A STUDY OF THE USE OF DOCKER COMPOSE AND DOCKERHUB IMAGES

by

Md Hasan Ibrahim

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
September 2019

Copyright © Md Hasan Ibrahim, 2019
Abstract

Docker is a tool used to encapsulate a software package with all of its dependencies and configurations into an isolated environment. Nowadays, developers compose multiple Docker images to form multi-component applications.

While prior studies of Docker examine the quality and evolution of a single Docker image, no study has explored the use of Docker in a multi-component setting. In this thesis, we first identify how multi-component applications are composed and maintained using Docker Compose, then we examine Docker images that are hosted on DockerHub, a commonly used online registry for Docker images.

From our first study of 4,103 open-source Github projects that use Docker Compose, we observe that 26.8% of the projects needlessly use Docker Compose to compose a single component application, multi-component applications stick to the basic
options of Docker Compose and ignore advanced ones such as security and monitoring related options, and multi-component applications rarely upgrade their Docker Compose version. We also observe that 77% of the studied applications use images from an online registry and 95.2% of these registry images are hosted on DockerHub.

Hence, we investigate the available Docker images on DockerHub. We studied 505 DockerHub images for five popular software systems (101 images for each system). We observed that DockerHub community images differ from their official image, while also differing from each other in terms of their installed libraries. We also observe that there exist community images that are more resource-efficient and which contain fewer security vulnerabilities compared to their official image. However, users might not find such images since they are not well-documented.

Our thesis suggests the need for tools and methodologies to help multi-component applications take full advantage of the capabilities and resources that are offered by Docker Compose and DockerHub.
At first, I would like to acknowledge my heartiest gratitude to almighty Allah for his endless blessings. Then I would like to render my deepest thanks to my supervisor Prof. Ahmed E. Hassan who gave me the opportunity to work under his supervision. His guidance and support made this work a success.

I am incredibly grateful to my mentor Dr. Mohammed Sayagh who always inspired me and gave invaluable advice. This work would be more difficult without his assistance. I came to know many useful things from him.

I extend my thanks to my fellow labmates Md Ahsanuzzaman, Abdullah Ahmad Zarir, Daniel Lee, Aaditya Bhatia, and Kundi Yao for their continuous support.

I want to give special thanks to my wife (Rubaiyat Jahan Mumu) for her continuous support and understanding. My family, specially my parents are my biggest inspiration to accomplish this work as they sacrificed a lot for me to be here. I want to dedicate this thesis to them.
# Table of Contents

Abstract .................................................. i

Acknowledgments ......................................... iii

List of Tables ............................................. vi

List of Figures ........................................... vii

1 Introduction ............................................ 1
  1.1 Thesis Statement .................................. 3
  1.2 Thesis Overview .................................. 4
  1.3 Thesis Contributions ............................. 6

2 Background and Related Work ......................... 7
  2.1 Background ........................................ 8
  2.2 Related Work ...................................... 13

3 A Study of How Docker Compose is Used to Compose Multi-component Applications ............ 18
  3.1 Introduction ....................................... 19
  3.2 Data Collection .................................... 22
  3.3 Results ............................................. 23
  3.4 Threats to Validity ................................ 52
  3.5 Chapter Summary ................................... 53

4 Too many Images on DockerHub! How Different are Images for the same System? ............ 54
  4.1 Introduction ....................................... 55
  4.2 Data Extraction .................................... 59
  4.3 Preliminary Study: The Advantages and Abundance of DockerHub Images ................ 61
  4.4 The Differences among DockerHub Images ........... 66
  4.5 Discussion .......................................... 79
## List of Tables

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Categories of options in Docker Compose across all versions.</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Options in Docker Compose that are never used. Note that the classification of these options is based on our own analysis.</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Options that were never changed in any revision of the studied Docker Compose files.</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Number of images available on DockerHub for the five studied software systems.</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>Library similarities of same name image pairs.</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>Number of images that have the same name.</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Paired Wilcoxon signed-rank test results between image size, and library count and wasted space.</td>
<td>73</td>
</tr>
<tr>
<td>4.5</td>
<td>Comparison of the resource efficiency between official and community images. Note that more than half of the Cassandra as well as Mongo images are more resource efficient than their corresponding official image (shown in bold).</td>
<td>74</td>
</tr>
<tr>
<td>4.6</td>
<td>The percentage of community images that are more resource efficient and have less security vulnerabilities compared to their corresponding official image.</td>
<td>80</td>
</tr>
<tr>
<td>4.7</td>
<td>Spearman's rank correlations between different metrics of the images.</td>
<td>82</td>
</tr>
<tr>
<td>4.8</td>
<td>Spearman's rank correlation coefficients between the resource efficiency, number of security vulnerabilities, and the popularity metrics.</td>
<td>84</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Building and sharing images on DockerHub. ............................................. 8
2.2 An example of a Dockerfile that defines an image for myapp.jar that runs on the Java runtime. .......................................................... 9
2.3 Official and community images on DockerHub.......................................... 10
2.4 DockerHub image meta-data. ................................................................. 11
2.5 Components and options in a Docker Compose file. ................................. 13

3.1 Number of commits in the collected projects. ............................................ 22
3.2 The number of projects for a given number of components. ....................... 24
3.3 The top 10 most used images by the percentage of projects. ..................... 27
3.4 Top 10 most popular image combinations used by the percentage of the multi-component applications that use more than one image from an online registry. ..................................................... 28
3.5 Usage of different option categories in Docker Compose files. ................. 30
3.6 The top 10 most used options across the studied projects. ....................... 32
3.7 Confidence metric values of co-occurrence of the top 10 most used options appearing in the same component. ...................................... 33
3.8 The frequency of change for Docker Compose files. The y-axis represents the cumulative percentage of Docker Compose files. For example, in Figure 3.8a, 18.4% of the Docker Compose files have at least 10 commits. ..................................................... 37
3.9 Number of changed lines across the proportion of revisions. The y-axis represents the cumulative proportion of revisions, while the x-axis represents the number of affected lines of code. For example, 50% of the revisions have changed at least five lines of code. .......................... 38
3.10 Correlation between the number of lines of code and the number changed lines of code. ................................................................. 39
3.11 The frequency of the changes of different option categories based on the analysis of 2,316 projects that revised at least one Docker Compose option. Note that in Figure 3.11b, the relevant projects are the projects that use a specific option category. ........................................ 40
3.12 Top 10 most frequently revised options in Docker Compose files based on the analysis of 2,316 projects that revised at least one Docker Compose option. Note that in Figure 3.12b, the relevant projects are the projects that use a specific option.

3.13 Types of modification done on images.

3.13 Examples of different types of changes to the volumes option.

3.14 Top five types of files that co-evolve the most with the Docker Compose file.

3.15 Exact versions of Docker Compose used by the percentage of projects.

3.16 An example of a Docker Compose version downgrade. This change downgraded the Docker Compose version from version 3 to version 2.

4.1 Overview of the data extraction approach used to answer our RQs. Each studied image installs the software system that is mentioned by its name using Advanced Package Tool (APT), Alpine Linux package manager (APK), or Red Hat Package Manager (RPM).

4.2 The number of installed libraries in different types of images.

4.3 The evolution of the number of images on DockerHub.

4.4 Optional caption for list of figures 5–8.

4.5 Optional caption for list of figures 5–8.

4.6 Optional caption for list of figures 5–8.

4.7 Size of different types of images, where the red star shows the size of the corresponding official image.

4.8 Resource efficiency distributions of the DockerHub images where the red star shows the resource efficiency of the corresponding official image.

4.9 Optional caption for list of figures 5–8.

4.10 The relation between the resource efficiency and security vulnerabilities of the images. Note that the green images are better than the official images in terms of resource efficiency and number of vulnerabilities, while the red images are the worst images on both perspectives. Blue and yellow images are better than the official image on just one of the two perspectives. We do not observe any Mongo image that is better than its corresponding official image in both perspectives.
CHAPTER 1

Introduction

Docker is one of the most popular containerization tools. It enables the encapsulation of software packages into containers. Docker containers speed up the deployment of packages without dealing with platform or third party libraries compatibility issues.

Due to its lightweight, portability, and self-sufficiency, Docker has become one of the most popular containerization tools. In 2016, container technologies generated a revenue of $762 million which is expected to reach $2.7 billion by 2020 (Serverwatch (2019)) where Docker accounts for 83% of all these containers (Carter (2019)). While Datadog (Datadog (2019a)) which is an emerging cloud service monitoring startup, reports that a quarter of their customer already adopted Docker to deploy their applications in the cloud (Datadog (2019b)).
One can compose their application using multiple components. Each component is represented by a Docker image and a set of options that customize the behaviour of that component, such as which data volumes to use, how to access other components, and how to behave when a component fails. Web applications are a typical examples of such multi-component applications since web applications consist of a set of layers (e.g., LAMP or MEAN stacks), where each layer can be deployed as a separate component.

Figure 1.1: An overview of building multi-component application with Docker and Docker Compose.

Technically, one can specify such complex applications using Docker Compose which composes multiple components where each component is specified using a Docker image. Docker Compose specifies the components of a multi-component application,
how such components should communicate, as well as additional options, as illustrated by the three components of Figure 1.1b. Figure 1.1a shows the process of containerizing a package where a developer writes a Dockerfile containing the specification of the target component from which a Docker image is built. When instantiating a multi-component application, Docker Compose instantiate the images of each component using Docker container.

Prior studies focus on single Docker images without considering the context in which an image is instantiated: the relation between that image and other images of a multi-component application. For example, Shu et al. (2017), Tak et al. (2018), and Zerouali et al. (2019) studied the security vulnerabilities of Docker images, while Cito et al. (2017), Zhang et al. (2018), and Zhang et al. (2019) studied the evolution and the quality issues of Docker image. However, no prior study has examined the creation and maintenance of multi-component applications using Docker Compose and Docker images.

1.1 Thesis Statement

In this thesis, we investigate on the composition of multi-component applications, as well as the differences among images on the DockerHub image registry. In summary, our thesis addresses the following thesis statement:

A deep empirically sound understanding of how multi-component applications are created in the wild is essential for shaping future research efforts and tooling support given the rapid growth and dominance of containerization as a crucial mechanism nowadays for creating complex software applications.
To examine this statement, we first conducted an empirical study on the composition of multi-component applications that use Docker Compose (a popular mechanism to create such applications using Docker images). Then, we empirically studied the differences among existing images on DockerHub, a commonly used online registry for Docker images.

1.2 Thesis Overview

1.2.1 Chapter 2: Background and Related Work

In this chapter, we provide the background information regarding Docker, DockerHub, and Docker Compose. We also discuss the prior research efforts which are related to our work.

1.2.2 Chapter 3: A Study of How Docker Compose is Used to Compose Multi-component Applications

Nowadays, practitioners commonly use Docker Compose to build multi-component application. To better understand how multi-component applications are composed, we conducted an empirical study on a large dataset of 4,103 open-source Github projects that use Docker Compose.

Our study revealed that multi-component applications do not fully benefit from the features and options of Docker Compose. In fact, we observed that our studied projects
use Docker Compose, but do not compose their applications from multiple components. We also observed that these projects leverage the basic options of Docker Compose, while ignoring more advanced options such as security-related options. Finally, we observe that multi-component applications rarely (8.5% of the projects) upgrade their Docker Compose version; In contrast, 2.4% of the projects downgraded their versions due to compatibility issues, either with the host (e.g., AWS) or due to needing options that were removed from a new version of Docker Compose.

1.2.3 Chapter 4: Too many Images on DockerHub! How Different are Images for the same System?

We observed in our first study that more than 95% of the used remote images are hosted on DockerHub. DockerHub contains a large number of images for the same software system, which makes the selection of an appropriate image a non-trivial task. For example, to build the LAMP (Linux, Apache, MySQL, and PHP) stack, DockerHub contains 22,036, 18,869, and 40,773 images just for Apache, MySQL, and PHP respectively. The differences among these images is not clear.

Therefore, we conducted a second empirical study on 505 DockerHub images for five software systems (101 image for each system) to better understand the differences among these images in terms of their installed libraries, their resource efficiency (how much duplicate resources exist on an image), and their security.

We observed that the official images are the most popular images on DockerHub, even though other choices might exist (i.e., community images). Such images might be more secure and more resource efficient compared to the official images. Indeed, we observed that DockerHub images are different from each other in terms of installed
libraries; a median of just 4% of the images have another identical image on Docker-Hub. Furthermore, DockerHub images are different from each other in terms of their resource efficiency and a median of 39% of the community images are more efficient than their respective official image. Finally, we observed that images are different from each other in terms of their security and a median of 83% of the community images have less security vulnerabilities compared to their respective official image.

1.3 Thesis Contributions

In this thesis, we study the composition of multi-component applications with Docker Compose as well as the differences among images hosted on DockerHub. Our thesis has the following contributions:

1. We are the first to report on how multi-component applications are being built using Docker Compose in the wild. We observed that multi-component applications do not fully benefit from Docker Compose options and versions.

2. We are the first to note the challenges of finding appropriate images on Docker-Hub.

Our thesis highlights the need for future efforts to take advantages of Docker Compose by using advanced options (if they are needed) as well as proposing approaches that distinguish between DockerHub images and help practitioners find the most appropriate image given the needs.
Background and Related Work

CONTAINERIZATION is the process of packaging a software system with all of its dependencies so that it can be run on any platform without any compatibility or dependency issues. Prior studies focus on the quality and evolution of Docker images, while no study explored how multi-component applications are composed using Docker Compose as well as the differences among the used images from DockerHub. This chapter provides a background about Docker, DockerHub, and Docker Compose, then discusses related work.
2.1 Background

According to Acharya et al. (2018), Li et al. (2017) and Shirinbab et al. (2017), containerization is an approach that provides an operating system level virtualization which is more efficient and faster than the traditional virtualization platforms. That efficiency makes Docker one of the most popular containerization platforms (Merkel (2014)).

2.1.1 Docker Image

The creation of a Docker image starts with the creation of a Dockerfile as shown in Figure 2.1. A Dockerfile is a script that contains the required instructions to install and configure a software system. Figure 2.2 shows an example of a Dockerfile that builds an image to run the application `myapp.jar`, which depends on Java. That Dockerfile uses Ubuntu as its operating system (line 2), updates the installed packages (line 8), installs Java (line 11-12), copies resources into the Docker image (line 15), and runs the application `myapp.jar` (line 21).
Starting from a Dockerfile, a Docker image can be built manually or automatically as shown in Figure 2.1. In the manual process, a developer can build an image from a Dockerfile using the Docker Engine. This manual process requires a developer to build the image whenever he or she updates their Dockerfile. On the other hand, to build an image automatically, a developer needs to push their Dockerfile to a Github (Github (2019)) or Bitbucket (Bitbucket (2019)) repository and link that repository with a DockerHub repository (discussed in the next subsection). DockerHub then automatically builds and updates the Docker image when a new Dockerfile version is pushed to the version control repository.

```bash
# extending base image
FROM ubuntu:18.04

# maintainer email address
LABEL maintainer="john.doe@example.com"

# updating libraries
RUN apt-get -y update

# installing java8
RUN add-apt-repository ppa:webupd8team/java && 
    apt install -y oracle-java8-installer

# copying target application package
COPY . /app/

# setting the working directory
WORKDIR /app/

# running the application
CMD ["java", "-jar", "myapp.jar"]
```

Figure 2.2: An example of a Dockerfile that defines an image for `myapp.jar` that runs on the Java runtime.
2.1.2 DockerHub

DockerHub is an online image registry which is used to share Docker images. The Docker community can in turn pull and consume these public images. One can also share their images privately with a selective group of users.

As shown in Figure 2.3, DockerHub contains two kinds of public Docker images: official and community images. Official images are provided by DockerHub itself, which ensures regular security updates on such official images (DockerHub (2019c)). On the other hand, community images are developed and shared by the community developers.
Every DockerHub image has the following meta-data associated with it as shown in Figure 2.4:

- **Full Name**: The full name of a DockerHub community image is composed of two parts: an image's owner name and a name provided by the image's developer. Note that official images do not contain an owner name. In this chapter, we consider the developer provided name as the name of the image.

- **Short Description**: Each image has a short description which briefly presents the image.

- **Full Description**: The full description provides a detailed description about an image.
• **Tags:** An image on DockerHub may have different versions. For example, one version of an image for Ubuntu and another version for Alpine. This distinction is made using tags.

• **Download Count:** It represents the number of times an image is downloaded. DockerHub UI shows the exact number of downloads when it is lower than 1,000, and shows that number using a thousand or million scale for images with a large number of downloads as shown in the two examples of Figure 2.4.

• **Star Count:** Similar to the number of downloads, DockerHub provides the number of stars on an image. Star count is also shown using a thousand or million scale when the count is equal or higher than 1,000.

All of these abovementioned meta-data are accessible through the DockerHub HTTP API V2 (DockerHub (2019a)).

### 2.1.3 Docker Compose

One can compose a multi-component application using Docker Compose which composes a set of **components, each of which is an image and a set of options that specify how the component should behave**. One can reuse the same image for different components; the reused images will result in different component once instantiated.

Such composition of components is specified using a configuration file such as `docker-compose.yml`. Figure 2.5 shows an example of a Docker Compose file, which composes a multi-component application from the **web** and **redis** components. The **web** component is represented by a local image (built from a local Dockerfile). The **redis** component is created from the “redis” image, which is hosted on online registry
(e.g., DockerHub (DockerHub (2019b))). Additionally, both components (in Figure 2.5) have additional options that configure their environments. The web component exposes the port 5000, stores its data in an external volume (stored in the “/code” path in the host machine), and uses the common-network specification to access other components of the same multi-component application.

2.2 Related Work

While a large body of research focused on different aspects of Docker despite it being a new technology (released in 2013), a little is known about how to compose a multi-component system using Docker. The research literature on Docker focuses primarily on the quality and evolution of Dockerfiles (the specification file for a Docker image), Docker security, and Docker performance.
2.2.1 Quality and the Evolution of Dockerfiles

Zhang et al. (2019) identified six different evolutionary patterns for Dockerfiles which are: increasing and holding, constantly growing, holding and increasing, increasing and decreasing, holding and decreasing, gradually reducing. Later, Zhang et al. (2018) investigated the impact of these different evolutionary patterns on the quality and build latency of Docker images. They observed that a decrease in the number of image layers and size along with more diverse instructions can improve both the quality and the build time significantly for Docker images. Cito et al. (2017) studied 70,000 Dockerfiles from Github and reported that Dockerfiles only change 3.11 times per year. They also reported that a typical Dockerfile violates on an average 3.1 of the best practices that are recommended by Docker (Docker (2019a)).

2.2.2 Docker Security

Tak et al. (2018) and Zerouali et al. (2019) analyzed images from DockerHub (DockerHub (2019b)) to examine the security vulnerabilities of existing images. Tak et al. (2018) analyzed the 10,000 most popular (based on their stars and downloads) images on DockerHub. They observed that over 92% of the studied images have security or compliance issues. While Zerouali et al. (2019) studied the outdated and most up-to-date containers by using 7,380 official and community Docker images on DockerHub. They observed that even the most up-to-date Docker images have severe security vulnerabilities. Shu et al. (2017) developed a Docker image Vulnerability Analysis tool named DIVA which discovers, downloads, and analyzes images from DockerHub and reports any security vulnerabilities in these images.
While other researchers focused on identifying the security issues of Docker itself. For example, Jian and Chen (2017) reported that Docker containers are vulnerable to escape attacks where an attacker can run their code in the host by escaping the boundary of an insecure privileged Docker container. Combe et al. (2016) and Chelladhurai et al. (2016) discussed that containers are vulnerable to the ARP poisoning attack due to them sharing the same network bridge. Luo et al. (2016) identified several potential covert channels which can pose security issues like information leak between containers. Martin et al. (2018) identified various vulnerabilities in the entire Docker ecosystem and categorized these vulnerabilities into five categories, while Mostajeran et al. (2017) identified three major sources of vulnerabilities.

On the other hand, researchers identified approaches to improve the security of Docker. For example, to protect against ARP poisoning attack, Combe et al. (2016) recommended that practitioners should avoid sharing the same network bridge among multiple containers. While Chelladhurai et al. (2016) showed that limiting the memory for each container can protect the system against DoS attacks. Higgins et al. (2016) proposed four general guidelines which can strengthen the security of a container. Luo et al. (2016) proposed minimizing the use of covert channels by restricting the channels regarding hardware information as well as controlling the reading of system log information to avoid information leak between containers through covert channels. Jian and Chen (2017) presented a tool that can detect the escape behavior of any process automatically at runtime to prevent the container escape attack. Lei et al. (2017) developed a security mechanism named SPEAKER that can minimize the system calls from Docker containers at runtime by 35-50% to minimize the possibility of attackers gaining access to the kernel through system calls. Rastogi et al. (2017) claimed that complex
Docker containers which contain multiple application packages along with many unnecessary resources can be vulnerable to attacks which can be minimized by developing a tool named CIMPLIFIER that can divide a complex container into multiple simple containers along with removing unnecessary resources. de Guzmán et al. (2018) developed the i2kit tool which uses Linuxkit (Linuxkit (2019)) to create lightweight VMs to deploy containers inside those VMs to ensure the VM level security as well as to preserve the benefits of containerization. Loukidis-Andreou et al. (2018) developed a tool called Docker-sec which can ensure the security of a Docker container at runtime by constraining the capability of the container based on a set of rules. Syed and Fernández (2018) proposed a reference architecture which they claim can improve the overall security and reliability of the Docker ecosystem. While Zhang et al. (2018) proposed a network control architecture for large-scale container clusters which use VLAN partitioning and iptables to ensure the security of the network.

### 2.2.3 Docker Performance

Another category of research focused on the performance analysis of Docker. For example, Morabito (2016), Zhang et al. (2016), Saha et al. (2018), Xie et al. (2018), and Muhtaroglu et al. (2017) reported that Docker containers add a small amount of overhead on constrained devices such as Raspberry Pi. However, Muhtaroglu et al. (2017) argued that Docker drastically reduces the deployment time of packages by 10-15 times. While Li et al. (2017), Wang et al. (2017), Shirinbab et al. (2017), Acharya et al. (2018), Jawarneh et al. (2019), Zhang et al. (2018), Kavitha and Varalakshmi (2018), and Salah et al. (2017) compared performance of the applications that are deployed with containers and VMs. Wang et al. (2017) and Shirinbab et al. (2017) showed that VMs have
more performance overhead than Docker containers by evaluating the performance of
different applications using both technologies. Zhang et al. (2018) observed that in the
big data environment a VM usually takes 50.72 times longer than a Docker container to
boot up. They also observed that the application performance is 14.31 times higher in
the Docker container than that of the VM. However, Li et al. (2017) observed that VMs
perform better than Docker containers for reading or writing data.

On the other hand, researchers worked on improving the performance of Docker.
For example, Anderson et al. (2016) showed that by using advanced networking mech-
anisms like NFV, Docker containers can achieve low latency as well as lower variability.
Li and Fang (2017) proposed a new container scheduling algorithm which they call
Multi-Algorithm Collaboration Scheduling Strategy to achieve a better load balancing
capability for Docker containers. Nakazawa et al. (2017) showed that per-container
swappiness can allow the accommodation of a large number of containers in a heav-
ily loaded environment. Mizusawa et al. (2018) proposed a method to minimize the
synchronization of OverlayFS which can significantly improve the performance of I/O
operations in containers without affecting their integrity.
A Study of How Docker Compose is Used to Compose Multi-component Applications

Many modern software applications are composed of several components (e.g., a web application is composed of a web server component and a database component). Each of these components can be instantiated from a Docker image. Each Docker image corresponds to a software package (e.g., Apache or MySQL) along with various configuration details. Such composition simplifies, speeds up, and enables the systematic deployment and maintenance of components at scale. As a natural progression of Docker, applications are now using “Docker Compose” to compose multi-component applications by specifying the various components and their relations – in turn simplifying the deployment and maintenance of complex multi-component applications. This chapter reports on a study of 4,103
open-source Github projects that use Docker Compose. Our primary goal is to better understand how it is used in the wild. We observe that over a quarter (26.8%) of the studied projects use Docker Compose needlessly to compose single-component application. The Docker Compose file for an application is infrequently updated with 30% of such files never changed. We also observe that most of the composed applications leverage basic Docker Compose options instead of using advanced options (e.g., just 4.3% of the multi-component applications use a security related option). While Docker Compose has evolved over the years (it is currently at version 3), applications rarely adopt the new versions and 2.4% of the studied projects downgraded to an earlier version due to platform and option compatibility issues. Our study highlights that while applications are using Docker Compose, they appear to be content with its basic options and earlier versions in many instances. Future studies are needed to better understand how to improve the uptake of the more advanced aspects of Docker Compose, if they are needed at all.

### 3.1 Introduction

Docker (Docker (2019b)) enables the containerization of a software package along with its associated configuration and setup details. Such containers can be easily and rapidly deployed while avoiding compatibility issues. In fact, a recent study reports that Docker can speed up the deployment of software components by at 10-15 folds (Muhtaroglu et al. (2017)).

Docker is one of the most popular containerization technologies nowadays. Docker has captured 83% of the containerization market (Serverwatch (2019)) – a market with an estimated revenue of $2.7 billion (451research (2019)) by 2020. Datadog (Datadog
(2019a) (an emerging cloud service monitoring startup) reported in 2018 that around a quarter of their customers have already adopted Docker (Datadog (2019b)).

Much of today’s applications are multi-component applications. For instance, a simple web application would require a web server and a database component. Docker Compose, a natural progression of Docker, enables practitioners to compose such complex applications (DockerCompose (2017b)). Applications transcribe such compositions in a Docker Compose file, where components are specified by describing their Docker image and associated configuration as well the relations between components. For example, one can specify a database component that uses a MySQL Docker image and stores data in a given directory. Furthermore, one can specify various actions to follow when a component fails.

Prior studies of Docker mostly focused on Docker images without considering the use of such images in multi-component applications. For example, Tak et al. (2018), Shu et al. (2017), and Zerouali et al. (2019) studied the security of Docker images, while Cito et al. (2017), and Zhang et al. (2018) studied the evolution of Dockerfiles (i.e., the specification files for Docker images).

This chapter reports on a study of 4,103 Github open-source projects that use Docker Compose. Our goal is to gain a solid empirical understanding of how applications use Docker Compose. We structure our study along the following research questions:

**RQ1. How do applications leverage Docker Compose?**

Over a quarter (26.8%) of the projects use Docker Compose needlessly to compose single-component applications. Among the multi-component applications, 23% of them compose only local images, 40.2% of them compose both local and registry-hosted images, and the remaining 36.8% compose only registry-hosted images.
RQ2. What are the most used Docker Compose options?

22.6% of the available Docker Compose options are not used in any studied application. Advanced options like the ones for security and logging are rarely used. The security and logging options are used by 4.3% and 1.7% of the studied projects respectively.

RQ3. How do Docker Compose files evolve?

Docker Compose files change infrequently. Changes primarily occur to image related options (33.9% of changes), i.e., how to build an image, and data management options (24.9% of changes), i.e., how to store the data. Applications opt to pin the version of their composed images after facing compatibility issues between images and their applications (e.g., due to library version updates).

RQ4. How do developers use different versions of Docker Compose?

21.5% of the studied applications are still using version 1 of Docker Compose which will become deprecated in the very near future. A small number (8.5%) of the projects upgraded their Docker Compose version. 2.4% of the projects downgraded their Docker Compose version – many of the downgrades are due to applications wanting to use options that are no longer available in the new version of Docker Compose. An upgrade/downgrade requires changing a median of 10 lines of Docker Compose file code, which is double the typical number of changed lines in a regular change.

Our study highlights that while applications are using Docker Compose, they appear to be content with its basic options and earlier versions of Docker Compose in many instances. Future studies are needed to better understand how to improve the uptake of the more advanced aspects of Docker Compose, if they are needed at all.
CHAPTER 3. A STUDY OF HOW DOCKER COMPOSE IS USED TO COMPOSE
MULTI-COMPONENT APPLICATIONS

3.2 Data Collection

We wish to study a large number of non-trivial applications which are composed using Docker Compose. Hence, we first queried the Github database in Google Big Query (Google (2019a)) to retrieve projects that contain at least one docker-compose.yml file. We obtained an initial list of 21,269 projects. We then removed deleted projects (since they are no longer available) and projects that were forked from another project (to avoid any bias due to the duplicated maintenance activities of such projects). We obtained such projects from the GHTorrent database (Gousios (2013)) (updated until April, 2019). We ended up with a list of 16,283 projects.
To avoid studying trivial (e.g., toy or personal) projects, we filtered out projects with less than 100 commits (this commit threshold was used by previous study (Gallaba and McIntosh (2018))), so we ended up with 5,139 projects as shown in Figure 3.1. We then cloned these projects from Github. However, we could only obtain 4,917 projects since the remaining projects are no longer available on Github (e.g., made private or deleted after April 2019). We limited our dataset to projects that contain a single docker-compose.yml file since it is not feasible to know which Docker Compose file is used. We ended up with 4,327 projects. We also removed any projects with an empty Docker Compose file, which resulted in a dataset with 4,136 projects. While parsing the Docker Compose files we could not parse 33 files due to the incorrect syntax of these files. Hence, we removed these projects which resulted in a final dataset of 4,103 projects.

3.3 Results

The goal of this chapter is to better understand how developers compose their applications using Docker images. To achieve this goal, we address the following four research questions:

RQ1. How do developers build their systems using Docker Compose?

RQ2. What are the most used Docker Compose options?

RQ3. How do Docker Compose files evolve?

RQ4. How do developers use different versions of Docker Compose?

RQ1. How do applications leverage Docker Compose?

Motivation: The goal of this research question is to understand how applications are composed using Docker Compose. Such an empirical understanding of how Docker
Compose, a relatively new approach, is used for automating the composition of multi-component applications would help researchers and practitioners understand common practices and help identify open research and practical challenges.

**Approach:** To understand how applications are composed using Docker Compose, we parse the Docker Compose file of each of the selected projects to find out the images that they use. In this regard, we examine the `build` and `image` options for each defined component in the parsed Docker Compose files (as shown in Figure 2.5). For instance, we identify local images and online registry images that are hosted on online registries such as DockerHub.

We also investigate the commonly used online registries. To identify images that are hosted on online registries, we identified images that are specified using the following four patterns (Brown (2015)):
• index.docker.io/{repository}/{image_name}

• docker.io/{repository}/{image_name}

• {repository}/{image_name}

• {image_name}

Furthermore, we also identify official DockerHub-hosted images by searching for images that are specified using the following patterns (Brown (2015)):

• library/{image_name}

• {image_name}

Results: Over a quarter (26.8%) of the studied applications needlessly use Docker Compose to compose single-component applications, even through the primary goal of Docker Compose is to compose multi-component applications (DockerCompose (2017b)). The studied applications are composed from a median of two components and as much as 78 components (as shown in Figure 3.2).

A closer examination of the identified single-component applications reveals that Docker Compose is used for two reasons: 1) 35% of such applications use the incremental build feature of Docker Compose so the application does not need to be re-instantiated unless its associated Docker image has been updated – such a use case can be achieved through a simple Makefile, and 2) the remaining (65%) of the single-component applications use Docker Compose to specify the mapping of the virtual storage in a component to an actual physical location on the host machine – such a use case can be achieved by specifying such a mapping at the command line of the Docker
command when the Docker image is being instantiated. As we noted both aforementioned use cases can be achieved without the use of Docker Compose, however applications are opting to use it. Future studies are needed to get a better understanding of the rationale for such use cases. In particular, whether such use cases should be supported by Docker itself or whether Docker Compose should provide richer support for such use cases.

15.6% of the studied multi-component applications specify components which reuse the same Docker image. We observe that some images are reused by as little as one component and as much as 51 components in the same multi-component application.

With many applications having many components that reuse the same Docker Image; we observe that definitions of such components have a large amount of duplication (as the definitions of these components are often quite similar modulo some minor differences). Instead Docker Compose should consider adding the option to create component templates to reduce the duplication. Surprisingly, versions 1 and 2 of Docker Compose had an extends option which helped in reusing the configuration of one component (similar to inheritance in object oriented programming). However, that option is no longer supported in version 3.

Multi-component applications leverage components that are built from local Docker images as well as registry-hosted (mostly DockerHub) images. Among the multi-component applications, 23% of them compose only local images, 40.2% of them compose both local and registry-hosted images, and the remaining 36.8% compose only registry-hosted images. Finally, 1.7% (134) of the multi-component applications are
CHAPTER 3. A STUDY OF HOW DOCKER COMPOSE IS USED TO COMPOSE
MULTI-COMPONENT APPLICATIONS

built dynamically. Such applications receive their image name as a command line pa-
rameter.

**DockerHub is the most used online registry for remote images.** 95.2% of the iden-
tified registry-hosted images are hosted on DockerHub, while just the remaining 4.8% of the registry-hosted images are hosted on registries such as Quay (124 images) or
Google Container Registry (21 images). 53.5% of the identified DockerHub images are
official images, while the remaining (46.5%) images are either community or private
images on DockerHub. We had expected more prominent use of official images. Fu-
ture research is needed to investigate the rationale for applications not using official
images as prominently in an effort to improve such official images.
**The most popular DockerHub images are related to infrastructure images.** Most of the popular images are related to infrastructure components such as databases (e.g., Postgres, Redis, Mongo, MySQL) and web servers (e.g., Nginx), as shown in Figure 3.3. Several popular combinations of images exist that are co-used in a considerable number of applications. For example, the most popular combination of images is Postgres and Redis, which accounts for 5.9% (99 applications) of the multi-component applications that use more than one registry-hosted image (1,686 applications) as shown in Figure 3.4.
CHAPTER 3. A STUDY OF HOW DOCKER COMPOSE IS USED TO COMPOSE
MULTI-COMPONENT APPLICATIONS

Summary of RQ1

26.8% of the studied applications needlessly use Docker Compose to create single-component applications. Multi-component applications are being composed from local and registry hosted components with most of the registry hosted components from DockerHub.

RQ2. What are the most used Docker Compose options?

Motivation: The goal of this research question is to identify the commonly used Docker Compose options and the ones that are not used. Follow up research is needed to better understand the rationale for such usage patterns and the impact of such patterns on the evolution of Docker Compose itself.

Approach: To identify the commonly used Docker Compose options, we first collect the keywords for the existing Docker Compose options that are available across the three major versions of Docker Compose (DockerCompose (2019a,c,d)). Then, we parse our dataset of Docker Compose files to determine the usage frequency for these options. To better understand how these options are used, we also use association rule mining to identify the co-occurrence of options (Agrawal et al. (1993)). Finally, we classified Docker Compose options based on their goals to identify the common reasons for using different options.

Results: Applications mostly use basic Docker Compose options, while advanced options such as security and logging are used infrequently. We categorized the 115
CHAPTER 3. A STUDY OF HOW DOCKER COMPOSE IS USED TO COMPOSE MULTI-COMPONENT APPLICATIONS

(a) Percentage of projects using different categories of Docker Compose options.

(b) Distribution of proportion of used options across each option category. The number inside parentheses in the y-axis indicates the number of options in that category.

Figure 3.5: Usage of different option categories in Docker Compose files.
existing options into 12 categories as shown in Table 3.1. The most used Docker Compose options are related to building a component (Image Config options), accessing it (Accessibility options), and managing its data (Data Management options). While sensitive options, such as security or logging options are rarely used. The
security and logging options are used by only 4.3% and 1.7% of the studied projects respectively as shown in Figure 3.5a. Even when applications use a category of options, they use a small proportion of its options as shown in Figure 3.5b. For example, applications use a median of 9% of the Security options.

22.6% of the total options that were introduced in major versions of the Docker Compose were never used in our studied projects. The unused options are listed in Table 3.2: 12 options are for resource management, 5 options are security related, 4 options are for process specification, and 3 options are for network configuration. For example, no studied project uses the rollback_config option, although it is important to specify the rollback action to be taken when a component fails.

ports is the most used option (used by 83.3% of the studied projects). ports exposes an internal port of an image so other images and external resources can access
it. The options `image` and `build` are within the top 4 most used options (as shown in Figure 3.6) since these options are the core Docker Compose options – they are used to create and compose images.

73.8% of the applications manage their resources outside their components, as shown in Figure 3.6. The `volumes` option helps export data outside a component. Since components are stateless and the data is destroyed once a component is destroyed, one must use the `volumes` option to make the application data accessible on the host machine, and to enable the sharing of the data with other components as well.

<table>
<thead>
<tr>
<th>restart</th>
<th>context</th>
<th>links</th>
<th>build</th>
<th>depends_on</th>
<th>environment</th>
<th>volumes</th>
<th>ports</th>
<th>command</th>
<th>image</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.2</td>
<td>0.3</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.3</td>
<td>1.0</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>0.1</td>
<td>0.5</td>
<td>0.4</td>
<td>0.4</td>
<td>0.3</td>
<td>0.5</td>
<td>0.1</td>
<td>0.5</td>
<td>0.0</td>
<td>0.3</td>
</tr>
<tr>
<td>0.2</td>
<td>0.3</td>
<td>0.2</td>
<td>0.2</td>
<td>0.3</td>
<td>1.0</td>
<td>0.3</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
</tr>
<tr>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>0.4</td>
<td>1.0</td>
<td>0.5</td>
<td>0.4</td>
<td>0.5</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7</td>
<td>0.5</td>
<td>1.0</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.6</td>
<td>0.0</td>
<td>0.6</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td>0.5</td>
<td>0.6</td>
<td>0.6</td>
<td>0.5</td>
<td>0.6</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td>0.2</td>
<td>1.0</td>
<td>0.2</td>
<td>0.3</td>
<td>0.3</td>
<td>0.4</td>
<td>0.3</td>
<td>0.3</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>1.0</td>
<td>0.6</td>
<td>0.7</td>
<td>0.6</td>
<td>0.7</td>
<td>0.6</td>
<td>0.2</td>
<td>0.5</td>
<td>0.0</td>
<td>0.8</td>
</tr>
</tbody>
</table>

The confidence metric values of co-occurrence of the top 10 most used options appearing in the same component.

Figure 3.7: Confidence metric values of co-occurrence of the top 10 most used options appearing in the same component.
Table 3.2: Options in Docker Compose that are never used. Note that the classification of these options is based on our own analysis.

<table>
<thead>
<tr>
<th>Category</th>
<th>Options</th>
<th>Supported versions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Management</td>
<td>storage_opt</td>
<td>2</td>
<td>Sets the option for using a storage driver in a component.</td>
</tr>
<tr>
<td></td>
<td>volume_driver</td>
<td>1, 2</td>
<td>Specifies the default volume driver for all used volumes in a component.</td>
</tr>
<tr>
<td>Network Config</td>
<td>dns_opt</td>
<td>2</td>
<td>List of custom DNS options to be added to a component’s resolv.conf file.</td>
</tr>
<tr>
<td></td>
<td>link_local_ips</td>
<td>2</td>
<td>Lists the link-local IPs for a network used by a component.</td>
</tr>
<tr>
<td></td>
<td>mac_address</td>
<td>2</td>
<td>Sets the mac address that will be used by a component.</td>
</tr>
<tr>
<td>Resource Management</td>
<td>blkio_config</td>
<td>2</td>
<td>Sets the limits for block IO in a component.</td>
</tr>
<tr>
<td></td>
<td>cpu_count</td>
<td>2</td>
<td>Sets the allocated number of CPUs for a component to run (option is only available for Windows systems).</td>
</tr>
<tr>
<td></td>
<td>cpu_period</td>
<td>2</td>
<td>Sets the period for the CPU CFS (Completely Fair Scheduler) used by a component.</td>
</tr>
<tr>
<td></td>
<td>cpu_rt_period</td>
<td>2</td>
<td>Sets the CPU real-time period for a component.</td>
</tr>
<tr>
<td></td>
<td>cpu_rt_runtime</td>
<td>2</td>
<td>Sets the real-time runtime for a component.</td>
</tr>
<tr>
<td></td>
<td>device_read_bps</td>
<td>2</td>
<td>Sets the limit in bytes per second for read operations on a given device.</td>
</tr>
<tr>
<td></td>
<td>device_read_iops</td>
<td>2</td>
<td>Sets the limit in operations per second for read operations on a given device.</td>
</tr>
<tr>
<td></td>
<td>device_write_bps</td>
<td>2</td>
<td>Sets the limit in bytes per second for write operations on a given device.</td>
</tr>
<tr>
<td></td>
<td>device_write_iops</td>
<td>2</td>
<td>Set a limit in operations per second for write operations on a given device.</td>
</tr>
<tr>
<td></td>
<td>mem_swapiness</td>
<td>2</td>
<td>Specifies the total memory limit including the swap memory of a component.</td>
</tr>
<tr>
<td></td>
<td>weight</td>
<td>2</td>
<td>Sets the proportion of allocated bandwidth of a component with respect to other components.</td>
</tr>
<tr>
<td></td>
<td>weight_device</td>
<td>2</td>
<td>Sets the relative bandwidth allocation of a device by a component.</td>
</tr>
</tbody>
</table>
### Chapter 3. A Study of How Docker Compose is Used to Compose Multi-Component Applications

<table>
<thead>
<tr>
<th>Category</th>
<th>Options</th>
<th>Supported versions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process Spec</strong></td>
<td><strong>oom_kill_disable</strong></td>
<td>2</td>
<td>Boolean value to enable or disable OOM (Out Of Memory) killer for a component.</td>
</tr>
<tr>
<td></td>
<td><strong>pids_limit</strong></td>
<td>2</td>
<td>Sets the PID limits for a component.</td>
</tr>
<tr>
<td></td>
<td><strong>priority</strong></td>
<td>2</td>
<td>Specifies the order in which components will be connected to the networks in case of multiple network connection.</td>
</tr>
<tr>
<td></td>
<td><strong>rollback_config</strong></td>
<td>3</td>
<td>Specifies the procedure to rollback in case of an update failure.</td>
</tr>
<tr>
<td><strong>Security</strong></td>
<td><strong>isolation</strong></td>
<td>2, 3</td>
<td>Specifies the isolation technology of a component.</td>
</tr>
<tr>
<td></td>
<td><strong>userns_mode</strong></td>
<td>2, 3</td>
<td>Used to disable user namespace in a component.</td>
</tr>
<tr>
<td></td>
<td><strong>device_cgroup_rules</strong></td>
<td>2</td>
<td>Adds rules to the cgroup allowed devices list.</td>
</tr>
<tr>
<td></td>
<td><strong>group_add</strong></td>
<td>2</td>
<td>Specifies additional groups (by name or number) of which the user inside the component should be a member.</td>
</tr>
<tr>
<td></td>
<td><strong>credential_spec</strong></td>
<td>3</td>
<td>Sets the credentials for the managed components in a Windows environment.</td>
</tr>
</tbody>
</table>

**Few options co-occur together.** Although some options are used to configure other options, we observe that such options do not co-occur (as shown in Figure 3.7). For example, in just 20% image options configure their restart option – even though one should ideally configure how to restart a component in case of a failure. While 50% of the components that use the image option use the ports option. That said either components are not exposing their ports or their Docker images have already exposed a port through their Dockerfile – highlighting that a few options are configurable at the Dockerfile level or the Docker Compose level leading to inconsistencies and making it
impossible to simply examine the Docker Compose file to gain a complete view of how a multi-component application is connected.

### Summary of RQ2

22.6% of the available Docker Compose options are not used in any studied application. Advanced options like the ones for security and logging are rarely used.

### RQ3. How do Docker Compose files evolve?

**Motivation:** The goal of this research question is to examine the type of changes that occur on Docker Compose files in order to better understand the stability of such files and the complexity of their evolution.

**Approach:** To understand the evolution of Docker Compose files, we study the change history of each of the studied Docker Compose files. We measure the amount of changes that these files exhibited, as well as changes to their usage of the different Docker Compose options. These options are obtained using an approach that is similar to the one that was used in the previous research question, but on each revision of the studied Docker Compose files. Note that we do not consider merged commits to avoid double counting.

We extended our investigation of the reasons behind such changes by manually investigating the commit messages and the changes for the top two most frequently changed options. In this regard, we randomly selected a statistical representative weighted sample of 370 commits that either added, removed, or modified those two options (out of 10,143 commits with a 95% confidence level and a 5% confidence interval).
Figure 3.8: The frequency of change for Docker Compose files. The y-axis represents the cumulative percentage of Docker Compose files. For example, in Figure 3.8a, 18.4% of the Docker Compose files have at least 10 commits.

Results: Docker Compose files are revised infrequently. The median number of commits in a Docker Compose file is just three commits including the initial commit. As shown in Figure 3.8a, around 30% of the projects never changed their Docker Compose files after the initial commit. Moreover, Docker Compose files were updated just once for 14% of the projects. Furthermore, the median number of revisions of a Docker Compose file is only two revisions per year as shown in Figure 3.8b, which is similar to the finding of Cito et al. (2017) on the evolution of Dockerfiles. The number of revisions of a Docker Compose file is moderately positively correlated with the number of lines of code in that Docker Compose file (Spearman’s rank correlation coefficients $r_s = 0.49$). Note that our finding is similar to prior studies on the frequency of changing other builds artifacts (Gallaba and McIntosh (2018)).
Docker Compose files exhibit small changes. The median number of added lines in a Docker Compose file per revision is just two, while the median number of removed lines is just one per revision. Overall, the median number of changed lines in a revision of a Docker Compose file is five as shown in Figure 3.9. We do not observe any correlation between the size (number of lines) of Docker Compose file and its number of changed lines on each commit (Spearman’s rank correlation coefficients ($r_s$) = -0.01) as shown in Figure 3.10.

10.2% of the projects moved from a single-component to multi-component applications, 5.1% of the projects moved from a multi-component to single-component application. Our initial investigations on 10 Docker Compose files shows that applications removed components to cleanup their Docker Compose files, particularly, we
observe five cases where applications removed images that were used for development purposes and which are no longer needed.

**Most of the studied changes are related to image and data management options.** 33.9% and 24.9% of the studied changes are related to the basic configuration (Image Config and Data Management) as shown in Figure 3.11a. Indeed, 27.6% and 24% of the changes modify the image and volumes options respectively as shown in Figure 3.12a.

On the other hand, a median of 47.7% (considering all used options) of the projects changed their used options. For example, 50.8% of the projects changed their used networks option as shown in Figure 3.12b. Furthermore, 61.6% of the 86 projects that used Components Relation category options, changed those options. Similarly, 58.8% of the 68 projects that used logging category options, changed those options as shown in Figure 3.11b.
CHAPTER 3. A STUDY OF HOW DOCKER COMPOSE IS USED TO COMPOSE
MULTI-COMPONENT APPLICATIONS

(a) Percentage of revisions that change the different categories of Docker Compose options.

(b) Percentage of projects that use a category of options and revise it.

Figure 3.11: The frequency of the changes of different option categories based on the analysis of 2,316 projects that revised at least one Docker Compose option. Note that in Figure 3.11b, the relevant projects are the projects that use a specific option category.
Figure 3.12: Top 10 most frequently revised options in Docker Compose files based on the analysis of 2,316 projects that revised at least one Docker Compose option. Note that in Figure 3.12b, the relevant projects are the projects that use a specific option.
Table 3.3: Options that were never changed in any revision of the studied Docker Compose files.

<table>
<thead>
<tr>
<th>Category</th>
<th>Option</th>
<th>Supported versions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Config</td>
<td>ipv6_address</td>
<td>2</td>
<td>Sets the IPV6 address of a network used by a component.</td>
</tr>
<tr>
<td></td>
<td>enable_ipv6</td>
<td>2</td>
<td>Boolean value to enable or disable IPV6 addresses.</td>
</tr>
<tr>
<td>Resource Management</td>
<td>cpu_quota</td>
<td>1, 2</td>
<td>Sets the number of allowed CPU CFS (Completely Fair Scheduler).</td>
</tr>
<tr>
<td>Process Spec</td>
<td>endpoint_mode</td>
<td>3</td>
<td>Sets the method for component discovery.</td>
</tr>
<tr>
<td></td>
<td>order</td>
<td>3</td>
<td>Sets the order of operations during an update or rollback.</td>
</tr>
</tbody>
</table>

Figure 3.13: Types of modification done on images.
Applications are made more stable by pinning the versions of the used images. Most of the image related changes (85.7%) modify the version of the used image, while just 14.3% of those changes switch from one image to another one, as shown in Figure 3.13. 3.7% of these modifications remove the image's version, which is not a good practice (Docker (2017)). In fact, version pinning is a better practice which is recommended by Docker for specifying an image in a Dockerfile (Docker (2017)), which is also applicable for writing Docker Compose files since applications also need to specify an image to build a component. Version pinning avoids any unintentional version upgrades of an image which may break the application. For instance, we observe 11 cases in our manual analysis where applications pined the exact version of their used images to fix issues that were introduced due to an update of the image.

Interestingly, we observe 10 cases where applications tag a version of their image similarly to tagging source code releases, which is a fundamental continuous integration principle where stable and reproducible infrastructure images are stored similarly to the source code (Humble and Farley (2010)). For example, when the 'meenakommo-64/squid' project releases a new version such as '3.3.8-23', it also releases a similar version for the infrastructure image such as '3.3.8-23'. The goal of this tagging is to associate an infrastructure version to the release version so that applications can be easily re-built from a previous release using the appropriate infrastructure and environment.

The volumes option is the second most frequently changed option since it is changed in 24% of the studied revisions. Applications commonly modify the way that their data is managed by modifying the volumes option. We observe three typical
changes related to the volumes option: (1) 51.9% of the changes modify the volumes configuration, (2) 32.3% of the changes add a volumes to their components, and (3) 15.8% of the changes remove the volumes option. We discuss below each of these types of changes.
Figure 3.13: Examples of different types of changes to the `volumes` option.
A manual study of 197 changes to the `volumes` option shows different reasons for which applications modify that option. We observe that applications modify the synchronization method of the mounted volumes, for example, whether the change in the component data will immediately be synchronized with the physical host or whether the data will be synchronized at the time of destroying the component as shown in Figure 3.14a. Applications also modify the values of `volumes` option to fix incorrect paths. Other modifications are related to the refactoring of the data and the path (11 cases) such as restructuring the files or variable substitution (DockerCompose (2017a)).

We also observe through our manual analysis three reasons for adding a volume: (1) persisting data in the host machine, (2) sharing data between components, and (3) improving the performance of an application. (1) Through our manual analysis, we observe that applications add one or more entries to the `volumes` option to keep data persistent between the host and the component, as shown in Figure 3.13b. (2) They
also add the same named volume to multiple components to share the data among these components. (3) Finally, instead of uploading data from the host to the component, applications add a volume that is directly accessible from components. That avoids uploading data into the component at each start-up or restart of the component, which can be time consuming.

Finally, applications remove the `volumes` option in 15.8% of the changes related to that option in order to remove unnecessary resources from their components so that the components become more lightweight as shown in Figure 3.13c. In addition, applications replace `volumes` option with `tmpfs` option to use the data temporarily without writing it in the file-system.

**Docker Compose files are coupled with other files that are often used to configure the infrastructure of an application as well.** In 70.4% of the revisions to the Docker Compose file, the file is changed with a median of two other files. While Dockerfile is the most co-evolving (41.3% of the 70.4% of the changes) file with Docker Compose as shown in Figure 3.14. We also observe that `.yml` files co-evolve in 30.3% of the revisions suggesting that applications also revise additional Docker Compose files other than their primary Docker Compose files. Note that we just studied the `docker-compose.yml` file which is the default file that is considered by Docker Compose when composing an application.
Summary of RQ3

Docker Compose files are quite stable with very few lines of code changed when such files change. In addition, applications are made more stable by pinning the used versions of the images that are used by their components. Finally, applications adjust their volumes in order to synchronize the data outside the components.

RQ4. How do developers use different versions of Docker Compose?

Motivation: The goal of this research question is to identify how applications upgrade or downgrade the version of their Docker Compose, and the challenges that they face as they perform such activities, so the developers of Docker Compose would have a better understanding of how applications are adopting their versions.

Approach: There are currently three main versions of Docker Compose. Version 1 was released on November, 2015, while version 2 was released on April, 2016 and version 3 was released on January, 2017. Docker Compose team has announced that Version 1 will become deprecated in a future release. Each version adds support for new options while removing rarely used options.

To better understand the versions of Docker Compose that applications use and how often they upgrade or downgrade across these versions, we collected the used Docker Compose version for each studied project by parsing the value of the version option. For the first version of Docker Compose, no version option existed hence files with no version option are considered as using version 1.

We also studied the upgrade/downgrade of Docker Compose versions by analyzing all code changes that modified the version option. We also manually investigated all
of the changes that downgraded the Docker Compose Version (107 commits) to better understand the rationale behind such downgrades.

**Results:** Most projects (~ 90%) stick to their initial Docker Compose version. 9.8% of the studied projects changed their used Docker Compose version. Such a change to the version line is often associated with additional changes. In particular, a median of 10 lines of code are changed when a Docker Compose version is changed – double the number of changed lines of code in a regular revision (as discussed in RQ2). Such additional changes might be due to the major syntactical differences between the different versions, the configuration of new options that are offered by the new version, or to deal with the removal of options in the new version.

8.5% of the projects upgraded their Docker Compose version while 2.4% of the projects downgraded their Docker Compose version. Several (59.9%) of the upgrades/-downgrades are between major versions, although some upgrades/downgrade occur between sub-versions. Almost all (99%) changes between major versions are between versions 2 and 3. We only find three cases where the version change (upgrade/downgrade) was between versions 1 and 2. Since the structure of Docker Compose has gone through major changes between these two versions, moving between these two versions might take a considerable amount of effort.

21.5% of the studied projects use Docker Compose version 1 which will be deprecated soon according to the documentation of Docker Compose ([DockerCompose (2019b)]) (as of August, 2019). Docker Compose should consider developing a migration support plan for these projects given the large differences between version 1 and the versions 2 and 3. 35.6% of the projects used the latest versions of Docker Compose file (version 3 and its sub-versions), while 42.9% of the projects used version 2
Figure 3.15: Exact versions of Docker Compose used by the percentage of projects.

of Docker Compose and its sub-versions as shown in Figure 3.15. Although Docker Compose documentation recommends applications to specify the used minor version of Docker Compose (DockerCompose (2018)), 79.2% of the studied projects that use version 2 or 3 of Docker Compose, do not specify the minor Docker Compose version.

Many unused options have been removed in the latest versions of Docker Compose. 21 out of the 26 options that were never used in our studied projects (as per RQ1) are no longer supported by the latest Docker Compose version (i.e., version 3), 19 of these options were introduced in version 2 and are no longer supported in version 3 of Docker Compose. On the other hand, 15 other currently-in-use options in version 2 have been removed from version 3. These 15 options are used by a median of two projects (minimum of one and a maximum of 206 projects per option).
Chapter 3. A Study of How Docker Compose Is Used to Compose Multi-Component Applications

Figure 3.16: An example of a Docker Compose version downgrade. This change downgraded the Docker Compose version from version 3 to version 2.

Platform and option compatibility are the main drivers for downgrading the Docker Compose version. We manually studied all of the 107 commits that downgraded the Docker Compose version. We read through each commit message, examined the changed code and studied the official documentation of Docker Compose versions to identify the rationale for such downgrades. We identified two major reasons for downgrading the version: (1) platform (e.g., AWS and Travis CI) compatibility, and (2) option compatibility.

Cloud platforms, like AWS, on which applications are deployed might not support the latest version of Docker Compose, forcing a version downgrade. Also, applications
downgraded their Docker Compose version to be compatible with Travis CI. Finally, applications downgrade to an older version (version 2) to use options that were removed in a later version (version 3) of Docker Compose as shown in Figure 3.16.

Summary of RQ4

Many applications are using the oldest version of Docker Compose (even though such version will be deprecated in a near future release). Migrating from one version to another is not straightforward due to compatibility issues with the host (e.g., AWS) or due to the removal of options in new releases of Docker Compose.

3.4 Threats to Validity

3.4.1 External Validity

Our external threat to validity concerns the generalization of our results. We cannot generalize our results to multi-component applications that configure the Docker Compose using other file than `docker-compose.yml`. However, our analysis considers a large number (4,103) of open-source Github projects.

3.4.2 Internal Validity

Our first internal threat to validity concerns the maturity of the projects that we studied. In fact, some projects might be just personal projects (e.g., in which one learns how to use Docker Compose). To mitigate this risk, we selected projects that have at least 100 commits. This commit threshold was also used by prior study (Gallaba and McIntosh (2018)).
A second internal threat to validity concerns the analysis of the right Docker Com- pose file, which is used by a studied project. A project might define multiple Docker Compose files and studying the not used file might lead to incorrect conclusions. To mitigate this risk, we focus on projects that use just one `docker-compose.yml` file.

### 3.5 Chapter Summary

In this chapter, we conduct an empirical study to better understand the use of Docker Compose. We observe that applications do not fully benefit from the available options nor versions of Docker Compose. Indeed, 26.8% of the projects build their applications from a single image, while the primary goal of Docker Compose is to compose a multi-component application. In addition, we observe that applications use and maintain just the basic options of Docker Compose, while more advanced options are almost ignored by applications. Finally, we observe that few projects upgrade their version of Docker Compose. Future studies are needed to better understand how to improve the uptake of the more advanced aspects and version of Docker Compose, if they are needed at all.
CHAPTER 4

Too many Images on DockerHub! How Different are Images for the same System?

CONTAINERIZATION is a technique used to encapsulate a software system and its dependencies into one isolated package, which is called a container. The goal of these containers is to deploy or replicate a software system on various platforms and environments without facing any compatibility or dependency issues. Developers can instantiate these containers from images using Docker; one of the most popular containerization platforms. Furthermore, many of these images are publicly available on DockerHub, on which developers can share their images with the community who in turn can leverage such publicly available image. However, DockerHub contains thousands of images for each software system, which makes the selection of an image a nontrivial task. In this chapter, we investigate the differences among
DockerHub images for five software systems and 101 images (a total of 505 images) for each of these software systems with the goal of helping Docker tooling creators and DockerHub better guide users select a suitable image. We observe that users tend to download the official images (images that are provided by Docker itself) when there exist a large number of image choices for each single software system on the community images (images that are provided by the community developers), which are in many cases more resource efficient (have less duplicate resources) and have less security vulnerabilities. In fact, we observe that a median of 96% of the DockerHub images for the same software system are different from each other in terms of their installed libraries. Furthermore, a median of 39% of the community images are more resource efficient and a median of 83% of the community images have less security vulnerabilities compared to their official images. Unfortunately, the description of 65% of the studied images do not guide users when selecting an image (the description does not exist at all or it does not highlight the particularities of the image), we suggest that Docker tooling creators and DockerHub design approaches to distinguish DockerHub images and help users find the most suitable images for their needs.

4.1 Introduction

DockerHub (DockerHub (2019b)) is a cloud-based image registry where developers can share their images either privately with a selective group of users, or publicly with the whole community of DockerHub users. DockerHub contains a rich database of images, from which one can reuse an existing image instead of building one from scratch.
However, a large number of images exists for installing the same software system making the task of choosing an image a not so straightforward task. For example, DockerHub contains around 53,000 images just for the Nginx software system. One may end up choosing an image which is not the most suitable to fulfill their needs when other images could be more suitable. Moreover, users do not always choose just a single image, they might often need to choose and connect multiple images together to build a distributed software installation or a stack of layers such as the MEAN stack, which is composed of the following software systems: MongoDB, Express.js, AngularJS, and Node.js. In fact, more than 5,000, 9,000, 11,000 and 55,000 images exist for MongoDB, Express.js, AngularJS, and Node.js respectively.

In this chapter, we study the differences among 101 DockerHub images (one official image and 100 randomly selected community images) for five software systems (505 DockerHub images in total) in order to help Docker tooling creators and DockerHub define approaches that guide users locate the most suitable images for their varying needs. Although prior studies focus on various aspects such as security (Shu et al. (2017); Tak et al. (2018); Zerouali et al. (2019)), quality (Cito et al. (2017); Zhang et al. (2018)), and evolution (Cito et al. (2017); Zhang et al. (2019)) of Docker images, understanding the differences among DockerHub images to help users identify a suitable image has not been explored in the literature. Brogi et al. (2017) is the closest work to our study since they proposed a prototype for a multi-attribute based search tool for DockerHub images. However, their approach is limited to searching for images based on their names, sizes, and installed software distributions. Their tool can report a large number of images while providing no assistance for users who might wish to differentiate among these reported images.
In the first part of this chapter, we perform a preliminary study to understand the benefits of using a DockerHub image, as well as the efforts to identify a suitable image on DockerHub by studying five software systems and 101 DockerHub images for each of these software systems (a total of 505 DockerHub images). We observe that Docker images require a considerable amount of effort to be built from scratch. However, finding a suitable image on DockerHub is not trivial since a large number (a median of 18,751) of images exists for each software system and that number of images continues to increase at a rapid pace. We also observe that the most popular DockerHub images are the official images, although they might not be the most suitable images to fulfill the needs of a user.

In the second part of this chapter, we study the differences among the DockerHub images for the same software systems in terms of their installed libraries, resource efficiency, and security vulnerabilities. In this regard, we answer the following three research questions:

**RQ1. How different are DockerHub images for the same software system in terms of their installed libraries?**

DockerHub contains a large number of images for each software system. These DockerHub images are different from each other in terms of their installed libraries and versions with only a median of 4% of the images being identical to at least one other image. Such differences are, unfortunately, not well documented with 38.8% of the images not having any description and the descriptions of 26.2% of the images not providing any indication about the installed libraries.
RQ2. How different are these images in terms of their size and resource efficiency?
Although official images are the most popular, a median of 39% of the community images are more resource efficient (i.e., have less duplicate resources) compared to their official images. DockerHub images for the same software system have different sizes, which indicates that there are a variety of DockerHub image choices based on their used resources. Images for Java have the largest interquartile range of 194.3 MB (83.7% of the size of the official image), while images for MySQL have the lowest interquartile range of 74.1 MB (60% of the size of the official image). Similarly to RQ1, we observe that developers do not document any particularities neither about the used resources nor about the resource efficiency of their images.

RQ3. How different are these images in terms of their security vulnerabilities?
While users prefer using official images, a median of 83% of the community images have fewer security vulnerabilities than their respective official images and a median of 53% of the community images are free from any security vulnerability. However, no indication about the security of community images is provided neither by the DockerHub UI nor by the description of these images. Note that DockerHub provides a feature that identifies the vulnerabilities of an image; however, that feature is available only for the official images.

Finally, our findings suggest to Docker tooling creators and DockerHub define approaches that help users differentiate among DockerHub images and find the suitable images of one’s needs.
The rest of this chapter is organized as follows: Section 4.2 presents our data extraction. Section 4.3 discusses the results of our preliminary study. Section 4.4 provides the results of our study. Section 4.5 presents a discussion based on our observations. Section 4.6 discusses the threats to validity of our results. Finally, Section 4.7 concludes the chapter.

4.2 Data Extraction

In this chapter, we study the differences among DockerHub images for the same software system using four different data sources as shown in Figure 4.1. Our data-set contains 505 DockerHub images, which consists of the official and 100 randomly-selected community images for each of the following five popular software systems: Cassandra, Java, Mongo, MySQL, and Nginx. We ensure that each studied community image respects the following criterion: the image should install the target software system (indicated by its name) using Advanced Package Tool (APT), Alpine Linux package manager (APK), or Red Hat Package Manager (RPM). Our comparisons use the following four perspectives:

**Libraries:** We use the *Snyk-docker-analyzer* ([Snyk (2019)](https://www.snyk.io/)) to extract the installed libraries on each of the 505 studied images. *Snyk-docker-analyzer* pulls each of the 505 images from DockerHub, builds a container for each of these images, executes these containers, then extracts their installed libraries. The *Snyk-docker-analyzer* extracts libraries that are installed with the APT, APK, or RPM package managers. The *Snyk-docker-analyzer* provides for each image the name of its installed libraries, their versions, and how they are installed: manually by the developer of that image, or automatically via an implicit dependency.
CHAPTER 4. TOO MANY IMAGES ON DOCKERHUB! HOW DIFFERENT ARE IMAGES FOR THE SAME SYSTEM?

Figure 4.1: Overview of the data extraction approach used to answer our RQs. Each studied image installs the software system that is mentioned by its name using Advanced Package Tool (APT), Alpine Linux package manager (APK), or Red Hat Package Manager (RPM).

**Resource Efficiency:** We use Dive (Wagoodman (2019)) to measure the resource efficiency for each of our selected images. The Dive tool evaluates the resource efficiency of a Docker image based on the amount of its wasted space (duplicate resources). For example, one image may contain multiple identical files in different layers or even in the same layer which is a waste of space. The resource efficiency of an image is measured as follows:

\[
E = \left(\frac{S - W}{S}\right) \times 100
\]

\(E\) = Resource efficiency
\(S\) = Size of the image
\(W\) = Wasted space due to duplicate file resources

**Security Vulnerabilities:** We use the publicly available Common Vulnerabilities and Exposures (CVE) (CVE (2019)) dataset to identify the security vulnerabilities
of the installed libraries on each of our studied images. That dataset of security vulnerabilities contains the list of all publicly known vulnerabilities, their affected libraries and versions, their status (open or resolved), and the severity of the vulnerability (low, medium, or high).

**Meta-data:** We collect the size, the number of downloads, and the number of stars of each of the 505 images using the DockerHub HTTP API (DockerHub (2019a)).

### 4.3 Preliminary Study: The Advantages and Abundance of DockerHub Images

The goal of our preliminary study is to understand the advantages of using an existing image on DockerHub instead of building one from scratch, as well as the difficulty of finding an image on DockerHub. We structure our preliminary study through the following preliminary research questions:

PQ1. How many libraries one has to install for a typical software system?

PQ2. How many images are available for a typical software system on DockerHub? How does the number of images evolve over time?

**PQ1. How many libraries one has to install for a typical software system?**

**Motivation:** The goal of this preliminary research question is to understand the benefits of using a DockerHub image. In other words, the goal is to understand the amount of efforts that practitioners can save by using one of the available DockerHub images instead of building an image from scratch.
**CHAPTER 4. TOO MANY IMAGES ON DOCKERHUB! HOW DIFFERENT ARE IMAGES FOR THE SAME SYSTEM?**

**Approach:** To measure the development and maintenance efforts that one can save using a DockerHub image, we count the number of libraries which are manually installed as well as the libraries that are automatically installed to quantify the required efforts for building and maintaining a single Docker image. Note that we extract the libraries that are installed in an image following the approach discussed in Section 4.2.

**Results:** Developing a new Docker image might require a considerable amount of installation and testing effort. Indeed, we observe that one has to manually install a large number of libraries whose median ranges between 81 and 110 installed libraries. The situation gets even worse when it comes to building a distributed software system, which involves building and testing multiple Docker images.

Besides installing and testing an image built from scratch, maintaining an image can be more challenging. Prior studies (Decan et al. (2018); Zerouali et al. (2019)) suggest that many libraries can introduce bugs as well as security issues in a Docker image. Therefore, developers need to check that the libraries that they install are secure and error free. Unfortunately, developers do not only need to check the health of the manually installed libraries, but also libraries that are automatically installed via implicit dependencies. In fact, the number of these automatically installed libraries is not negligible since we observe that a median of 71 to 154 libraries are installed automatically as shown in Figure 4.2.

**PQ2. How many images are available for a typical software system on DockerHub? How does the number of images evolve over time?**

**Motivation:** The goal of this preliminary research question is to understand the variety of choices that exist to build a single software system using a DockerHub image. We
believe that the larger the number of available images is, the more challenging it is to find the appropriate image. Therefore, this question investigates the number of images that are provided for each of our five studied software systems, and how that number of images grows over time.

**Approach:** To understand the variety of DockerHub image choices a user has and how such choices evolve, we first report on the number of images that are reported by DockerHub when searching for the name of each of our five studied software systems. Then, we study the evolution of the number of existing images. In this regard, we collect for each image when its first tag was updated using the DockerHub API since DockerHub does not provide the exact creation date of an image.
CHAPTER 4. TOO MANY IMAGES ON DOCKERHUB! HOW DIFFERENT ARE IMAGES FOR THE SAME SYSTEM?

Figure 4.3: The evolution of the number of images on DockerHub.

Table 4.1: Number of images available on DockerHub for the five studied software systems.

<table>
<thead>
<tr>
<th>Image type</th>
<th># of images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nginx</td>
<td>52,749</td>
</tr>
<tr>
<td>Java</td>
<td>22,891</td>
</tr>
<tr>
<td>MySQL</td>
<td>18,751</td>
</tr>
<tr>
<td>Mongo</td>
<td>12,240</td>
</tr>
<tr>
<td>Cassandra</td>
<td>2,366</td>
</tr>
</tbody>
</table>

Results: For each software system, users have a variety of image choices on DockerHub. We observe that DockerHub has an active community of developers as the median number of images that are provided for each software system is 18,751 images. The number of available images for a software system ranges between 2,366 images for Cassandra to 52,749 images for Nginx as shown in Table 4.1. Therefore, choosing one out of the thousands of the available images might be challenging.
Furthermore, as the number of images for each single software system continues to increase over time, the difficulty of choosing a suitable image increases. We observe an increasing trend in the number of images for each of our studied software systems as shown in Figure 4.3. MySQL and Cassandra have taken around just 2 years (2017 to 2019), while Java, Mongo, and Nginx have only taken around 1.5 years (mid-2017 to 2019) to double their number of images. While that increase in the number of images may enrich the DockerHub community, it increases the difficulty of choosing a suitable image.

Although a large number of community images exists, the official images are the most popular images. We observe that official images have a median of 941,443,586 downloads and a median of 5,978 stars, while the most popular community images for each type of software system have a median of 7,495,429 downloads and a median of only 60 stars. This result shows that official images have more than 100 times downloads and nearly 100 times more stars than the most popular community images.

Summary of Preliminary Study

Using an image from DockerHub might save the efforts to install, test, and maintain a large number of libraries. However, choosing one image out of the thousands of available images is challenging. Therefore, users often download and star official images, while better community images might exist. Our results suggest a need for a study to identify how different are DockerHub images, which can indicate the number of true choices one has for a single software system.
4.4 The Differences among DockerHub Images

In this section, we present the research questions along with their motivations, approaches and the findings of our study. We also provide suggestions to DockerHub based on our findings.

The goal of this section is to better understand the differences among DockerHub images in order to help users find suitable Docker images. Since we find in our preliminary study that users have a large number of image choices for building a single software system on DockerHub, this section compares the images that are provided for the same software system on DockerHub. For example, we compare images together whose names contain Nginx. We summarize our comparisons along the following perspectives:

RQ1. How different are DockerHub images for the same software system in terms of their installed libraries?

RQ2. How different are these images in terms of their size and resource efficiency?

RQ3. How different are these images in terms of their security vulnerabilities?

**RQ1. How different are DockerHub images for the same software system in terms of their installed libraries?**

**Motivation:** While DockerHub provides a large number of images for each software system (PQ1) and users tend to select official images (PQ2), the goal of this research question is to understand whether there are a large variety of choices for each single software system, or if all the images are similar to the official image and to each other.

**Approach:** To evaluate the variety of choices one has for each software system, we compare DockerHub images based on their installed libraries. We first compare the
community images with their corresponding official image. Afterwards, we compare each pair of community images in terms of their installed libraries. For example, considering the images a, b, and c, we compare a-b, a-c, and b-c based on their installed libraries. Note that we use the Snyk-docker-analyzer to extract the installed libraries of an image as discussed in Section 4.2.

Figure 4.4: The Percentage of libraries that are shared between community and official images. The x-axis represents the cumulative percentage of shared libraries between the community images and their corresponding official image. The y-axis represents the percentage of community images.

Figure 4.5: The percentage of libraries that each pair of images share. The x-axis represents the cumulative percentage of shared libraries between a pair of images. The y-axis represents the percentage of image pairs.
Results: DockerHub images are different from each other in terms of their installed libraries considering the exact versions of these libraries. We first observe that community images are different from their corresponding official image. Indeed, only one community Cassandra image and two community Java images share 100% of their installed libraries (with exact library versions) with their corresponding official image, while none of the images for the other three software systems is identical to their corresponding official image. Furthermore, the official image and a median of 15% of the community images for each software system share just 25% of the libraries as shown in Figure 4.4a.

In addition to the differences between community and official images, community images are different from each other in terms of their installed libraries considering the exact versions of the installed libraries. A median of just 4% (ranges between 3% and 8%) of the images have at least one other image with which they share 100% of their libraries and versions. Moreover, our comparison shows that a median of just 12.6% of the image pairs share 25% of the exact version of their installed libraries as shown in Figure 4.5a.

DockerHub images are different from each other in terms of their installed libraries independently from these libraries' versions. Ignoring the versions of the libraries, we observe that just 10% of the Nginx, 5% of the Cassandra, 5% of the Java, 2% of the Mongo, and 2% of the MySQL images are identical to their corresponding official image. Figure 4.4a shows that almost none of the community images is identical to their official image, and a median of just 33% of the community images share 50% of the installed libraries with their corresponding official image (irrespective of their different versions) as shown in Figure 4.4b.
In addition to the differences between community and official images, community images are different from each other. A median of just 24% (ranges between 10% and 45%) of the images share exactly the same set of libraries with at least one other image (irrespective of their versions). Our comparison also shows that a median of just 19.1% of the image pairs for the same software system share a maximum of 50% of their installed libraries (irrespective of their versions) as shown in Figure 4.5b.

While DockerHub images are different from each other, **distinguishing images from their description is not straightforward as most of the DockerHub images are not well documented.** We observe that 38.8% of the studied images do not have any description. Furthermore, our manual study of the remaining 61.2% of the images shows that 26.2% of the images have a weak description which does not present the features of the images, and only 35% of the images have a good description that highlights the features of the images. Figure 4.6a and 4.6b show an example of a DockerHub image that documents its features and an image with a weak description respectively.

Furthermore, developers hardly provide any information about the original image, which can lead users to waste their efforts on unnecessary installation and testing of identical images. We manually analyze all the 25 images that are identical to at least one other image according to their installed libraries and versions, and we observe that only one single image explicitly mentions that it has been copied from the official image. Similarly, by manually inspecting the descriptions of 128 images that are different from at least one other image only by the versions of the installed libraries, we observe that only 6.25% of the images explicitly mention that they are based on another image. However, they do not provide any information regarding the differences from their original image.
CHAPTER 4. TOO MANY IMAGES ON DOCKERHUB! HOW DIFFERENT ARE IMAGES FOR THE SAME SYSTEM?

In addition to the weak documentation of DockerHub images, their names are often confusing as many different images have exactly the same name, which makes choosing a suitable image a nontrivial endeavour. We observe that many images have exactly the same name despite being different from each other in terms of the installed libraries. In fact, a median of 98.1% of the image pairs (between 97.5% and 98.4%) that have the same name are different in terms of their installed libraries (ignoring versions) as shown in Table 4.2. Interestingly, a median of 44% of the images (between 31% and
CHAPTER 4. TOO MANY IMAGES ON DOCKERHUB! HOW DIFFERENT ARE IMAGES FOR THE SAME SYSTEM?

Table 4.2: Library similarities of same name image pairs.

<table>
<thead>
<tr>
<th>Image type</th>
<th>% of the same name image pairs with different libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ignoring library versions</td>
</tr>
<tr>
<td>Cassandra</td>
<td>97.9</td>
</tr>
<tr>
<td>Java</td>
<td>98.4</td>
</tr>
<tr>
<td>Mongo</td>
<td>98.1</td>
</tr>
<tr>
<td>MySQL</td>
<td>98.1</td>
</tr>
<tr>
<td>Nginx</td>
<td>97.5</td>
</tr>
</tbody>
</table>

68%) have at least one other image with exactly the same name (Table 4.3). For example, Cassandra has 55 images that have exactly the same name "cassandra". However, these images share a few libraries since we observe that the pairs of images which share the same name share a median of just 28.57% of their installed libraries (ignoring versions).

Table 4.3: Number of images that have the same name.

<table>
<thead>
<tr>
<th>Image type</th>
<th># of images</th>
<th># of groups with identical image name</th>
<th>% of the images matching at least one other image name</th>
<th>Max. size of the groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>100</td>
<td>6</td>
<td>68</td>
<td>55 (80.9%)</td>
</tr>
<tr>
<td>Java</td>
<td>100</td>
<td>7</td>
<td>33</td>
<td>19 (57.6%)</td>
</tr>
<tr>
<td>Mongo</td>
<td>100</td>
<td>7</td>
<td>44</td>
<td>20 (45.5%)</td>
</tr>
<tr>
<td>MySQL</td>
<td>100</td>
<td>8</td>
<td>53</td>
<td>32 (60.4%)</td>
</tr>
<tr>
<td>Nginx</td>
<td>100</td>
<td>4</td>
<td>31</td>
<td>22 (71%)</td>
</tr>
</tbody>
</table>

Summary of RQ1

There are a large variety of choices for each single software system on Docker-Hub images which are hard to distinguish since developers do not describe the particularities of their images.
**RQ2. How different are these images in terms of their size and resource efficiency?**

**Motivation:** The goal of this research question is to identify the differences among the images in terms of their size and resource efficiency. Duplicate resources in a Docker image can make the containers unnecessarily larger in size. A prior study (Zheng and Thain (2015)) shows that container deletion time increases as the image size increases. The slower deletion time affects the performance of the container redeployment process. Moreover, the increased container size due to duplicate resources negatively affects the cost of resource provisioning in the Cloud. Hence, it is important to know the size and the resource efficiency of an image and how they are different from other similar images.

**Approach:** To identify how images are different in terms of their sizes and resource efficiency, we collect for each image its size and calculate its resource efficiency following the approach discussed in Section 4.2. Note that the resource efficiency of an image consists of the amount of wasted space due to duplicated resources. Similarly to RQ1, we first compare the official to the community images, then community images to each other.

**Results:** DockerHub images are different from each other in terms of their sizes. A median of 47% of the community images are larger in size compared to their official images as shown in Figure 4.7, which suggests that such images might contain additional resources that can differentiate images.

The size of community images which build the same software system varies. Java images differ the most in size where their interquartile range of image size is 194.3 MB, which accounts for 83.7% of the size of their official image. Although MySQL images
Figure 4.7: Size of different types of images, where the red star shows the size of the corresponding official image.

differ the least with an interquartile range of 74.1 MB, the interquartile range is 60% of the size of their official image.

Note that the last difference among the images in terms of the size is not necessarily associated to the number of installed libraries. We observe a statistically significant difference between the number of installed libraries and the size of images for three out of five software systems, which suggests that images may contain additional resources

Table 4.4: Paired Wilcoxon signed-rank test results between image size, and library count and wasted space.

<table>
<thead>
<tr>
<th>Image type</th>
<th>Image size vs library count p-value</th>
<th>Image size vs wasted space p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>0.076</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Java</td>
<td>0.045</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Mongo</td>
<td>0.96</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>MySQL</td>
<td>0.004</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Nginx</td>
<td>&lt;0.001</td>
<td>&lt;0.001</td>
</tr>
</tbody>
</table>
Table 4.5: Comparison of the resource efficiency between official and community images. Note that more than half of the Cassandra as well as Mongo images are more resource efficient than their corresponding official image (shown in bold).

<table>
<thead>
<tr>
<th>Image type</th>
<th>Efficiency of official image</th>
<th>Median efficiency of community images</th>
<th>% of the community images more efficient than their official image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>98.2</td>
<td>98.3</td>
<td>54</td>
</tr>
<tr>
<td>Java</td>
<td>98.9</td>
<td>97</td>
<td>30</td>
</tr>
<tr>
<td>Mongo</td>
<td>91.4</td>
<td>98.2</td>
<td>78</td>
</tr>
<tr>
<td>MySQL</td>
<td>98.6</td>
<td>98.2</td>
<td>39</td>
</tr>
<tr>
<td>Nginx</td>
<td>98.2</td>
<td>97.4</td>
<td>39</td>
</tr>
</tbody>
</table>

that are not necessarily the size of the installed libraries as shown in Table 4.4. We observe that these differences are not statistically significant for the Cassandra, and Mongo images. Therefore, images may differ on resources in addition to differing in terms of the installed libraries (RQ1).

**DockerHub images are different from each other in terms of their resource efficiency.** In addition to the variety of choices that exist on DockerHub images, some images are even better than their corresponding official image in terms of their resource efficiency. Indeed, a median of 39% of the community images are more resource efficient than their corresponding official image as shown in Table 4.5. For example, 78% of the Mongo community images are more resource efficient than their official image. Besides, we do not observe any official image that is 100% resource efficient (free from any duplicate resources), while we observe a median of 5% community images that are 100% resource efficient.

Besides the difference between community and official images, community images are also different from each other in terms of their resource efficiency. For instance,
MySQL has the largest difference among the resource efficiency of its community images, which ranges between 41% (59% of space is wasted) and 100% (no wasted space). While Java has the least variation in their resource efficiency, which ranges between 69% and 100% as shown in Figure 4.8. However, for all types of images, more than 90% of the images are at least 85% resource efficient. Note that there is no association between the size of an image and its resource efficiency since the amount of wasted space is significantly different from the size of an image (Wilcoxon test, p-values < 0.001, $\alpha = 0.05$) for all types of images as shown in Table 4.4.
DockerHub image developers do not provide information regarding the resource 
efficiency of their images. By manually studying the descriptions of 50 DockerHub im-
ages (the top 10 most resource efficient images from each image type), we observe that 
image descriptions do not reveal any information about the efficiency of the images, 
while just 6% of the images specify the resources that they contain.

<table>
<thead>
<tr>
<th>Summary of RQ2</th>
</tr>
</thead>
<tbody>
<tr>
<td>In addition to the variety of choices in terms of the installed libraries, DockerHub images differ in terms of their resources as well. In addition, some choices can be better than other in terms of images resource efficiency. While users tend to download official images, a median of 39% of the community images are more resource efficient. However, finding such efficient images is not trivial since developers do not document the resource efficiency of their images.</td>
</tr>
</tbody>
</table>

**RQ3. How different are these images in terms of their security vulnerabilities?**

**Motivation:** The goal of this research question is to compare the security among Dock-
nerHub images since security is a major concern when deploying a Docker image (Bet-
tini (2015)).

**Approach:** To compare DockerHub images based on their security vulnerabilities, we extract the security vulnerabilities of the libraries that are installed on each of the 505 studied DockerHub images following the approach discussed in Section 4.2.
CHAPTER 4. TOO MANY IMAGES ON DOCKERHUB! HOW DIFFERENT ARE IMAGES FOR THE SAME SYSTEM?

Figures 4.9: Number of security vulnerabilities in the DockerHub images where the red stars show the number of security vulnerabilities in the official images.

Results: DockerHub images have different number of security vulnerabilities. We surprisingly observe that there are community images which have less security vulnerabilities than their corresponding official image, although Docker regularly applies security updates on the official images (DockerHub (2019c)). Four out of the five studied official images are not free from security vulnerabilities and 84% of the MySQL, 84% of the Nginx, 83% of the Java, and 67% of the Cassandra community images are more secure compared to their corresponding official images, as shown in Figure 4.9a.

While community images and official images are different in terms of the number of security vulnerabilities they have, we observe a large variance on the number of security vulnerabilities between community images. In fact, the interquartile ranges of security vulnerabilities are 10.25, 12.25, 15, 24.25, and 73.75 for Mongo, Java, MySQL, Nginx, and Cassandra images respectively. Moreover, 64% of the Java, 63% of the Mongo, 53% of the Nginx, 49% of the MySQL, and 34% of the Cassandra community images are
free from any type of security vulnerabilities. The median number of security vulnerabilities on the images of three out of five software systems is zero, while that median is 2.5 for MySQL images and a median of 10 vulnerabilities exists on Cassandra images as shown in Figure 4.9a.

Our last observation holds for high severity security vulnerabilities. Similar to the all types of security vulnerabilities, we observe that the median number of high security vulnerabilities in the images of four out of five software systems is less than the number of high security vulnerability in the corresponding official image. Moreover, 77% of the Nginx, 73% of the MySQL, 68% of the Mongo, and 63% of Cassandra community images have less number of high security vulnerabilities than their corresponding official image. Finally, we observe that the official image for Java and 68% of its community images do not have any high security vulnerabilities.

While the number of high security vulnerabilities in community images are different from their corresponding official image, we also observe a variance in terms of the number of high security vulnerabilities between community images for the same software system. We observe that only Cassandra images have a median of one high security vulnerabilities per image, while the remaining four image types have a median of zero high security vulnerability. The interquartile ranges for Java, Mongo, and Nginx are one, while for Cassandra and MySQL images the interquartile ranges are five and two security vulnerabilities respectively as shown in Figure 4.9b. We also observe that 68%, 68%, 67%, 51%, and 36%, of the Java, Mongo, Nginx, MySQL, and Cassandra community images do not contain any high security vulnerabilities respectively.
Except for the official images, DockerHub image developers do not provide any information regarding the security aspect of their images. By manually inspecting the descriptions of 50 DockerHub images (the top 10 most secure images from each of the five image types), we do not observe any image where developers discuss its security aspect. Note that DockerHub evaluates and shows the security vulnerabilities of just the official images and not the community images.

### Summary of RQ3

Distinguishing secure and insecure images on DockerHub is not straightforward. Official images, for which Docker evaluates and publicly displays their security vulnerabilities, are not the most secure images. Indeed, a median of 53% of the community images do not contain any security vulnerability, and a median of 83% of the community images have less security vulnerabilities than their corresponding official images.

### 4.5 Discussion

We observe from our preliminary study that using an image from DockerHub can save one’s efforts of building, testing, and maintaining an image from scratch. However, finding a suitable image might be challenging due to the large number of existing images for each software system on DockerHub. Thus, we conducted an empirical study that compares DockerHub images to better understand their differences.
Table 4.6: The percentage of community images that are more resource efficient and have less security vulnerabilities compared to their corresponding official image.

<table>
<thead>
<tr>
<th>Image type</th>
<th>% of community images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>28</td>
</tr>
<tr>
<td>Java</td>
<td>24</td>
</tr>
<tr>
<td>Mongo</td>
<td>0</td>
</tr>
<tr>
<td>MySQL</td>
<td>34</td>
</tr>
<tr>
<td>Nginx</td>
<td>33</td>
</tr>
</tbody>
</table>

DockerHub contains a large number of choices of each single software system, while the particularities of images are not well documented. Our study shows that there are a variety of choices of each single software system on DockerHub since DockerHub images install different libraries and have different resources. We observe that community images are different from their corresponding official images, while they also differ among themselves in terms of their installed libraries (RQ1). Besides, images provide different sizes which indicate the installation of different resources (RQ2). However, developers do not document the particularities of their images. Neither do they document which libraries are installed nor which resources are available on their images.

Some of these image choices might be better than others in terms of improved resource efficiency and reduced security vulnerabilities. A median of 28% and an average of 23.8% of community images that are at the same time more resource efficient and have less security vulnerabilities compared to the official images, as shown in Table 4.6 and Figure 4.10. For example, “aemr3/nginx-fpm” and “artemstd/nginx” have a good resource efficiency of 99% while containing 0 and 7 vulnerabilities respectively. These images are better than their official image which is 98% resource efficient and
Figure 4.10: The relation between the resource efficiency and security vulnerabilities of the images. Note that the green images are better than the official images in terms of resource efficiency and number of vulnerabilities, while the red images are the worst images on both perspectives. Blue and yellow images are better than the official image on just one of the two perspectives. We do not observe any Mongo image that is better than its corresponding official image in both perspectives.
contains 29 security vulnerabilities. Although these better images on both perspectives exist, they represent only a median of 28% of the DockerHub community images; which make them hard to identify among the very large number of images.

While few images are better than the official images in both security and resource efficiency, there are other images that are better than the official image on just one of the two perspectives as shown in Figure 4.10. A median of 39% of the images are more resource efficient than their corresponding official image (RQ2), while a median of 83% of the images have less security vulnerabilities than their corresponding official image (RQ3). That is even if Docker regularly updates the security fixes and provides a feature that measures such security aspect of its official images. However, there are also community images which are less resource efficient and contain more security vulnerabilities. For example, “dawidnowak/spark-cassandra” image is 76% resource efficient and contains 76 security vulnerabilities, while its corresponding official image is 98% resource efficient and contains 47 security vulnerabilities as shown in Figure 4.10. From the other side, community developers do not document the efficiency and security perspectives of their images, which make them harder to identify.

Table 4.7: Spearman’s rank correlations between different metrics of the images.

<table>
<thead>
<tr>
<th>Image type</th>
<th># of libraries vs # of security vulnerabilities correlation coefficient ($r_s$)</th>
<th>Resource efficiency vs # of security vulnerabilities correlation coefficient ($r_s$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>-0.147</td>
<td>0.42</td>
</tr>
<tr>
<td>Java</td>
<td>0.490</td>
<td>0.23</td>
</tr>
<tr>
<td>Mongo</td>
<td>0.184</td>
<td>0.23</td>
</tr>
<tr>
<td>MySQL</td>
<td>0.100</td>
<td>0.17</td>
</tr>
<tr>
<td>Nginx</td>
<td>0.011</td>
<td>0.10</td>
</tr>
</tbody>
</table>
Note that the number of security vulnerabilities of an image is neither associated with its number of installed libraries nor its resource efficiency. While we observe a positive weak correlation between the number of libraries and number of security vulnerabilities for Java images, we do not observe any correlation for the remaining four image types as shown in Table 4.7. For example, the image “4gekkman/docker-nginx_phpffpm-dev” has a large number of 559 libraries, but its libraries are free from any security vulnerability. From the other side, the image “i386/nginx” has just 108 libraries, but it contains 29 security vulnerabilities. Similarly, we observe a positive and weak correlation between security vulnerabilities and resource efficiency for just Cassandra images as shown in Table 4.7.

**The best images in terms of resource efficiency and security vulnerabilities are not explored by users.** While resource efficient images and images with less security vulnerabilities compared to the official image exist, users neither download nor star these images. We do not observe any significant correlations (Spearman’s correlation coefficient) neither between the resource efficiency and the number of downloads and stars nor between the number of security vulnerabilities and number of downloads and stars as shown in Table 4.8.

**Summary**

Our results suggest to Docker tooling creators and DockerHub should provide approaches to distinguish images based on their installed libraries, resources, resource efficiency, and security vulnerabilities.
Table 4.8: Spearman’s rank correlation coefficients between the resource efficiency, number of security vulnerabilities, and the popularity metrics.

<table>
<thead>
<tr>
<th>Image type</th>
<th>Popularity metric</th>
<th>Resource efficiency</th>
<th># of security vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cassandra</td>
<td>#downloads</td>
<td>0.026</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td>#stars</td>
<td>-0.108</td>
<td>0.045</td>
</tr>
<tr>
<td>Java</td>
<td>#downloads</td>
<td>0.069</td>
<td>0.046</td>
</tr>
<tr>
<td></td>
<td>#stars</td>
<td>-0.095</td>
<td>-0.226</td>
</tr>
<tr>
<td>Mongo</td>
<td>#downloads</td>
<td>-0.049</td>
<td>-0.040</td>
</tr>
<tr>
<td></td>
<td>#stars</td>
<td>-0.079</td>
<td>0.007</td>
</tr>
<tr>
<td>MySQL</td>
<td>#downloads</td>
<td>0.073</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>#stars</td>
<td>-0.172</td>
<td>-0.156</td>
</tr>
<tr>
<td>Nginx</td>
<td>#downloads</td>
<td>-0.231</td>
<td>-0.228</td>
</tr>
<tr>
<td></td>
<td>#stars</td>
<td>-0.317</td>
<td>-0.130</td>
</tr>
</tbody>
</table>

4.6 Threats to Validity

4.6.1 External Validity

We perform our experiments on the 505 images that install five different popular software systems, which may threaten the generalizability of the results of our study. We choose five popular software systems and select their community images randomly to mitigate any bias that can affect the generalizability of our observations.

The results are only applicable on DockerHub images since we only study images from DockerHub. The results may vary in other Docker image registry systems such as Quay (CoreOS (2019)) or Google Container Registry (Google (2019b)). Therefore, additional replication studies can be performed to identify the differences among the images on other Docker image registry systems.
4.6.2 Internal Validity

A first threat to internal validity for threat concerns the collection of installed libraries. We only consider in this study the images that use the APT, APK, or RPM package manager to install their required libraries. However, we believe that our study can be replicated for manually installed libraries, by using the appropriate tools to extract the libraries that are installed manually.

A second threat to internal validity concerns the evolution of the number of DockerHub images (PQ2). While DockerHub API does not provide the creation date of an image, rather DockerHub provides the last update date for each tag (version) of an image. To mitigate this problem, we consider the last update date of the oldest tag for each image as the creation date of that image.

4.7 Chapter Summary

In this chapter, we study the differences among DockerHub images. We observe in our preliminary study that images from DockerHub can save a considerable amount of efforts when building, testing, and maintaining a Docker image. However, we observe that a large number of Docker image choices exist for any given software system, which makes choosing the most suitable image a nontrivial endeavor. Therefore, we empirically compare DockerHub images for the same software system in terms of their installed libraries, sizes, resource efficiency, and security vulnerabilities. We observe that users have a variety of choices for each software system in terms of installed libraries, while these choices vary in terms of resource efficiency and security. In fact, a median of 39%, and 83% of the community images are more resource efficient and
have less security vulnerabilities than their corresponding official image respectively. We also observe that developers do not document the particularities of their images regarding the resource efficiency and security; and users neither select the most resource efficient images nor images with less security vulnerabilities. Hence, we recommend that Docker tooling creators and DockerHub provide mechanisms for users to distinguish among images and select the most suitable images for their needs. We suggest future work to investigate other Docker image registry systems as well as comparing DockerHub images along additional perspectives.
Conclusions and Future Work

DOCKER is an emerging technology that is making a strong impact on how applications have traditionally been deployed. Docker also introduced Docker Compose to reduce the complexity and hassle of composing and managing multi-component applications. However, prior research did not explore the state of practice of composing multi-component applications using Docker Compose. Therefore, in this thesis, we hypothesize that a deep empirically sound understanding of how multi-component applications are created in the wild is essential for shaping future research efforts and tooling support given the rapid growth and dominance of containerization as a crucial mechanism nowadays for creating complex software applications. Through an analysis of 4,103 open-source Github projects that use Docker
Compose, we observe that 26.8% of the applications are not even developing any multi-component applications using Docker Compose. Besides, applications mostly use the basic options of Docker Compose, while they rarely use advanced options such as security or monitoring related options. Applications hardly upgrade their version of Docker Compose, even when they do so, few of them are downgraded due to platform and option compatibility issues. Moreover, many (21.5%) applications are still using the version 1 of Docker Compose which will be deprecated soon.

Furthermore, in our first study, we observe that more than 95% of the used online registry images are from DockerHub. With a preliminary study, we observe that many images are available on DockerHub for each software system. Through our empirical analysis of 505 images that install five different software systems, we observe that 96% of the community images are different from each other, while their installed libraries are also different from their official image. We also observe that the resource efficiency of these images vary, and a median of 38% of the images are more resource efficient than their official image. Besides, a median of 83% of the community images contain less security vulnerabilities than their official image.

5.1 Future Work

1. We observe that multi-component applications do not benefit from all the features and options of Docker Compose. Therefore, we suggest future work to investigate the usability of the existing options and how to help practitioners better benefit from the advanced capabilities of Docker Compose.
2. Our study focuses on composition of multi-components applications using Docker Compose without considering the platforms on which they run. We suggest future work to investigate on the integration of Docker Compose with Docker Swarm and Kubernetes.

3. We observed that DockerHub images are different from each other, while their documentation does not help understand the features of an image, neither how they are different from each other, we suggest future work to propose approaches to guide practitioner in locating the appropriate image.


