29) can use the default authentication method.

4.3.3 Identifying deployment issues

Several issues have been detected during the first deployments of the container. These errors require the relevant fixes in the Docker image as well as the additional tags for the `docker run` command. The possible solutions can be identified by analyzing the current setup of the *existing solution*. For instance, the contents of the directories, the file permissions and the configuration files can be reviewed at the host containing Ansible. Then, they can be duplicated in a relevant way to the container in order to solve the issues.

The first error was detected during the build time of the image. Figure 8 shows the output in the terminal.

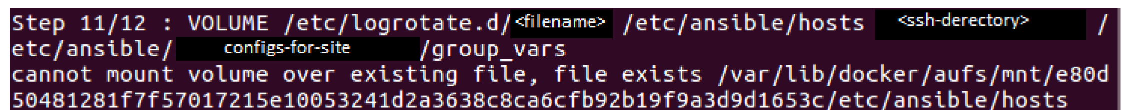

Figure 8. Error message for mounting volumes over existing files

The reason is that Docker does not allow creating volume points over existing files. This is the case for the */etc/ansible/hosts* inventory file which is mounted to the Ansible release image. It is automatically created by the parent image at the build time during the installation of Ansible. However, it is a runtime configuration file and has to be changed according to the environment. The problem can be solved by removing the inventory file before the `VOLUME` statement in the Ansible release image's Dockerfile:

```
RUN rm /etc/ansible/hosts
```

Obviously, the container becomes dysfunctional, if this file is not mounted. However, the absent inventory file is beneficial from the user experience point of view. In such a case, the container explicitly raises the error that this file is missing. Therefore, it becomes clear to the user that the correct inventory file should be mounted to the container. Otherwise, if the hosts file is present in the Docker image, the Ansible container continues with its tasks until facing the problem with unreachable hosts. This error could be caused by multiple

reasons apart from the missing inventory. Therefore, the user could take unnecessary troubleshooting steps.

As soon as the image is successfully built, the running container raises the warning. It states that the *<logfile-path>* cannot be created (Figure 9).

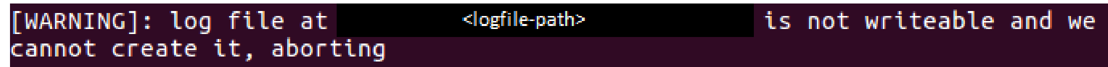

Figure 9. The warning regarding the missing log file

The reason the container needs this log file is that the logging is activated by *ansible.cfg.* However, the *<logfile-path>* directory is not present inside the container. It is decided to eliminate the error by editing the `VOLUME` statement in the following way:

```
VOLUME ["/etc/logrotate.d/<ansible-logrotate-filename>",
"/etc/ansible/hosts", "/etc/ansible/configs-for-site/group_vars",
"<SSH-keys-at-container>", "<logfile-path>"]
```

Also, the `docker run` command should be modified accordingly (Listing 8).

Listing 8. Mounting the logging directory

```
> docker run \
    -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
    -v /etc/ansible/hosts:/etc/ansible/hosts:ro
    -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
    -v <SSH-keys-at-host>:<SSH-keys-at-container>:ro \
    -v <logfile-path>:<logfile-path> \
    --network=host \
    ${IMAGE_NAME}:${IMAGE_TAG}
```

The volume is chosen instead of `COPY` in the Dockerfile, since the container can discard all the files it contains as soon as it finishes running the playbooks. As for mounting the directory, the logging information is written to the host directly. Therefore, the logs are retained despite the lifecycle of the container.

The next error encountered is an inability to input SSH and *sudo* passwords when they are prompted. As mentioned before, this input is required by the first two `ansible-playbook` commands of the *entrypoint* script (Listing 3, p. 29). The reason of the prompt issue is that the Docker container runs in the non-interactive mode. In order to open *stdin*, the `--interactive` (`-i`) tag can be used with the `docker run` command. Therefore, this command can be modified accordingly (Listing 9).

Listing 9. Running the container in the interactive mode

```
> docker run \
     -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
     -v /etc/ansible/hosts:/etc/ansible/hosts:ro
     -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
     -v <SSH-keys-at-host>:<SSH-keys-at-container>:ro \
     -v <logfile-path>:<logfile-path> \
     -i \
     --network=host \
     ${IMAGE_NAME}:${IMAGE_TAG}
```

The next error relates to the failure of hosts authentication. Figure 10 shows the terminal output.

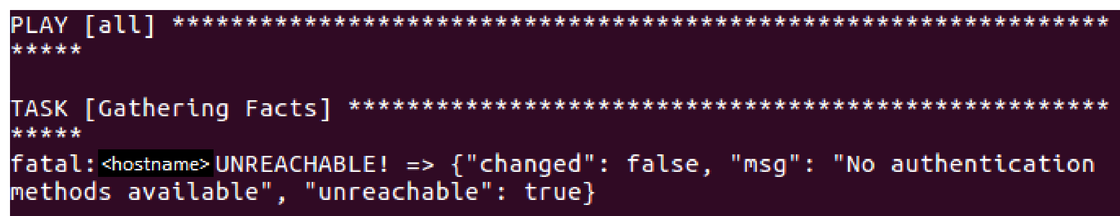

Figure 10. The Ansible container unable to authenticate the hosts

In spite of mounting the correct SSH keys, the container is still unable to use them. The reason is that these keys have certain user permissions on the underlying host. Namely, the files are owned by the user which was added previously to the Dockerfile. However, merely adding the user to the image is not enough for the container to be able to access the keys. One of the solutions is to run the container as the user who owns these files. In this case

it is implemented by adding the `--user` tag and specifying the user ID explicitly (Listing 10).

Listing 10. Running the container as a specific user

```
> docker run \
     -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
     -v /etc/ansible/hosts:/etc/ansible/hosts:ro
     -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
     -v <SSH-keys-at-host>:<SSH-keys-at-container>:ro \
     -v <logfile-path>:<logfile-path> \
     -i \
     --network=host \
     --user ${UID} \
     ${IMAGE_NAME}:${IMAGE_TAG}
```

This eliminates the problem, since the userspace is managed by the kernel. Once again, the kernel is shared. Therefore, according to Campbell (2017), the user IDs are similar on the host and its containers. As for the usernames, they are used as aliases and not reliable enough for the user identification (Campbell 2017). Thus, if the `--user` tag provides the correct `UID`, all the permissions of this user on the host become available to the container. The required `UID` can be fetched with the internal command `$(id -u <username>)`.

The previous improvement allows Ansible tasks to start executing from the container. Figure 11 shows the result.

```
> docker run -v /etc/logrotate.d/<filename>:/etc/logrotate.d/<filename>:ro -v /etc/ansible/hosts:
/etc/ansible/hosts:ro -v /etc/ansible/ configs-for-site /group_vars:/etc/ansible/ configs-
for-site /group_vars:ro -v / <SSH-directory> :/ <SSH-directory> :ro -v / <logfile-path> :/ <logfile-
path> -i --network=host --user $(id -u <user>) <image-name> : <image-tag>
/usr/lib/python2.7/getpass.py:83: GetPassWarning: Can not control echo on the terminal.
  passwd = fallback_getpass(prompt, stream)
Warning: Password input may be echoed.
SSH password:

Warning: Password input may be echoed.
SUDO password[defaults to SSH password]:


PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [ <hostname>

TASK [ <task-name> ****************************************
ok: [ <hostname>
```

Figure 11. Running the first Ansible playbooks in the container successfully

Despite the successful start, one of the Ansible tasks of the second *entrypoint*'s command (Listing 3, p. 29) interrupts the execution of the container. Therefore, the container is not able to start the main set of playbooks (`site.yml`). Figure 12 shows the failing task and its error message.

```
TASK [ <task-name> ***********************
 [WARNING]: Unable to find '/ <path-for-secrets> ' in expected paths.
fatal: <host> FAILED! => {"failed": true, "msg": "An unhandled exception occurred while run
ning the lookup plugin 'file'. Error was a <class 'ansible.errors.AnsibleError'>, original
 message: could not locate file in lookup: / <path-for-secrets> "}
```

Figure 12. The container failing to access the directory with secrets

The case is that the playbook tries to access one of the public keys at `<path-for-secrets>`. This directory is not available at the container at the moment. Therefore, it is decided to manage the secrets with the same method as the SSH keys previously, namely, via mounting them from the host at the runtime:

```
VOLUME ["/etc/logrotate.d/<ansible-logrotate-filename>",
"/etc/ansible/hosts", "/etc/ansible/configs-for-site/group_vars",
"<SSH-keys-at-container>", "<logfile-path>", "<path-for-secrets>"]
```

Also, the `docker run` command has to be modified accordingly (Listing 11).

Listing 11. Mounting the secrets directory

```
> docker run \
     -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
     -v /etc/ansible/hosts:/etc/ansible/hosts:ro
     -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
     -v <SSH-keys-at-host>:<SSH-keys-at-container>:ro \
     -v <logfile-path>:<logfile-path> \
     -v <path-for-secrets>:<path-for-secrets> \
     -i \
     --network=host \
     --user ${UID} \
     ${IMAGE_NAME}:${IMAGE_TAG}
```

However, the volume is not limited with the read-only access. Figure 13 shows the error if the read-only access is forced. The reason of this error is that one of the Ansible tasks has to write to the <path-for-secrets> directory.

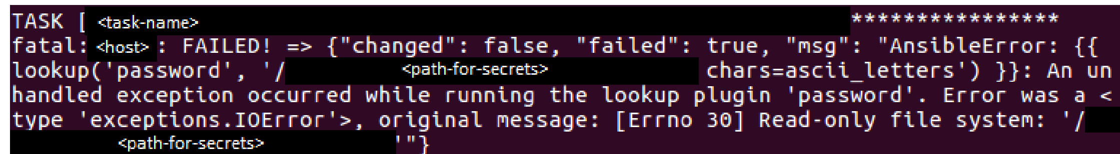

Figure 13. The read-only access preventing from writing to the volume

In spite of the fact that the <path-for-secrets> directory is mounted, the container still cannot manage to access it and keeps displaying the error from Figure 12 (p. 43). This happens, because this directory is owned by a certain group on the underlying host. Therefore, the container is not able to access the files inside it. The issue is solved by providing the required group ID (GID) to the docker run command (Listing 12).

Listing 12. Running the container as a member of a specific group

```
> docker run \
    -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
    -v /etc/ansible/hosts:/etc/ansible/hosts:ro
    -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
    -v <SSH-keys-at-host>:<SSH-keys-at-container>:ro \
    -v <logfile-path>:<logfile-path> \
    -v <path-for-secrets>:<path-for-secrets> \
    -i \
    --network=host \
    --user ${UID} \
    --group-add ${GID} \
    ${IMAGE_NAME}:${IMAGE_TAG}
```

As soon as the modification with `GID` is implemented, the container is able to access the required directory. Figure 14 shows the final results after introducing this change.

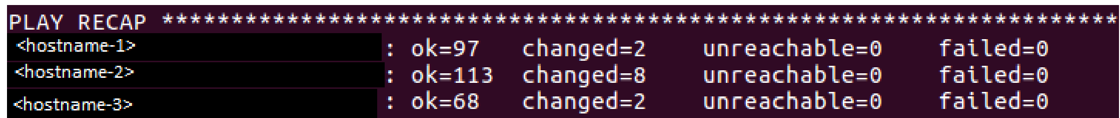

Figure 14. All the Ansible playbooks are executed successfully

As Figure 14 displays, there are no failed tasks and this Docker container is able to run the whole set of Ansible playbooks for this product. Therefore, the `docker run` command from Listing 12 is the final one.

Listing 13 shows the final version of the Dockerfile for the Ansible release image according to the identified errors.

Listing 13. The final version of the Dockerfile for the Ansible release image

```
FROM ansible-base-image:local
ARG user
ARG password
RUN <adding-the-user-to-the-system>
WORKDIR /home/$user
RUN mkdir -p /etc/ansible/configs-for-site/roles/<role-
name>/defaults
COPY main.yml /etc/ansible/configs-for-site/roles/<role-
name>/defaults/main.yml
COPY <entrypoint-filename> .
COPY ansible.cfg /etc/ansible
RUN rm /etc/ansible/hosts
COPY configs-for-site /etc/ansible/configs-for-site
VOLUME ["/etc/logrotate.d/<ansible-logrotate-filename>",
"/etc/ansible/hosts", "/etc/ansible/configs-for-
site/group_vars", "<SSH-keys-at-container>", "<logfile-path>",
"<path-for-secrets>"]
ENTRYPOINT ["./<entrypoint-filename>"]
```

All the changes in Listing 13 compared to Listing 2 (p. 28) are highlighted in bold.

4.3.4 Replacing the existing solution with the container

Despite managing to run all the playbooks from the container, Chapter 4.3.3 has not validated yet that this container is entirely functional. The reason is that the container performed all the tasks on the running site which has already been configured by the *existing solution* and its Ansible playbooks previously. Therefore, the container does not perform most of the tasks. In fact, Ansible can detect that these tasks have already been implemented so it skips them with *ok* status as Figure 14 (p. 45) shows.

This chapter shows the process of testing the container when the playbooks are not applied yet. In other words, this Docker container is run against the unconfigured site. Thus, it is validated if the container is indeed capable of orchestrating this site and replacing the *existing solution*.

One networking issue was detected when the container was run for the first time on the unconfigured site. Namely, as soon as the *entrypoint* of the container finishes its first ansible-playbook command (Listing 3, p. 29), it is not able to proceed with the next ones. The hosts become unreachable as Figure 15 shows.

```
PLAY RECAP *********************************************************************
cnf                        : ok=17   changed=10   unreachable=0    failed=0

/usr/lib/python2.7/getpass.py:83: GetPassWarning: Can not control echo on the te
rminal.
  passwd = fallback_getpass(prompt, stream)
Warning: Password input may be echoed.
SSH password:

Warning: Password input may be echoed.
SUDO password[defaults to SSH password]:


PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
fatal: <host> : UNREACHABLE! => {"changed": false, "msg": "[Errno -3] Temporary fai
lure in name resolution", "unreachable": true}
```

Figure 15. The failing second command in the container's *entrypoint*

The reason of this error is that the container's first *entrypoint* command (Listing 3, p. 29) changes the networking settings with Ansible on the virtual machine beneath this container. However, such network-related configuration files as */etc/hosts* and */etc/resolv.conf* are not synchronized between the running container and the underlying host. As a result, the Ansible container is not able to reach other virtual machines anymore. In fact, according to Docker Inc. (2017), Docker daemon does not rewrite network configuration files in running containers.

Referring to Docker Inc. (2017), it is recommended restarting containers to update them with the recent network changes. As for this case, there is no need to maintain the restarting behavior for the Ansible container. Instead, the first *entrypoint*'s command (ansible-playbook -t <network-tag>) can run in the separate instance of the container. As soon as it finishes its instructions, another container can be started to execute the rest of the commands in the *entrypoint*. Therefore, the container's start script should be changed accordingly as Listing 14 shows.

Listing 14. The *entrypoint* script modified

```
if [[ $1 == "<network-tag>" ]] ; then
     ansible-playbook -Kk -t <network-tag> \
       /etc/ansible/configs-for-site/site.yml
else
     ansible-playbook -Kk \
       /etc/ansible/configs-for-site/<ssh-playbook-name>.yml
     ansible-playbook /etc/ansible/configs-for-site/site.yml
fi
```

Now, the Ansible commands for the container can be chosen depending whether the `<network-tag>` tag is provided to the *entrypoint* or not. As mentioned before, the `ENTRYPOINT` statement in Dockerfiles allows providing a tag to a script directly via the `docker run` command. For instance, in order to run only the first `ansible-playbook` command from Listing 14, the following command from Listing 15 can be run.

Listing 15. Running the container to configure the networking only

```
> docker run \
     -v /etc/logrotate.d/<ansible-logrotate-
filename>:/etc/logrotate.d/<ansible-logrotate-filename>:ro \
     -v /etc/ansible/hosts:/etc/ansible/hosts:ro
     -v /etc/ansible/configs-for-
site/group_vars:/etc/ansible/configs-for-site/group_vars:ro \
     -v <SSH-keys-at-host>:<SSH-keys-at-container>:ro \
     -v <logfile-path>:<logfile-path> \
     -v <path-for-secrets>:<path-for-secrets> \
     -i \
     --network=host \
     --user ${UID} \
     --group-add ${GID} \
     ${IMAGE_NAME}:${IMAGE_TAG} \
     <network-tag>
```

Such a split into two container instances helps solving the problem with outdated network settings. Therefore, these instances are now able to run all three commands from the *entrypoint* (Listing 14) if run one after another.

Despite the resolved issue with the networking, there are a few Ansible tasks detected which are not able to be executed from the container. Altogether, there are around 200 tasks in *configs-for-site*. As for the failing tasks, there are only 13 of them. Moreover, these tasks are not critical for the functionality of the product. Therefore, these errors are omitted for solving inside the scope of this thesis. Figure 16 shows the results of running the container without these 13 Ansible tasks.

```
PLAY RECAP *********************************************************************
<hostname1>          : ok=116  changed=80   unreachable=0    failed=0
<hostname2>          : ok=150  changed=109  unreachable=0    failed=0
<hostname3>          : ok=94   changed=64   unreachable=0    failed=0
```

Figure 16. Running the Ansible container successfully on the unconfigured site

As Figure 16 shows, the container successfully manages to orchestrate the unconfigured site in such a case.

5 RESULTS

The following list summarizes the practical work done in this thesis:

1. The container was created with the *on-top-of-legacy* approach. Namely, this container was implemented to be run on the existing *Configurator VM* in this case. Moreover, the current stack remained untouched.
2. Overall, there are three Docker images in the hierarchy: the local Docker base image with Ubuntu, the image with the Ansible software and, lastly, the image with Ansible configured for this product. The first image was already implemented by the development team and the last two were created in the scope of this thesis.
3. The top level Docker image was implemented according to the definition of the *release image* from the Docker documentation. Namely, it includes the present Ansible playbooks to orchestrate the site (*configs-for-site*). Additionally, it includes the generic configurations applicable for all the environments. As for the runtime configurations, it was decided to mount them from the host to the containers with the volumes.
4. The build/deploy solution was implemented for the container as well. Additionally, it was automated. This was achieved by adding new contents to the existing automated scripts for the product’s builds and deployments.
5. The container was validated using the local environment.
6. The container manages to connect to other hosts by using the host Docker networking.
7. The required secrets were injected to the container with the volumes.
8. Several issues were identified and fixed when the container was run on top of the real product’s site. This test method was applied to both the running site as well as the empty site, which was not configured with

Ansible playbooks previously. The problems were generally related to the missing configurations, file permissions and network updates.

Figure 17 shows the overall container's structure.

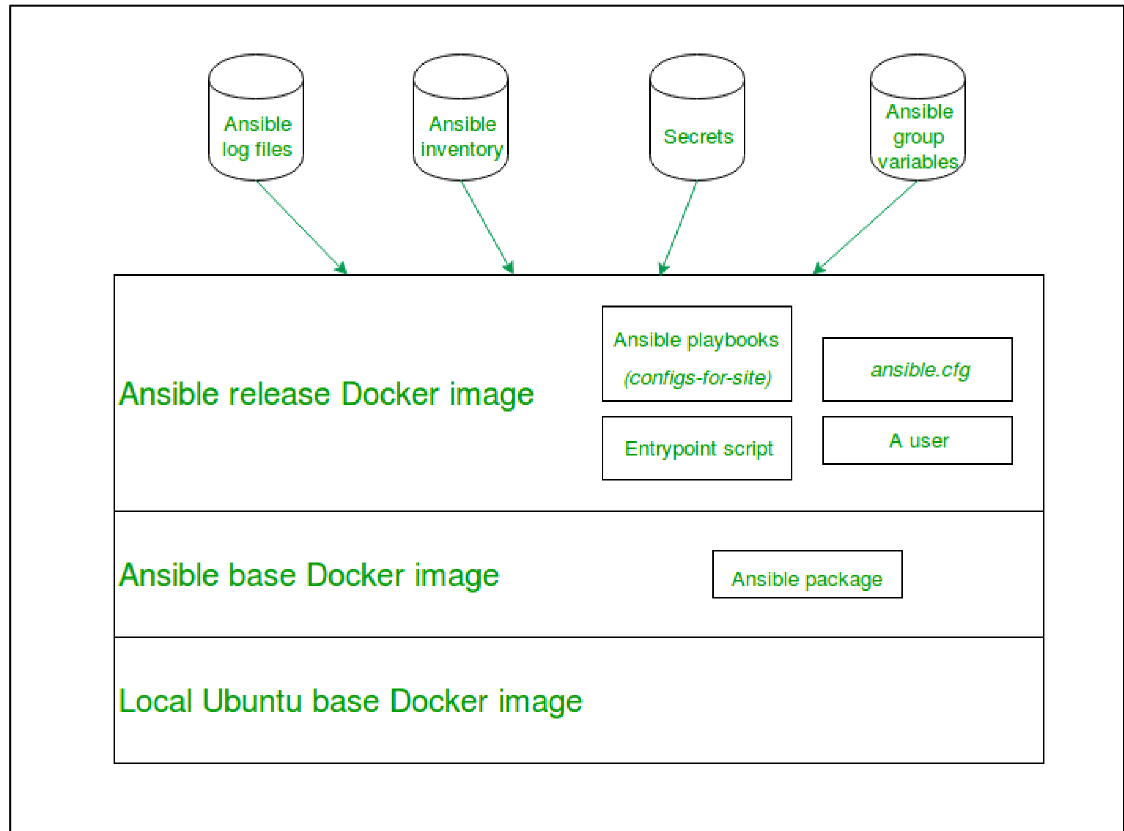

Figure 17. The structure of the final Docker container solution for Ansible

All in all, such a container manages to orchestrate the site with the Ansible playbooks it contains. Despite the current state of the hosts, the container is capable of enforcing the state defined in its playbooks. The only exception identified is the first Ansible run on an unconfigured site. In such a case, 13 out of around 200 Ansible tasks fail. However, these instructions are not critical for the product's performance. Therefore, they are omitted from the scope of this thesis.

The functionality of the container was additionally validated with the existing product's test suite. Namely, the container was deployed to the empty unconfigured site. Then, the full set of Ansible playbooks was run with the exception of the 13 tasks mentioned before. After that, the test suite was executed. As Figure 18 shows, all the test cases pass in such a case.

```
================================================================================
Robot :: Test suite initialization                                    | PASS |
7 critical tests, 7 passed, 0 failed
7 tests total, 7 passed, 0 failed
================================================================================
```

Figure 18. Passing the test cases after configuring the site with the Ansible container

The update time for Ansible and its playbooks has decreased significantly with the container solution. This time is the interval from the beginning of the build to the point when the playbooks are ready to be run on the deployed site. The measurements were performed in the local development environment. All the cache and build output were cleaned before the builds. Five samples were taken per the *existing solution* and per the *proposed solution*. The former one requires the whole base VM image's rebuild and redeployment to get updated. As for the latter, it only needs to rebuild the Docker image and upload it to the running site. The results for the *existing* and *proposed solutions* are shown in Figure 19. The measurements are also sorted.

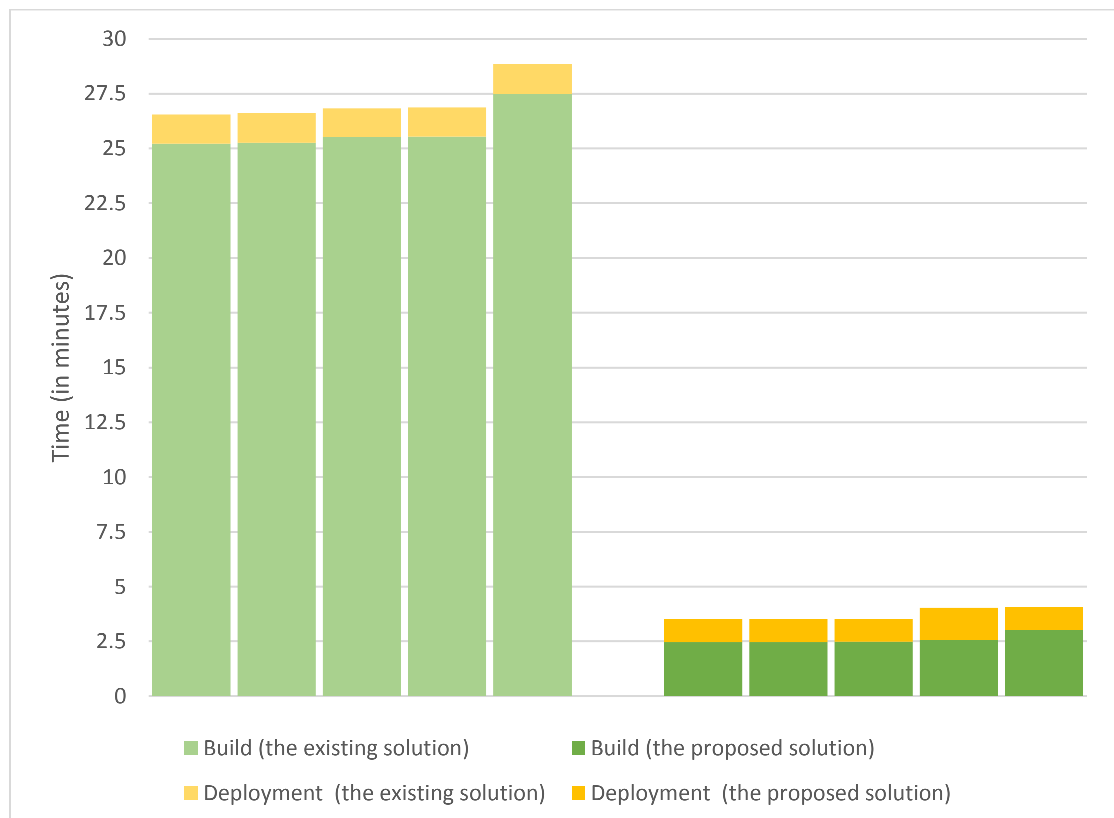

Figure 19. The update time measurement (sorted) for the existing and proposed solutions

It is estimated for the *existing solution* to take 27.14 minutes on average in order to update the Ansible playbooks or Ansible version. As for the container,

the update procedure takes 3.74 minutes on average. All in all, the containerization reduces the update time by 86.22%.

The *proposed solution* has also changed the size of the update package for Ansible and its playbooks. Currently, the base VM image has the size of 3.74 GB. As for the Ansible release image, it takes 451 MB of the disk space. Therefore, there is a reduction of 88.22%. As for the archived and compressed image, it is 171 MB and, as a result, there is a 95.53% reduction. Figure 20 shows the graph with all the three units.

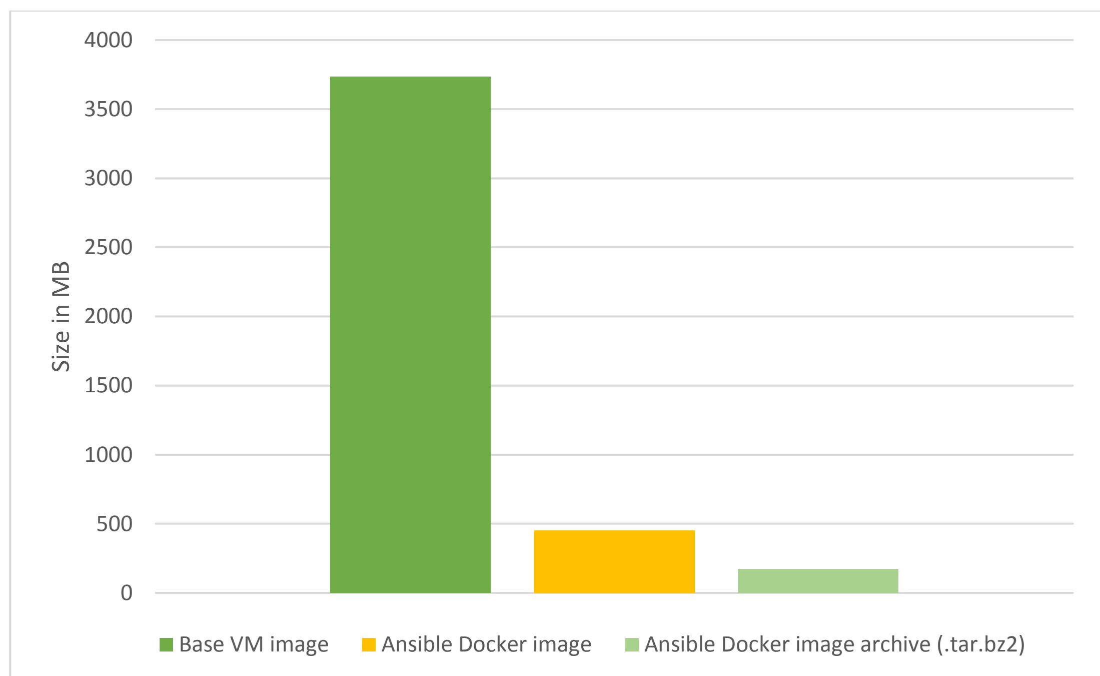

Figure 20. The comparison of disk consumption

The performance of Ansible playbooks was also compared between the container and the VM. Figure 21 shows the results of the measurements.

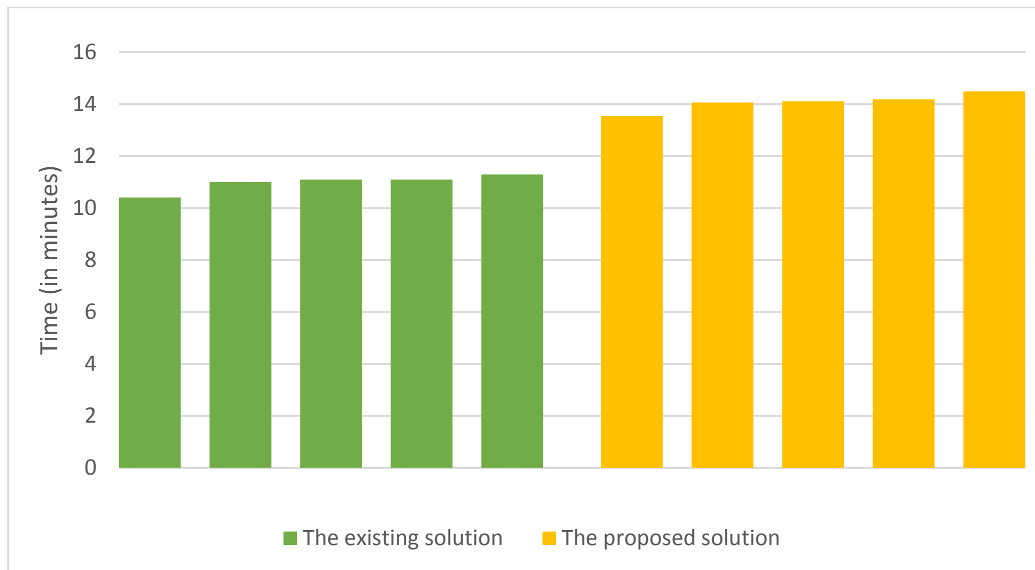

Figure 21. The comparison of performance time for Ansible (sorted)

The *existing solution* manages to finish the whole set of playbooks in 10.9 minutes on average (measured 5 times). As for the Docker container, 5 measurements show that it takes on average 14 minutes to orchestrate the site. Thus, the *existing solution* completes the playbooks 22.14% faster than the container solution.

6 CONCLUSION

The original problems of this thesis were the low speed and inconvenience of updates for Ansible and its playbooks in the product. Therefore, the aim was to make these updates faster and separate them from the general software's upgrade procedure. The proposed solution was to containerize the existing Ansible software with Docker. In the given timeframe, such a container was implemented and tested. As a result, it successfully managed to solve two original problems stated above. Additionally, it is able to orchestrate the whole site similarly to the *existing solution.*

In addition to achieving two original improvements, there are other advantages reached with the container solution. First of all, the container is straightforward to build and run, especially with the implemented automation scripts. This advantage together with decreased update time improves the processes of testing and roll-back inside the development team. Lastly, the update package

takes significantly less disk space in case of the Docker image than the base VM image.

However, there are several future studies which could be conducted for this solution outside the timeframe of this thesis. The list is the following:

- Particular tasks should be fixed in this set of Ansible playbooks. Currently, 13 out of around 200 Ansible tasks fail inside the container. They are not crucial for the product's functionality, but still require attention for the sake of the complete performance of the container.
- The *proposed solution* could be validated at the remaining untested product's environments. Thus, the container would be verified in most of the possible cases: from a local environment to clients' premises.
- The input of SSH and sudo passwords should be improved. Currently, they might be echoed to the terminal. Therefore, the passwords are not secured well enough from possible security attacks.
- Version control should be improved for this container. Currently, the same tag is assigned to the container at every build. Thus, the tag does not explicitly express which version of the Docker image is present in the Docker daemon at the moment. Moreover, the compatibility between the container's version and the version of the underlying infrastructure should be optimized.
- File permissions of the mounted configuration files could be adapted to the container differently. Currently, the Ansible container is adjusted to the required permissions by merely injecting user and group IDs to it at the runtime. However, this solution creates a few limitations and makes the container more dependable on the underlying host. Moreover, some of the file permissions might not be preferable to be granted to this container from the security point of view.
- More comprehensive secrets management system could be needed. Currently, the keys are injected to the container via volumes. However, it makes the solution less portable since the secrets might not always be available at the specified location. Moreover, it introduces security vulnerabilities, since the container has to be granted higher permissions to be able to access these keys.
- Another networking method could be applied to the case. Currently, host networking is utilized. It limits the scalability and involves several security implications, since the container is directly exposed to the network interface. Also, it creates additional dependency of the container on the underlying host. Moreover, as detected during the implementation process, the running containers do not synchronize with the host's network updates. Therefore, if one of the Ansible tasks perform network changes on the node, the running container can lose connectivity to other virtual machines.
- The Ansible container runtime could be improved. As the results show, this container executes all the playbooks slower than the *existing solution*. Therefore, the factors decreasing the performance should be detected.
- Docker registry is suggested for the usage in this product case. Thus, the images are easier to maintain and distribute between different environments.

In general, the whole containerization process could have been done differently from the approach point of view. Namely, instead of using the *Configurator VM* and the *on-top-of-legacy* method, a VM with no pre-configurations could have been used. Thus, the resulting Docker image would be more portable. The reason for the improved portability is that the container would be adapted to the environment where nothing but the Docker daemon is present. Therefore, as soon as implemented, it can be run in various hosts and sites having Docker installed, since it does not have any prerequisites. Moreover, if a highly portable container is created, the *existing solution* for Ansible can be safely discarded in the future. All in all, such an approach is recommended to take as a first step for containerization in order to obtain a maximum possible portability. However, it takes considerably more effort and research compared to the approach taken in this work. As for this case, an empty VM can be utilized as a next step for the improvement of the container's portability.

The development of the container including the learning and research took approximately 250 man-hours in this thesis. The results gained during this restricted timeframe provide a proof-of-concept rather than a production-grade solution. In order to make the container suitable for production, the Docker images still require a number of improvements listed above. The implementation of these advancements could take approximately similar effort. From the product management point of view, the resources might be restricted taking into account other tasks, customer requirements and maintenance operations. Therefore, the development team should decide whether the advantages of the quick and independent updates for the Ansible playbooks are worth the estimated effort in the context of this project.

All in all, in theory Docker allows containerization of any kind of software, and configuration management systems are not an exception. However, in practice, the success of containerization depends heavily on the properties of an application as well as how it is packed and run in a container. Also, additional challenges emerge, if an existing product with a mature architecture is planned to be migrated into a container platform. In such a case, the proper containerization approach and strategy should be chosen. Moreover, it should

be evaluated if the effort put into such a migration worth the advantages which could be gained by the implementation of Docker containers.

REFERENCES

About images, containers, and storage drivers. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers [Accessed 8 September 2017].

Ansible. 2017. Company's page. WWW document. Available at: https://www.ansible.com [Accessed 22 September 2017].

Ansible Docker base. 2014. GitHub. WWW document. Updated 4 September 2015. Available at: https://github.com/ansible/ansible-docker-base [Accessed 19 October 2017].

An overview of Chef. 2017. Chef Docs. WWW document. Available at: https://docs.chef.io/chef_overview.html [Accessed 12 October 2017].

Basic networking with Docker. 2017. Slash Docker. WWW document. Available at: https://runnable.com/docker/basic-docker-networking [Accessed 12 October 2017].

Babcock C. 2015. Containers explained: 9 essentials you need to know. WWW document. Updated 2 October 2016. Available at: https://www.informationweek.com/strategic-cio/it-strategy/containers-explained-9-essentials-you-need-to-know/a/d-id/1318961 [Accessed 15 September 2017].

Benevides R. 2016. 10 things to avoid in docker containers. WWW document. Updated 24 February 2016. Available at: https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers [Accessed 15 August 2017].

Bernardes M. 2016. 15 things you should know about Ansible. WWW document. Updated 12 January 2016. Available at: http://codeheaven.io/15-things-you-should-know-about-ansible/ [Accessed 12 October 2017].

Best practices for writing Dockerfiles. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices [Accessed 10 October 2017].

Campbell M. 2017. Understanding how uid and gid work in Docker containers. WWW document. Updated 30 January 2017. Available at: https://medium.com/@mccode/understanding-how-uid-and-gid-work-in-docker-containers-c37a01d01cf [Accessed 25 October 2017].

Case study: Microsoft IT modernizes traditional apps with Docker EE and Azure. 2017. Docker Inc. Video clip. Available at: https://www.youtube.com/watch?v=pHgHUVXpGIg [Accessed 31 June 2017].

Coleman M. 2016. Containers are not VMs. WWW document. Updated 24 March 2016. Available at: https://blog.docker.com/2016/03/containers-are-not-vms/ [Accessed 6 September 2017].

Configuration file. 2016. Red Hat. WWW document. Updated 26 September 2017. Available at: http://docs.ansible.com/ansible/latest/intro_configuration.html [Accessed 12 October 2017].

Configure container DNS. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/userguide/networking/default_network/configure-dns [Accessed 26 October 2017].

Continuous integration. 2017. ThoughtWorks. WWW document. Available at: https://www.thoughtworks.com/continuous-integration [Accessed 20 October 2017].

Create a base image. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/userguide/eng-image/baseimages [Accessed 6 October 2017].

Daniel, H. 2013. Ansible Configuration Management. Birmingham: Packt Publishing.

DeHamer B. 2015. Dockerfile: ENTRYPOINT vs CMD. WWW document. Updated 16 July 2015. Available at: https://www.ctl.io/developers/blog/post/dockerfile-entrypoint-vs-cmd/ [Accessed 18 October 2017].

Docker container networking. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/userguide/networking/ [Accessed 12 October 2017].

Docker Hub. 2017. Main page. WWW document. Available at: https://hub.docker.com/ [Accessed 10 October 2017].

Docker reference architecture: development pipeline best practices using Docker EE. 2017. Docker Inc. WWW document. Updated 21 September 2017. Available at: https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Development_Pipeline_Best_Practices_Using_Docker_EE [Accessed 31 June 2017].

Docker registry. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/registry/ [Accessed 10 October 2017].

Docker security. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/security/security [Accessed 10 October 2017].

Dockerfile reference. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/reference/builder [Accessed 17 October 2017].

Fowler M. 2016. InfrastructureAsCode. WWW document. Updated 1 March 2016. Available at: https://martinfowler.com/bliki/InfrastructureAsCode.html [Accessed 6 October 2017].

Gallagher, S. 2016. What you need to know about Docker. Birmingham: Packt Publishing.

Get started, part 1: orientation and setup. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/get-started [Accessed 5 September 2017].

Getting started. 2016. Red Hat. WWW document. Updated 21 September 2017. Available at: http://docs.ansible.com/ansible/latest/intro_getting_started.html [Accessed 24 October 2017].

Gienow M. 2017. Docker basics: diving into the essential concepts, tools, and terminology. WWW document. Updated 4 August 2017. Available at: https://thenewstack.io/docker-station-part-one-essential-docker-concepts-tools-terminology/ [Accessed 6 September 2017].

Hausenblas M. 2016. What is Docker networking? WWW document. Updated 11 April 2016. Available at: https://www.oreilly.com/learning/what-is-docker-networking [Accessed 12 October 2017].

Heidi E. 2016. An introduction to configuration management. WWW document. Updated 24 March 2016. Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-configuration-management [Accessed 6 October 2017].

Higginbotham J. 2016. How to get code into a Docker container. WWW document. Available at: http://blog.cloud66.com/how-to-get-code-into-a-docker-container [Accessed 10 October 2017].

How Ansible works. 2017. Red Hat. WWW document. Available at: https://www.ansible.com/how-ansible-works [Accessed 9 October 2017].

How Chef works. 2014. Chef Software. Video clip. Available at: https://www.youtube.com/watch?v=_pUdL4w9E7k [Accessed 12 October 2017].

Infrastructure as code. 2017. Puppet. WWW document. Available at: https://puppet.com/solutions/infrastructure-as-code [Accessed 22 September 2017].

Introduction to containers: concept, pros and cons, orchestration, Docker, and other alternatives. 2016. Medium. WWW document. Updated 30 September 2016. Available at: https://medium.com/flow-ci/introduction-to-containers-concept-pros-and-cons-orchestration-docker-and-other-alternatives-9a2f1b61132c [Accessed 12 September 2017].

Inventory. 2016. Red Hat. WWW document. Updated 19 September 2017. Available at: http://docs.ansible.com/ansible/latest/intro_inventory.html [Accessed 9 October 2017].

Khan Z. 2017. From monolith to containers: How Verizon containerized legacy applications on OpenShift. Video clip. Available at:

https://www.youtube.com/watch?v=Q6i0LK4vHsU [Accessed 14 August 2017].

Make – Linux man page. 2017. Free software foundation. WWW document. Available at: https://linux.die.net/man/1/make [Accessed 20 October 2017].

Manage sensitive data with Docker secrets. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/swarm/secrets [Accesssed 24 October 2017].

Moving legacy applications to Docker - presentation from DockerCon Seattle. 2016. Apcera. WWW document. Updated 19 July 2016. Available at: https://www.apcera.com/blog/moving-legacy-applications-docker-presentation-dockercon-seattle [Accessed 6 October 2017].

Overview of Docker Hub. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/docker-hub/ [Accessed 10 October 2017].

Pillai S. 2012. Puppet tutorial: how does Puppet work. WWW document. Updated 12 July 2012. Available at: http://www.slashroot.in/puppet-tutorial-how-does-puppet-work [Accessed 12 October 2017].

Portnoy, M. 2012. Virtualization essentials. 1st edition. Hoboken: John Wiley & Sons, Incorporated.

Posta C. 2017. Low-risk monolith to microservice evolution part I. WWW document. Updated 19 September 2017. Available at: http://blog.christianposta.com/microservices/low-risk-monolith-to-microservice-evolution [Accessed 17 October 2017].

Setting up logrotate on RedHat Linux. 2014. LinuxConfig. WWW document. Updated 22 July 2014. Available at: https://linuxconfig.org/setting-up-logrotate-on-redhat-linux [Accessed 20 October 2017].

Seven configuration management (CM) tools you need to know about. 2017. UpGuard. WWW document. Updated 18 July 2017. Available at: https://www.upguard.com/articles/the-7-configuration-management-tools-you-need-to-know [Accessed 28 September 2017].

Shaw G. 2017. Perform an unattended installation of a Debian package. WWW document. Available at: http://www.microhowto.info/howto/perform_an_unattended_installation_of_a_debian_package.html [Accessed 19 October 2017].

Tehranian D. 2015. How should I get application configuration into my Docker containers? WWW document. Updated 25 March 2015. Available at: https://dantehranian.wordpress.com/2015/03/25/how-should-i-get-application-configuration-into-my-docker-containers/ [Accessed 4 September 2017].

Templating (Jinja2). 2017. Red Hat. WWW document. Updated 7 June 2017. Available at: http://docs.ansible.com/ansible/latest/playbooks_templating.html [Accessed 20 October 2017].

The history of containers. 2015. Red Hat enterprise Linux blog. WWW document. Updated 28 August 2015. Available at: http://rhelblog.redhat.com/2015/08/28/the-history-of-containers [Accessed 21 August 2017].

Use volumes. 2017. Docker Inc. WWW document. Available at: https://docs.docker.com/engine/admin/volumes/volumes [Accessed 25 October 2017].

Using SSH keys inside Docker container. 2013. Stack overflow. WWW document. Updated 3 October 2017. Available at: https://stackoverflow.com/questions/18136389/using-ssh-keys-inside-docker-container [Accessed 24 October 2017].

Venezia P. 2013. Review: Puppet vs. Chef vs. Ansible vs. Salt. WWW document. Updated 21 November 2013. Available at: https://www.infoworld.com/article/2609482/data-center/data-center-review-puppet-vs-chef-vs-ansible-vs-salt.html [Accessed 6 October 2017].

Wang C. 2017. What is Docker? Linux containers explained. WWW document. Updated 29 June 2017. Available at: https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html [Accessed 22 September 2017].

What is a container. 2017. Docker Inc. WWW document. Available at: https://www.docker.com/what-container [Accessed 5 September 2017].

Wootton B. 2017. Who's using Docker? WWW document. Updated 31 January 2017. Available at: https://www.contino.io/insights/whos-using-docker [Accessed 15 September 2017].

Zanzane D. 2016. How Ansible works. WWW document. Updated 11 February 2016. Available at: https://www.linkedin.com/pulse/how-ansible-works-dadaso-zanzane/ [Accessed 9 October 2017].