Studying Dependency Updates and a Framework for Multi-Versioning in Docker Containers

by

Sara Gholami Ghasem Abad

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering
University of Alberta

Abstract

Containerized software systems are becoming more popular and complex as they are one of the essential techniques that enable cloud computing. One of the enabling technologies for containerized software systems is the `Docker` framework. `Docker` is an open-source framework for deploying containers, lightweight, standalone, and executable units of software with all their dependencies (packages and libraries) that can run on any computing environment. `Docker` images facilitate deploying and upgrading systems as all of the dependencies required for a software package are included in an image. However, there exist several risks with running `Docker` images in production environments. One risky situation can occur when upgrading images, as an upgrade may result in many changing packages or libraries at once.

Therefore, in this thesis, we study the `Docker` images and analyze them to identify the risks of package changes. Also, we propose our solution, `DockerMV`, to mitigate this risk by running multiple versions of an image at the same time.

In this first part of this thesis, we analyze the official `Docker` image repositories that are available on `Docker Hub`, `Docker`'s public registry that holds `Docker` images. For each image in these repositories, we extract details about its native, *Node*, and *Python* packages. Afterward, we investigate which types of applications have more package changes in their image upgrades. We find that, depending on the type of applications, the package changes have different trends. For example, *Operating systems* and *Base Images* repositories have a lower median number of changes. However, *Analytics* and *Application Services* repositories have the highest median number of package changes. Our findings show that practitioners should be extra cautious when doing in-place upgrades of images of such applications in their production environments.

In the second part of this thesis, we provide a solution for mitigating this risk by applying software multi-versioning to `Docker` images. We present `DockerMV`, an open-source extension of the `Docker` framework that supports multi-versioning for containerized software systems. We demonstrate the usefulness of `DockerMV` from the performance point of view and test it on two open-source subject systems. In particular, we demonstrate how `DockerMV` can be used to balance the workload between `Docker` images that contain different versions of the same application. In both experiments, `DockerMV` maintained the system's performance while using a limited set of resources.

Preface

The research of this thesis has been conducted in the Analytics of Software, GAmes, and Repository Data (ASGAARD) lab led by Dr. Cor-Paul Bezemer and the Performant and Available Computing Systems (PACS) Lab led by Dr. Hamzeh Khazaei.

Chapter 2 has been submitted as S. Gholami, H. Khazaei, and C.P. Bezemer. Should you Upgrade Official Docker Hub Images in Production Environments? *IEEE Software.* I was responsible for data collection and analysis, as well as the manuscript composition. Dr. Bezemer and Dr. Khazaei were the supervisory authors and were involved with concept formation and manuscript composition.

Chapter 3 is published as S. Gholami, A. Goli, C.P. Bezemer, and H. Khazaei, 2020, April. A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning. *In Proceedings of the ACM/SPEC International Conference on Performance Engineering* (pp. 150-160) [26]. I was responsible for the development of the framework, conduction of both sets of experiments and the manuscript composition. A. Goli assisted with the `Znn` application experiments and contributed to manuscript edits. Dr. Bezemer and Dr. Khazaei were the supervisory authors and were involved with concept formation and manuscript composition.

Acknowledgements

I would like to thank all the people who contributed in some way to the work described in this thesis. First and foremost, I would like to express my sincere gratitude to Dr. Cor-Paul Bezemer for his overarching advice, encouragement, and support throughout my study. Your guidance always puts things in perspective, and I am deeply indebted to you.

I am also extremely grateful to Dr. Hamzeh Khazaei for his valuable advice and support. I would like to thank my thesis examiners, Dr. Marek Reformat and Dr. James Miller, to accept being part of my thesis examiners.

Thanks to all of my friends in both the ASGAARD and PACS labs who helped me in the last two years with their kind supports. I would like to especially thank Alireza Goli, one of my labmates in the PACS lab, who helped me conduct some experiments.

To my family and friends who have supported me on my journey through my studies, you have my sincerest thanks. To my mother, Fariba, and my brother, Mahdi, thank you for everything you have done to support me in my life. I would like to thank my partner, Armin, for his unparalleled patience, care, and support.

Table of Contents

1 Introduction and Background **1**

1.1 Introduction 1

1.2 Containerized Software Systems 3

1.3 Docker 4

1.3.1 Docker Container 5

1.3.2 Docker Architecture 5

1.3.3 Docker Hub 6

1.4 Outline 7

2 Should You Upgrade Official Docker Hub Images in Production Environments? **8**

2.1 Abstract 8

2.2 Introduction 9

2.3 Background 10

2.3.1 Docker 10

2.3.2 Docker Hub 11

2.4 Methodology 13

2.4.1 Selecting Repositories 13

2.4.2 Collecting Image Tags 15

2.4.3 Collecting Packages and Latest Update Dates 15

2.4.4 Identifying Package Changes 15

2.5 Results 17

2.6 Related Work 20

2.6.1 Security Analysis of Package Changes in `Docker` 20

2.6.2 Package Updates 21

2.7 Threats to Validity 23

2.7.1 Internal Validity 23

2.7.2 External Validity 23

2.8 Conclusion 23

2.9 Acknowledgements 24

3 A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning 25

3.1 Abstract 25

3.2 Introduction 26

3.3 Background 28

3.3.1 Containerized Software Systems 29

3.3.2 Managing Containers 30

3.4 A Motivating Example 31

3.5 Our Approach 32

3.5.1 Implementation 32

3.5.2 Load Balancing 34

3.5.3 On the Necessity of Our Approach 36

3.6 Experimental Setup 37

3.6.1 Subject Systems 37

3.6.2 Introducing Multi-Versioning in the Subject Systems 38

3.6.3 Experiments 38

3.6.4 Workload 40

3.6.5 Deployment of the Subject Systems 41

3.6.6 Load Balancing 42

3.7 Experimental Evaluation 43

3.7.1 Experiments with the `TeaStore` Application 44

3.7.2 Experiments with the `Znn` Application 44

3.8 Related Work 46

3.8.1 Software Multi-Versioning 46

3.8.2 Software Multi-Versioning for Containerized Systems 46

3.8.3 Performance Engineering of Containerized Systems 47

3.9 Threats to Validity 48

3.9.1 External Validity 48

3.9.2 Internal Validity 48

3.9.3 Construct Validity 49

3.10 Conclusion 50

4 Conclusions & Future Work **52**

4.1 Conclusions 52

4.2 Future Work 53

Bibliography **54**

List of Tables

2.1 `Docker Hub`'s repository types as of April 2020 13

2.2 `Docker Hub` official repositories categories 17

2.3 The packages with the most often changes 22

3.1 A description of the experiments that we conducted for the `TeaStore` and `Znn` applications 40

3.2 Description of the containers in the experiments 42

3.3 Description of the virtual machines 42

List of Figures

1.1 Docker search trend from 2013 to 2020 4

1.2 Container creation process for the MongoDB application [42] 5

1.3 Docker architecture 6

2.1 Docker Hub consists of collections of images that can be pulled and used to create and run Docker containers. 11

2.2 Overview of our data collection process for our study on package changes in Docker Hub repositories 13

2.3 Pull counts and star counts of the official and community repositories. The red dot indicates the median value. 14

2.4 An example of branches in a repository over time 16

2.5 Median number of upgrades in each category 19

2.6 Median number of downgrades in each category 20

2.7 Median number packages in each category 21

3.1 High-level architecture of the TeaStore application with multi-versioning of the Recommender microservice. 28

3.2 High-level architecture of a regular service in Docker where requests are load balanced in a Round Robin manner. 29

3.3 High-level architecture of a service with multi-versioning where requests are balanced based on a rule-set. 36

3.4 High-level architecture of the Znn application 37

3.5 Containerized deployment of Znn in which we have two different versions of the Media service. 39
3.6 Shape of the Znn application workload 41
3.7 The TeaStore application experiments and the distribution ratio of traffic using software multi-versioning and adaptive load balancing . . 44
3.8 The Znn application experiments using only the multimedia vs only the text version of the service. Note that the scales on the y-axes are different. 45
3.9 The Znn application experiment using software multi-versioning and adaptive load balancing 45

Abbreviations

API	Application Programming Interfaces.
DockerMV	Docker with Multi-Versioning.
OCI	Open Container Initiative.
OS	Operating System.
REST	REpresentational State Transfer.
VM	Virtual Machine.

Chapter 1

Introduction and Background

1.1 Introduction

Containerization is a popular technology that allows developers to create and deploy applications in a faster way. Containerization involves packaging software code and all of its dependencies to run on any computing environment [33]. One of the technologies that enable containerization is the `Docker` framework, which facilitates the development, deployment, and shipment of the containerized software systems. `Docker` containers are running instances of `Docker` images [17], which are composed a software application and its libraries and packages. As a result, `Docker` images make developing and upgrading the software application easier, as all of the required packages come with the source code. However, for the same reason, `Docker` images bring additional threats to the system. In each upgrade, many packages could change together, which could result in poor performance and security [57, 64, 65].

There have been studies on the risks that are introduced to the client's systems by upgrading their packages [37, 44, 55, 62]. For instance, a study on the available packages in the *Maven* repository showed that one-third of major package upgrades and one-third of minor package upgrades contain breaking changes [55]. Major upgrades could have incompatible Application Programming Interface (API) changes and minor upgrades often add backward compatible functionality [52]. Similarly, a study on the *Node.js* packages showed that there are breaking changes in the minor and patch upgrades [44] (e.g., patch changes have backward compatible bug fixes [52]). These changes in packages can lead to poor performance or security vulnerabilities in the client's system [37]. In addition, some aspects of packages such as performance are often not tested very well [8, 41]. As a result, unexpected package changes after a `Docker` image upgrade could result in a reduced software experience, e.g., in terms of performance and security.

In this thesis, we focus on the `Docker` framework and identify the application types that introduce more threats to their clients as a result of package changes. In addition, we propose our solution to mitigate the identified threats using software multi-versioning. We conducted the two following studies:

Research Study 1: Should You Upgrade Official Docker Hub Images in Production Environments?

In this study, we focus on the official `Docker` repositories on `Docker Hub` and analyze the images in these repositories for changes to their native (operating system), *Node*, and *Python* packages. In this work, we quantify how many packages are changing in each type of `Docker Hub` repository.

Research Study 2: A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning

One approach to prevent having poor performance in a system after upgrading its packages is to keep the previous version along with the upgraded version and gradually switch to the later version. Therefore, in our second study, we focused on extending the `Docker` framework with multi-versioning (through our proposed `DockerMV` framework), which facilitates having different versions of an image under one service. In this study, we focused on the performance point of view and showed how multi-versioning could help to satisfy the system's performance requirements while maintaining the quality of the service at an acceptable level with a limited set of resources.

The results of our first study could help to understand how package changes are happening in different types of applications, which can help developers of containerized software systems in decision making about upgrading the container images in a production environment. Also, developers can use `DockerMV` to facilitate the deployment of their systems with a customizable load balancer.

1.2 Containerized Software Systems

Containerization is a major trend in software development. Containerization is the encapsulation of software code and all of its dependencies so that it can run on any computing environment or infrastructure [33]. The following are some of the popular container technologies:

- **Docker containers** are one of the most popular open-source container technologies [32]. `Docker` containers do not require an Operating System (OS) per application as they share the host OS kernel, which makes them lightweight [17].
- **Unikernels** optimize the resources required by a container at runtime. Unikernels can boot and run on their own without requiring a host OS or external libraries [60].
- **LXD** is a container platform that is build and operated using the same tools as VMs. However, LXD containers can achieve similar runtime performance as other containers and improved utilization in comparison to VMs [60].
- **OpenVZ** is a container platform for running a complete operating system. OpenVZ requires both host and guest OS to be running *Linux* as it shares the host OS kernel. OpenVZ is faster and more efficient than traditional VMs [60].
- **Rkt** (pronounced Rocket) containers came from CoreOS to address security vulnerabilities in early versions of `Docker` [60].
- **Windows Server Containers** are the same as *Linux* containers for Microsoft

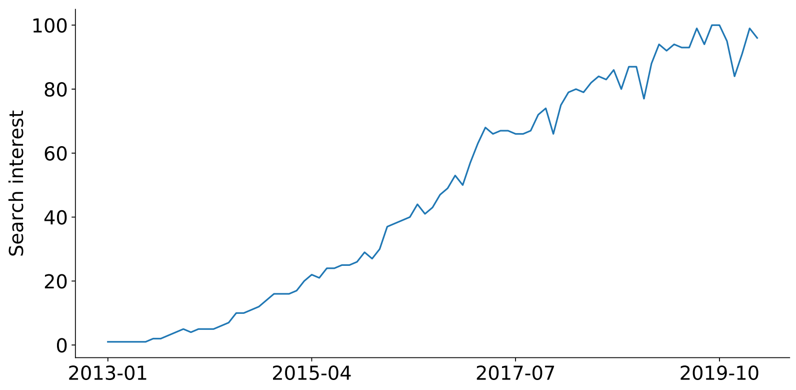


Figure 1.1: `Docker` search trend from 2013 to 2020

workloads. Microsoft enabled container technology on core windows OS. However, these containers are only available on Windows 10, Server 2016, and Azure [60].

- **Hyper-V Containers** can bring a higher degree of security by hosting Windows Server Containers. Hyper-V containers can be used when higher security and isolation are required, but it would reduce the host's efficiency and density. Hyper-V containers are built and managed in a similar way to Windows Server Containers [60].

1.3 Docker

`Docker` is a framework for developing, deploying, and shipping applications, which separates the application from the infrastructure. `Docker` provides the ability to run applications in loosely isolated environments called containers [18]. `Docker` initially used to be a closed-source framework under the name dotCloud Inc., but in March 2013, it was released as an open-source project by Docker Inc [20]. Figure 1.1 shows `Docker`'s significant popularity rise based on the Google search trend from 2013 to 2020 worldwide, where a popularity value of 100 is the maximum popularity for the term, and the rest of the data is normalized based on that.

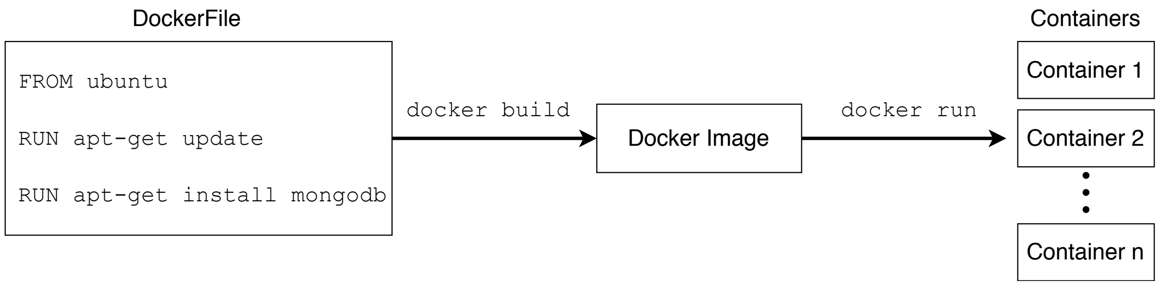


Figure 1.2: Container creation process for the MongoDB application [42]

1.3.1 Docker Container

A `Docker` container is a lightweight, standalone, and executable unit of software that packages software code and all of its dependencies (packages and libraries) to run in any computing environment [17]. Figure 1.2 shows the process of creating a container for the MongoDB application. First, a `DockerFile`, a text file containing all of the commands for creating a container [19], is used to build a `Docker` image. Then, a `Docker` container is created based on a `Docker` image, which means that `Docker` containers are running instances of `Docker` images.

1.3.2 Docker Architecture

`Docker` has two important components: the `Docker Engine`, the virtualization technology, and the `Docker Registry` which is a service for sharing `Docker` images. Figure 1.3 shows the `Docker` architecture and how these components interact.

`Docker Engine` is a client-server application with three major components: the `Docker` client, the `Docker` daemon, and the REpresentational State Transfer (REST) API. The `Docker` daemon listens for the `Docker` API requests and manages `Docker` objects such as images, containers, and volumes. Besides, `Docker` daemons can communicate with each other to manage `Docker` services. The `Docker` client is the main way in which most users interact with `Docker`, enabling them to run commands. As can be seen in Figure 1.3, the `Docker` client communicates with the `Docker` daemon through the REST API and `Docker` daemon communicate with the `Docker` registry

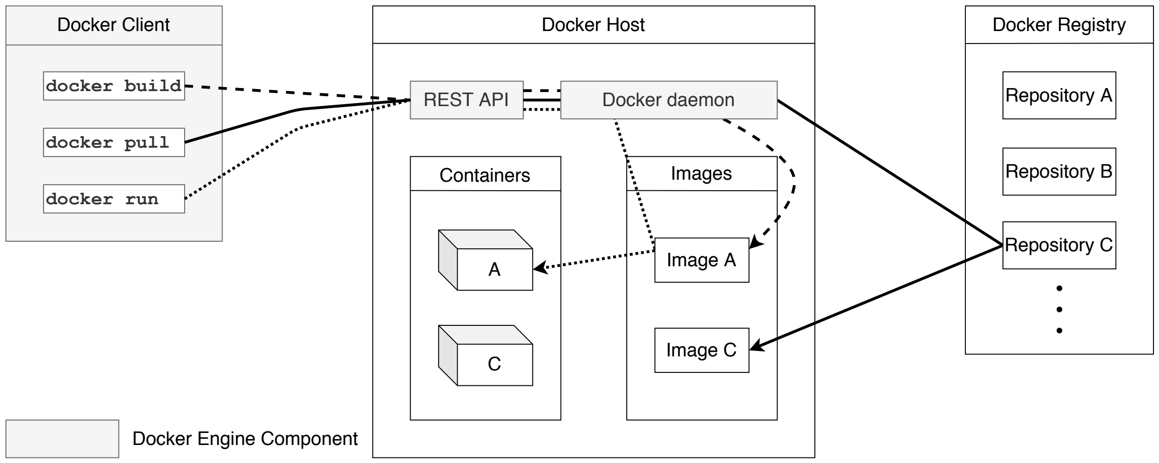


Figure 1.3: `Docker` architecture

or `Docker` objects to execute a command. For example, the `docker build` command is used to build an image, or the `docker run` command creates a container from an image. As a result, the `Docker` client and `Docker` daemon can run on the same machine or can connect remotely. Also, a `Docker` client can communicate with more than one `Docker` daemon.

A `Docker Registry` is where `Docker` images are stored. There are private and public registries where users can use either of them. `Docker Hub` is `Docker`'s default public registry, which is used by default when commands such as `docker push` (i.e., to upload an image to the registry) and `docker pull` (i.e., to download an image from the registry) are executed [18].

1.3.3 Docker Hub

As explained, `Docker Hub`[1] is a `Docker` registry, which is a centralized resource storing `Docker` images. `Docker Hub` contains over 3 million repositories. In addition to `Docker Hub`, there used to be `Docker Store` and `Docker Cloud`, which were other `Docker` registries to share or sell `Docker` images [16, 47]. However, since December 2018, `Docker Store` and `Docker Cloud` were merged into `Docker Hub` [45], and `Docker` Certified and Verified images were added to the `Docker Hub`. Currently, `Docker Hub`

[1]https://hub.docker.com/

has four types of repositories:

- *Community* repositories are created by the community users of `Docker Hub`, which means anyone with an email address can create a community repository.
- *Verified* repositories are developed and maintained by verified third-party software vendors.
- *Official* repositories are reviewed and published by a team sponsored by Docker Inc. These repositories provide base operating systems, programming languages, databases, and other application services.
- *Certified* repositories are a subset of verified repositories that had passed some additional `Docker` quality, best practices, and support requirements.

1.4 Outline

The rest of this thesis is as follows: Chapter 2 presents our study on `Docker Hub` images and the risks of upgrading these images. Chapter 3 contains the study on `DockerMV`, which is our proposed solution for maintaining the performance requirements of containerized software systems with multi-versioning. Finally, Chapter 4 concludes this thesis and highlights our findings and the potential future works of our study.

Chapter 2

Should You Upgrade Official Docker Hub Images in Production Environments?

2.1 Abstract

Containerized software systems are a crucial technology in cloud computing. With the growth of these systems, containerized software systems become more complicated and complex to manage. `Docker` is one of the most popular containerization technologies. `Docker` allows a user to deploy `Docker` images, software code that is packaged with the packages it depends on, to create and run containers. While `Docker` images facilitate the deployment and in-place upgrading of an application in a production environment by replacing its container with one based on a newer image, many internal dependencies could change at once during such an image upgrade, which can potentially be a source of risks.

In this chapter, we study the official `Docker` images on `Docker Hub`, `Docker`'s official image registry, and explore how internal packages are changing in these images. We analyze the native, *Node*, and *Python* packages in `Docker` images, and investigate which types of applications tend to have the most changes to their dependencies. Our findings can help developers, who want to do upgrades on `Docker` images in their systems, to make a more cautious decision regarding the unwanted changes that

could happen in their system.

2.2 Introduction

Containerization is a popular approach to deploy software systems [33]. One of the enabling technologies for containerization is `Docker`, an open-source framework to deploy containers in different computing environments [11]. `Docker` containers are composed of an image that encapsulates software code and all its required package and library dependencies [33]. As a result, deploying `Docker` containers into a production environment and applying in-place upgrades by replacing them with containers created from newer images is easy. However, with every upgrade of a `Docker` image, many packages could change at once, which could e.g., result in reduced performance or security of the application.

Several related works studied the `Docker` images available on `Docker Hub` from a security point of view [57, 64, 65]. For instance, Shu et al. [57] studied over 300,000 `Docker` images for the spread of vulnerabilities from one image to another image that uses it. They found that images inherit security vulnerabilities from their parent image. Similarly, Zerouali et al. [64] showed that vulnerabilities in *npm* packages might impact `Docker` images.

Besides the potential security risks in `Docker` images, there is another potential risk, which is the risk of changing many components of a system at once. The risk of package changes was studied in different environments and languages such as *Maven*, *Node.js*, and *Java* [37, 44, 55, 62]. These studies show that package changes can lead to broken functionality, poor performance, or security vulnerabilities in the applications that depend on the packages. In addition, other studies show that certain aspects of packages, such as performance, are often not well-tested [8, 41]. As a result, although applying in-place upgrades on `Docker` images is easy, it can put the whole system at risk through issues that are caused by internal packages.

In this chapter, we study the package changes in official `Docker` images (images reviewed by the `Docker` team) in the `Docker Hub` registry. We focus on the native (operating system), *Node*, and *Python* packages and investigate which types of applications tend to have more package changes.

Our study shows how frequent changes are happening in the packages used by different types of applications on `Docker Hub`. Our study helps to raise awareness with developers of containerized software systems of the necessity of being cautious when upgrading `Docker` images in a production environment.

The rest of the chapter is organized as follows. Section 2.3 provides background information about `Docker` and `Docker Hub`. Section 2.4 describes our methodology. Section 2.5 presents our findings. Section 2.6 provides an overview of the related work. Section 2.7 explains the threats to the validity of our work, and Section 2.8 concludes our chapter.

2.3 Background

In this section, we provide background information about `Docker` and `Docker Hub`.

2.3.1 Docker

`Docker`[1] started as an open-source project in 2013 as a Platform-as-a-Service company [43]. `Docker` is a container virtualization technology [4] that puts together several kernel-level technologies such as `LXC` and `Cgroups` to facilitate the deployment and use of containers. `Docker` provides interfaces to create and deploy containers. `Docker` containers are lightweight, packaged applications that can run on different computing environments without modification. `Docker` relies on two major components: the `Docker Engine`, which is the virtualization technology, and `Docker Hub`, a service for sharing `Docker` images [11].

`Docker` containers are created from `Docker` images by executing the `docker run` command. `Docker Hub` is where `Docker` images are stored by default, which means

[1]https://www.docker.com/

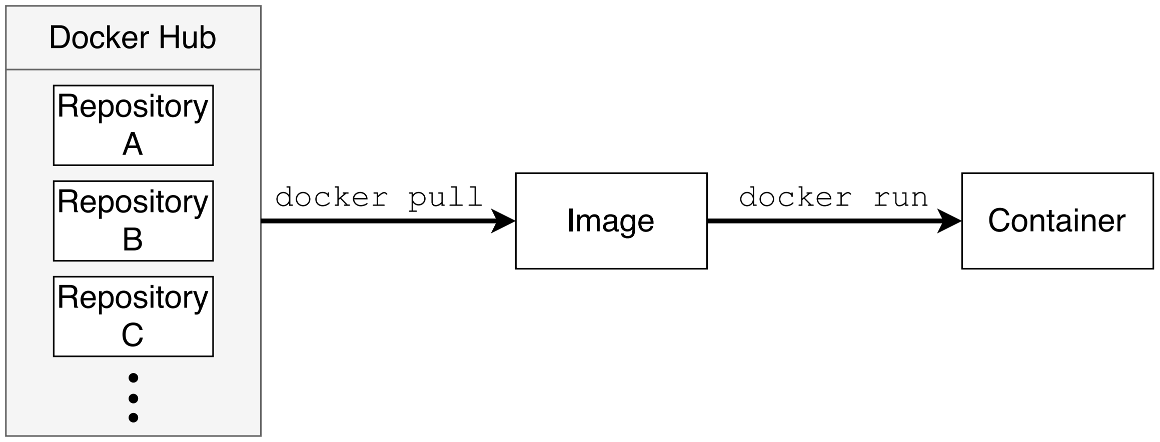


Figure 2.1: `Docker Hub` consists of collections of images that can be pulled and used to create and run `Docker` containers.

that by default `docker push` uploads an image to `Docker Hub` and `docker pull` downloads an image from `Docker Hub`. Figure 2.1 shows how images in `Docker Hub` are used for creating containers. First, an image is pulled from `Docker Hub` by executing the `docker pull` command, and then by executing the `docker run` command the image is used to create a container. Each image is created based on a `DockerFile`, a text file consisting of a series of commands to create an image. Listing 2.1 shows an example of a `DockerFile`. In the first line, *Ubuntu* is used as the base image, meaning that the packages required for *Ubuntu* are added to the final image. Then lines 3 and 4-6 install *Python* and *Node.js*, which means that packages required for these languages will be added to the final image. In general, the used packages in an image are either packages that are native to a Linux distribution such as *debian*, *arch*, or *alpine*, or packages that are installed by popular package managers such as *PyPy*, *npm*, or *CRAN* [64].

2.3.2 Docker Hub

As explained, `Docker Hub`[2] is `Docker`'s default registry for finding and sharing container images. `Docker Hub` is a collection of repositories. Currently, there are over 3 million repositories in `Docker Hub`. Each repository is a collection of images, which

[2]https://hub.docker.com/

Listing 2.1: Example `DockerFile`

```
FROM ubuntu

RUN apt-get install -y software-properties-common python
RUN add-apt-repository ppa:chris-lea/node.js
RUN apt-get update
RUN apt-get install -y nodejs
RUN mkdir /var/www
ADD app.js /var/www/app.js

CMD ["/usr/bin/node", "/var/www/app.js"]
```

allows users to share container images with other users, such as their team members or customers. Images in a repository are identified by unique user-identified tags. The following four types of repositories are available on `Docker Hub`. Table 2.1 gives an overview of the frequency with which these types occur at the time of writing.

1. *Community* repositories are maintained and delivered by community developers, including all users with a `Docker Hub` account. As a result, there is no guarantee on security, maintainability, or following best practices for development in these repositories. More than 99% of the `Docker Hub` repositories are community repositories.

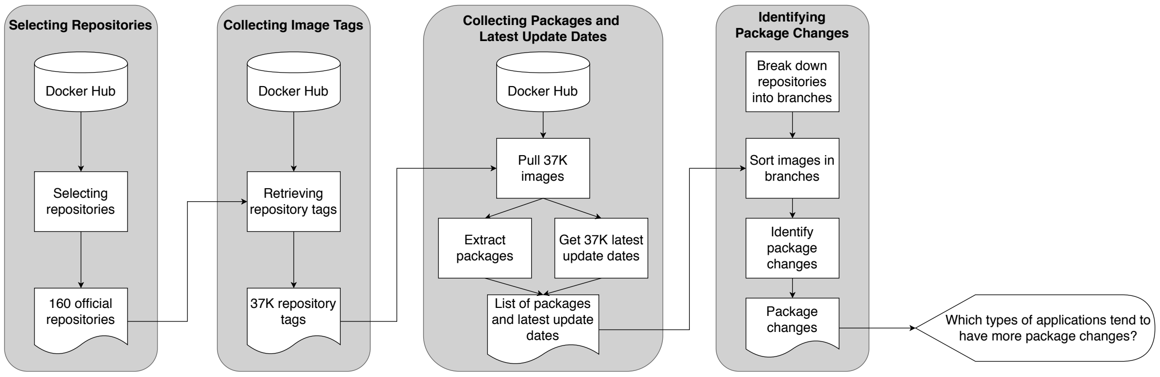
2. *Verified* repositories are published and maintained by verified third-party publishers such as IBM or Microsoft [45]. There are 339 verified repositories on `Docker Hub`.

3. *Official* repositories are reviewed and published by a team that is sponsored by `Docker Inc`. `Docker` community members can contribute to developing the official images. Images in the official repositories exemplify `DockerFile` best practices and ensure that security updates are applied in a timely manner. As can be seen from Table 2.1, there are 160 official repositories.

4. *Certified* repositories are a special type of verified repository that are built

Table 2.1: `Docker Hub`'s repository types as of April 2020

Repository type	# of repositories	Proportion
Community	3,354,643	99.9%
Verified	339	< 1%
Official	160	< 1%
Certified	51	< 1%



Selecting Repositories
Docker Hub
Selecting repositories
160 official repositories
Collecting Image Tags
Docker Hub
Retrieving repository tags
37K repository tags
Collecting Packages and Latest Update Dates
Docker Hub
Pull 37K images
Extract packages
Get 37K latest update dates
List of packages and latest update dates
Identifying Package Changes
Break down repositories into branches
Sort images in branches
Identify package changes
Package changes
Which types of applications tend to have more package changes?

Figure 2.2: Overview of our data collection process for our study on package changes in `Docker Hub` repositories

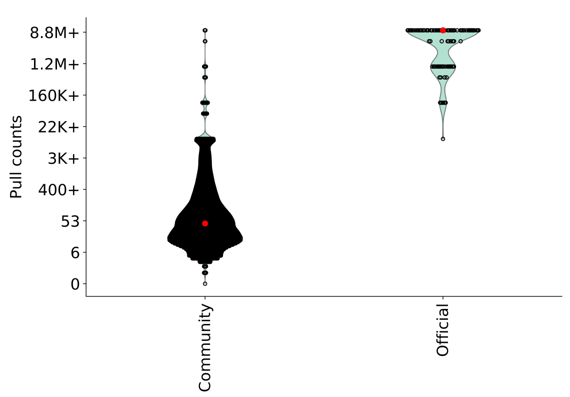
following best practices, tested and validated against the `Docker` Enterprise Edition platform and APIs, passed security requirements and are collaboratively supported [45]. There exist 51 certified repositories on `Docker Hub`.

2.4 Methodology

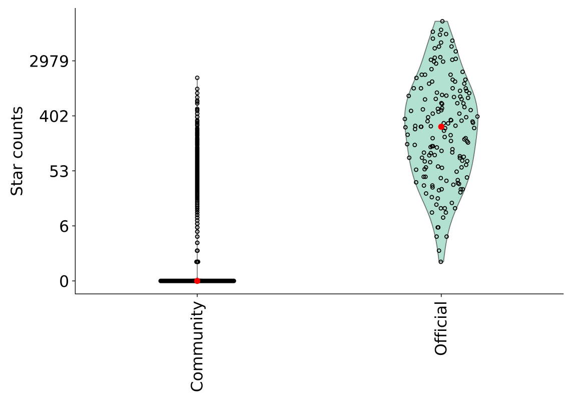
In this section, we present our methodology for studying package changes in official `Docker Hub` images. Figure 2.2 displays the steps of our methodology: selecting repositories, collecting image tags, collecting packages and latest update dates, and identifying package changes. We detail each step below.

2.4.1 Selecting Repositories

In the first step, we select a set of repositories to study. We selected the `Docker Hub` official repositories as these 160 repositories are considerably more popular than the



(a) Number of pull counts in official and community repositories



(b) Number of star counts in official and community repositories

Figure 2.3: Pull counts and star counts of the official and community repositories. The red dot indicates the median value.

other types of repositories. Figure 2.3a shows the number of pull counts for official and community repositories. `Docker Hub` does not provide this information for verified and certified repositories. As can be seen, the median number of pull counts for the official repositories is 10 million, while the median number of pull counts for the community repositories is 45 and varies from 1 pull to 10 million.

Similarly, Figure 2.3b presents the number of star counts for both official and community repositories, where the median number of star counts is 271 for the official repositories and 0 for the community repositories. Both of these figures are evidence of the popularity of the official repositories. In addition, as `Docker Inc.` sponsors a team to verify and publish the official repositories' content, users can be more confident about the credibility of the images in the official repositories. Therefore, we focused our study on the official repositories.

In our study, we did not consider the *Scratch* and *OpenSUSE* repositories as they did not contain images. Hence, we study 158 repositories in total.

2.4.2 Collecting Image Tags

In the second step, we get the list of available tags for each repository. We used the code available on the Source{d} GitHub page[3] with a few modifications to retrieve all available tags. We collected a list of almost 37K tags from all the official repositories.

2.4.3 Collecting Packages and Latest Update Dates

In our third step, we download images one by one and analyze their packages. We focused on the native, *Node*, and *Python* packages in this study. Native packages are packages used by the operating system, which is *Linux*. *Node* and *Python* packages are the packages installed for *Node.js* and *Python* applications. In our study, all repositories contained native packages. In addition, six repositories used *Node* packages, six repositories used *Python* packages, and two repositories used both *Node* and *Python* packages. Similar to the previous step, we used the Source{d} code to analyze the available packages in each image. In addition to the list of packages, we collected the latest update date for each image by executing the `docker inspect` command.

2.4.4 Identifying Package Changes

In the last step, we study the package changes when upgrading an image. We split the images in each repository into versioning branches, as images in different branches might use different packages, and comparing them would not be insightful. Figure 2.4 shows an example timeline of how branches could evolve in a repository. In this example repository, there is a branch with the *alpine* ending, which indicates those images are using *alpine*, a *Linux* distribution. As the branches may progress independently, we should not compare images from different branches. For example, we did not compare *1.1-alpine* to *1.1.1* as they are from different branches. After identifying the branches, we sorted the images in each branch based on their latest update date.

[3]https://github.com/src-d/datasets

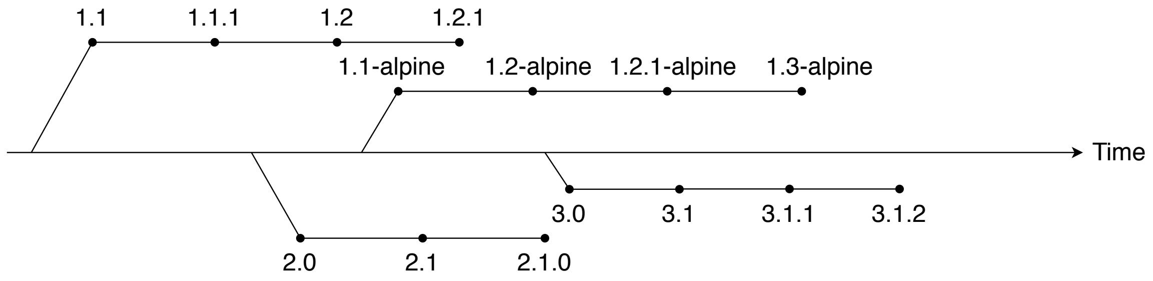


Figure 2.4: An example of branches in a repository over time

However, there were cases in which several images were updated on the same day. In these cases, we manually sorted the images based on the versioning specified in the tags. As tag names do not follow any naming convention, we could not automate this process. Some repositories used the release date as their version number (e.g., *20200415*), and some used semantic versioning to indicate major, minor, and patch releases (e.g., *1.13.2*).

In addition, the tags in *TomEE*, *NeuroDebian*, *ROS*, *BuildPack-Deps*, and *AdoptOpenJDK* repositories were not clear to divide into branches. For example, in the *ROS* repository, all of the tags are names, such as *lunar-perception-stretch*, *melodic-perception-stretch*, and *melodic-perception*. Therefore, we excluded these five repositories from our analysis.

Finally, we compared the packages used in each image with its adjacent image in the same branch to identify any major, minor, or patch upgrades or downgrades. To determine if a change is upgrade or downgrade, we compared the group of numbers and characters in the package versioning. For instance, a version change from *1.3.0* to *2.0* or from *1.3-a* to *1.3-b* are upgrades. Also, a change from version *2.1.0* to *2.0* or from *3.3-b* to *3.3-a* are downgrades. In addition, we needed to determine if a change is major, minor, or patch. Based on the semantic versioning definition [53], major changes make incompatible API changes, minor changes add functionality in a backwards compatible manner, and patch changes make backwards compatible bug fixes. To identify each type of change, we separated the numbers in the version tags.

Table 2.2: `Docker Hub` official repositories categories

Category	Example repositories	# of repositories
Analytics	Telegraf, Logstash	4
Application Frameworks	Mongo-express, Drupal	21
Application Infrastructure	Nginx, HTTPd	14
Application Services	Ghost, Elasticsearch	25
Base Images	Ubuntu, Alpine	16
Databases	Redis, MySQL	15
DevOps Tools	Consul, SonarQube	8
Featured Images	AmazonLinux, Registry	5
Messaging Services	NATS, LightStreamer	4
Monitoring	Kapacitor	1
Operating Systems	CentOS, Debian	15
Programming Languages	Python, Golang	20
Storage	Couchbase, Memcached	4

If the first set of digits were different, then the change is major. If the second set of digits were different, then it is a minor change. Otherwise, it is a patch change. For example, version *1.2.0* to *2.0* is a major change, while a change from version *1.2.0* to *1.3.1* is a minor change, and a version change from *1.2.0* to *1.2.1* is a patch change.

Table 2.2 shows the categories of the official repositories on `Docker Hub` with two example repositories and the number of repositories in each category. From the 153 official repositories, 115 belong to one or two of these categories. We categorized 38 official repositories that did not belong to any category as *Others*. We used these categories to compare the package changes in repositories of different categories.

2.5 Results

This section presents the results of our study on which types of applications tend to have more package changes in official `Docker` images. As `Docker` images facilitate deployment and upgrading of an application system in a production environment, it is important to study whether package changes are more likely across different types of applications. Therefore, practitioners can use this information to make more careful decisions regarding upgrading an image in a production environment.

There is a median of 8.6 upgrades per image across official `Docker` images. Figure 2.5 displays the distribution of major, minor, and patch upgrades in images of different categories. The *Application Services* applications have a median number of 1.4, 2.2, and 11.1 major, minor, and patch upgrades, respectively, which are the highest medians across categories. More specifically, in the *Application Services* category, the *ZNC* application has the highest number of upgrades (6.2 major upgrades, 21.9 minor upgrades, and 79.2 patch upgrades). The *Analytics* category has the second-highest median number of major (0.6) and patch (8.1) upgrades, and the third-highest median number of minor (1.5) upgrades. Afterward, the *Programming Languages*, *Application Infrastructure*, and *Databases* categories have the next highest median number of patch upgrades.

There is a median of 2.1 downgrades per image across official `Docker` images. Figure 2.6 illustrates the distribution of major, minor, and patch downgrades per image across different categories. The *Analytics* applications with 0.4, 0.8, and 3.8 have the highest median number of major, minor, and patch downgrades. The *Application Infrastructure*, *Application Services*, and *Programming Languages* categories have the second-highest median number of package downgrades in major, minor, and patch changes, respectively.

Images of *Analytics* applications are the least stable. The official images specify a median of up to 36 third-party packages. Figure 2.7 shows the median number of packages per image specified in each category. As can be seen, the *Operating Systems* and *Base Images* categories have the lowest median number of packages, which is why these applications also have the lowest median number of package changes in both upgrades and downgrades. The images in the *Operating Systems* and *Base Images* applications tend not to add many additional packages and pro-

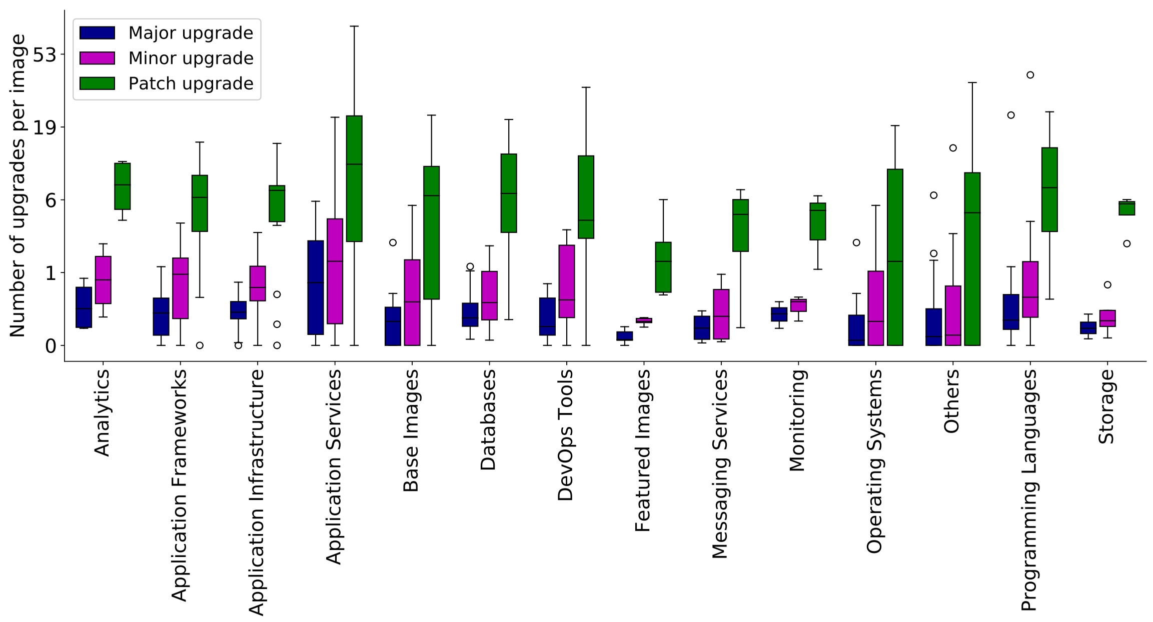


Figure 2.5: Median number of upgrades in each category

vide the base operating system in an image. Although the images in the *Application Services* category have one of the highest median numbers of packages changes, the median number of packages used in these applications is not the highest. In contrast, applications in the *Analytics* category have the highest median number of package changes and total packages used in the images. This finding suggests that images for the *Analytics* applications are less stable than images for other types of applications.

The packages that are changed the most often are common utility packages. There are over 9K different packages used in the official `Docker` images. Table 2.3 shows the top 10 packages with the most number of changes across `Docker` official repositories with a description of the package and the number of applications that used these packages. As can be seen, the ones with the highest number of changes are utility packages. In many cases, when upgrading a system, we do not want to upgrade several other utility packages unless absolutely necessary, as such upgrades might cause incompatibilities. In addition, newer versions of these utility packages may contain bugs. Therefore, practitioners need to carefully check the packages which are changing in an image upgrade and consider the consequences on their system.

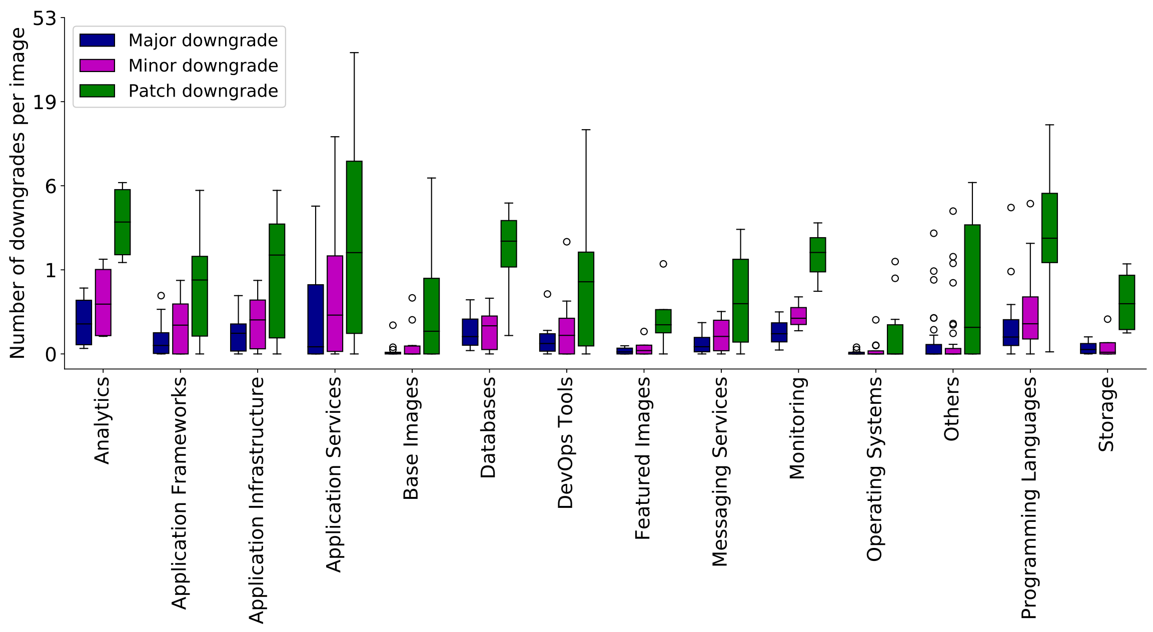


Figure 2.6: Median number of downgrades in each category

Summary: *Practitioners need to be cautious when doing in-place upgrades of images from the official* `Docker Hub` *repositories as in all studied applications, many packages are changing.*

2.6 Related Work

In this section, we discuss the related work to our study. More specifically, we discuss prior work on the security analysis of package changes in `Docker` images and package changes in different environments and languages.

2.6.1 Security Analysis of Package Changes in `Docker`

There have been several studies on package upgrades in `Docker` images. However, they all focus on the security aspect [57, 64, 65]. Shu et al. [57] developed a framework to discover, download, and analyze images for security vulnerabilities and the propagation of the vulnerabilities from parent images to their children. They analyzed more than 300,000 `Docker Hub` official and community images and found that there are more than 70 vulnerabilities in each image on average, where child images have

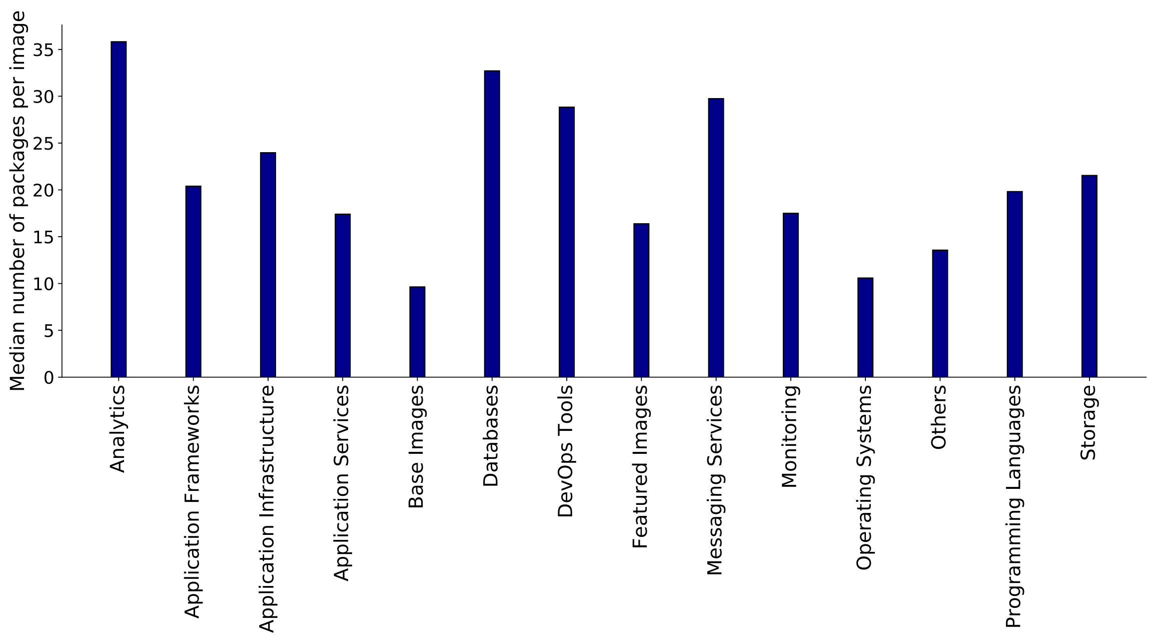


Figure 2.7: Median number packages in each category

20 more vulnerabilities. Zerouali et al. [64] empirically studied the use of *JavaScript* packages in `Docker` images. They analyzed 961 images from three official repositories, which used *Node.js* packages. They found that all of the official images which used *Node* packages have security vulnerabilities with an average of 16 vulnerabilities per image, suggesting that `Docker` deployers should keep their *JavaScript* packages up to date.

All the prior studies focused on the `Docker` images from the security point of view. In contrast, we study `Docker` images to identify package changes as a potential risk.

2.6.2 Package Updates

There have been studies on package dependencies in different environments and languages [37, 44, 55, 62]. Kerzazi et al. [37] studied botched releases in an application for 1.5 years. Botched releases are releases that cause abnormal behaviors such as poor performance, crashes, or hangs in the system after deployment into the production environment. Based on their study, about 22.5% of the releases are botched releases, which can significantly affect the systems that are using this application. As another

Table 2.3: The packages with the most often changes

Package name	Description	# of changes	# of applications that used them
tzdata	Time zone and daylight-saving time data	4096	106
base-files	Debian base system miscellaneous files	3794	104
libsystemd0	Provides interfaces to various systemd components	3446	87
libudev1	Provides access to udev device information	3437	88
openssl	Cryptography and SSL/TLS toolkit	3035	102
curl	Command line tool for transferring data with URL syntax	2633	75
libc-bin	Utility programs related to the GNU C Library	2560	97
gpgv	GNU's tool for secure communication and data storage	2435	99
libc6	GNU C Library	2315	85
apt	Command line package manager	2307	99

example, Raemaekers et al. [55] conducted a study on version changes of the jar files in *Maven* repository where about one-third of the major changes and one-third of the minor changes had at least one breaking change. Breaking changes are vital as they can have a significant impact on the client's software system and lead to compilation errors and crashes. In a study by Mezzetti et al. [44] on *Node.js* libraries, they found that 5% of the packages have been affected by breaking changes due to a minor or patch update in their dependencies. Xavier et al. [62] studied breaking changes in updates of 317 *Java* libraries, where 14.8% of changes caused incompatibilities with previous versions.

The previous studies investigated the effect of package changes in different environments and languages such as *Node.js*, *Maven*, and *Java*. In contrast, in our study, we analyze `Docker` images to identify package changes in official images.

2.7 Threats to Validity

In this section, we discuss the threats to the validity of this study.

2.7.1 Internal Validity

To sort the images in each repository, we first separated the images into possible branches. This process has been done manually as there is no concept of branch defined on `Docker Hub`. We did not include repositories in our study when we were not sure about the branches. Future studies should investigate automated approaches for identifying branches from version numbers.

2.7.2 External Validity

In this study, we analyzed the `Docker Hub` images for their native, *Node* and *Python* packages. Although we extracted the native packages for all of the images, only six repositories used *Node* packages, six repositories used *Python* packages, and two repositories use *Node* and *Python* packages. Therefore, future studies should analyze changes in `Docker` images for other types of packages (such as *R* packages that are managed by *CRAN*).

2.8 Conclusion

In this chapter, we studied the official `Docker Hub` repositories and analyzed over 37K images in these repositories for their native (operating system), *Node*, and *Python* packages. Our study shows that all studied applications have changing packages.

Although the *Operating Systems* applications did not have many package changes, it could be due to the fact that these types of applications do not use many third-party packages and provide the base operating system. Therefore, even a few changes in these applications are important as they are used as the base image in other images.

In addition, common utility packages are changing the most often among all the packages. In many cases, these packages are not essential to the main application, so one could wonder whether it is worth to risk breaking the system for.

In conclusion, we advise practitioners to take extra caution when doing in-place upgrades on `Docker` images as in all studied applications, several packages are changing.

2.9 Acknowledgements

Special thanks to Source{d} for providing valuable code for data collection and analysis.

Chapter 3

A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning

3.1 Abstract

With the increasing popularity and complexity of containerized software systems, satisfying the performance requirements of these systems becomes more challenging as well. While a common remedy to this problem is to increase the allocated amount of resources by scaling up or out, this remedy is not necessarily cost-effective and therefore often problematic for smaller companies.

In this chapter, we study an alternative, more cost-effective approach for satisfying the performance requirements of containerized software systems. In particular, we investigate how we can satisfy such requirements by applying software multi-versioning to the system's resource-heavy containers. We present `DockerMV`, an open source extension of the `Docker` framework, to support multi-versioning of containerized software systems. We demonstrate the efficacy of multi-versioning for satisfying the performance requirements of containerized software systems through experiments on the `TeaStore`, a microservice reference test application, and `Znn`, a containerized news portal. Our `DockerMV` extension can be used by software developers to introduce

multi-versioning in their own containerized software systems, thereby better allowing them to meet the performance requirements of their systems.

3.2 Introduction

As the popularity and complexity of software systems increase, it becomes more challenging to satisfy the performance requirements of such systems. For example, one common problem that may happen for a web-based software system is the Slashdot effect. The Slashdot effect is a resource allocation problem that happens when a high-traffic website posts a link to a low-traffic website [1]. If the low-traffic website is not capable of handling the sudden increase in traffic, it may experience prolonged response times or unavailability, thereby violating the website's performance requirements. One common remedy to this problem is to allocate more server resources to make sure that the performance of the website satisfies the requirements. However, this approach can become very expensive and could add high over-provisioning costs, which not every project can afford. An alternative solution could be to have different versions of the services provided by the website. For instance, if the website had lightweight versions of some of its essential, resource-heavy components, it could use them during the high load to reduce its resource usage while maintaining reasonable response times. A similar example of this *software multi-versioning* concept has been used by Google's Gmail, which has a lightweight HTML-based version that is used when the user's browser does not support the feature-rich but resource-heavy JavaScript-based version [28]. By falling back on the lightweight version, the user would still be able to use Gmail, albeit at a reduced quality of service.

Software multi-versioning is traditionally applied to mission-critical systems, such as flight or nuclear power plant control systems, to improve their dependability, reliability or fault tolerance [5, 6, 24, 36]. As these systems are often monolithic, software multi-versioning requires maintaining several full versions of the system, making it a costly process. As a result, software multi-versioning has never been widely used for non-critical systems, as the cost of maintaining several versions usually does not outweigh the benefits for non-critical systems.

However, the advent of systems with containerized architectures, such as microservice-based ones, opens many new opportunities for applying software multi-versioning. As these systems are divided into smaller components that each run inside their own container, we can apply software multi-versioning to a component rather than the whole system. Figure 3.1 shows an example of an architecture of a microservice-based application (the `TeaStore` application [39]) in which the Recommender microservice uses multi-versioning. For every request, the system can select at runtime whether the LightWeight or HeavyWeight version of the Recommender microservice will be used to fulfill the request.

In this chapter, we examine how software multi-versioning can help satisfy the performance requirements of containerized software systems. We conduct two experiments on the performance of two containerized systems under varying loads. In the first experiment, we study the `TeaStore` application [39], which is a reference application for benchmarking and testing microservices. We applied multi-versioning to its Recommender service to simulate an accurate but resource-heavy recommendation algorithm, and a less accurate but more lightweight version. In our second experiment, we study a containerized three-tier online news application (the `Znn` application [13]) where our adapted version of the `Znn` application reduces the level of service during the high load by using different versions of the content-providing component.

To implement our experiments, we present an extended version of the `Docker` container platform (`DockerMV`) that allows the creation of multi-version services by deploying several containers for each version of the service. To allow service developers to control the load balancing between the multiple versions of their service in a transparent manner, `DockerMV` provides a rule-based load balancer which can be configured at the service-level rather than at the system level. Hence, by using `DockerMV`, developers can extend their own containerized systems with multi-versioned services

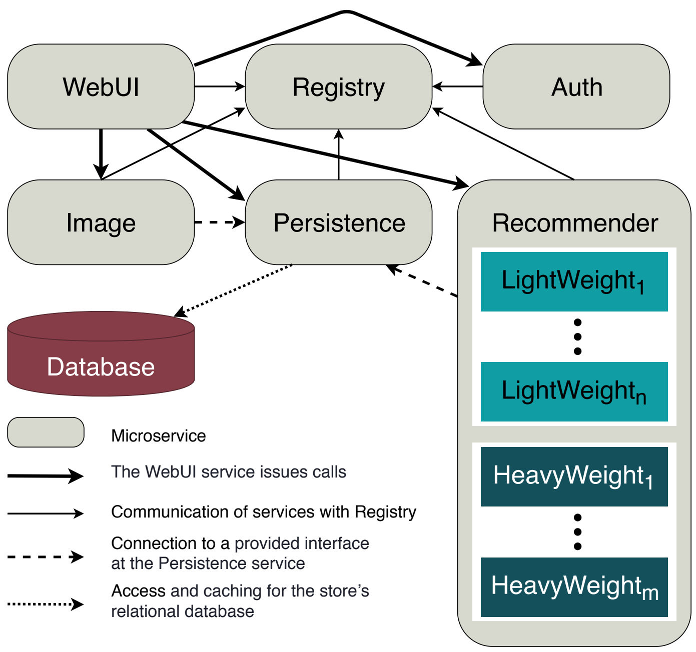

WebUI
Registry
Auth
Image
Persistence
Recommender
LightWeight$_1$
LightWeight$_n$
HeavyWeight$_1$
HeavyWeight$_m$
Database
Microservice
The WebUI service issues calls
Communication of services with Registry
Connection to a provided interface at the Persistence service
Access and caching for the store's relational database

Figure 3.1: High-level architecture of the `TeaStore` application with multi-versioning of the Recommender microservice.

in a manner that is transparent to the rest of the system.

The rest of the chapter is organized as follows. Section 3.3 provides background information about containerized software systems, microservices and managing containers. Section 3.4 presents a motivational example for our approach. In Section 3.5, we present the concept of our approach. In Section 3.6, we explain our experimental setup. Section 3.7 discusses the results of our experiments. Section 3.8 gives an overview of the related work, and Section 3.9 explains the threats to the validity of our work. Finally, Section 3.10 concludes the chapter.

3.3 Background

In this section, we provide background information about containerized software systems and managing containers.

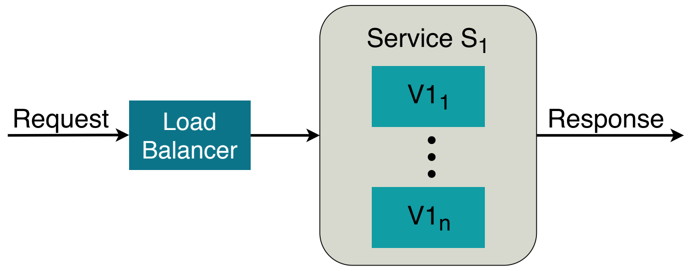


Figure 3.2: High-level architecture of a regular service in `Docker` where requests are load balanced in a Round Robin manner.

3.3.1 Containerized Software Systems

One of the essential techniques that enable cloud computing is virtualization [50], which is used to create virtual environments in which processes or services are isolated from each other [7], thereby allowing multi-tenancy of hardware resources [9, 35]. Traditionally, virtualization is achieved using a hypervisor. A hypervisor is a process to create and run virtual machines (VMs) on a host system, making it appear that each VM is using its own independent hardware resources. Some well-known examples of hypervisors are VMware ESX, KVM, Xen, and Hyper-V [2, 43, 59]. When using hypervisors for virtualization, each virtual machine runs its own operating system (OS) on the host system, which makes the virtual machines resource-heavy and severely limits the number of virtual machines that can run in parallel on a single host.

A recent advancement in virtualization techniques is the advent of lightweight software containers, which share the OS, binaries and libraries of the host system. As a result, containers are smaller and more lightweight than virtual machines that are started by a hypervisor. Hence, it is possible to run hundreds of containers on a single host machine. Also, as these containers use the host's OS, they can be started much faster [7].

Microservices

Microservices are a popular architectural approach for creating containerized software systems which is inspired by service-oriented computing [23]. In the microservices architecture, the system is developed from a set of small independent services [46]. While microservices can be deployed in virtual machines, the best way to leverage their full potential is to run them inside containers [7, 58]. The independence of microservices allows developers to work on them separately and use the most suitable technology to develop each of them [46]. Also, microservices can be modified independently as the requirements of the system change. Microservices communicate through RESTful APIs or a message-based protocol, which allows to scale an application quickly by replicating the microservice that is under heavy load [38].

3.3.2 Managing Containers

A popular framework for deploying software containers is the open source `Docker` container platform.[1] `Docker` combines several kernel-level technologies such as `LXC` and `cgroups` to enable the deployment and reuse of highly portable, lightweight containers [43]. A `Docker` container is a runtime instance of a `Docker` image. A `Docker` image specifies everything that is necessary to run an application as a container, for example, which libraries should be enabled in the container and how they should be configured [17]. When `Docker` executes in `swarm` mode [22], (replications of) containers are started as services that are part of a larger, service-based system (e.g., a microservice-based one) [21]. Figure 3.2 shows the high-level architecture of a service S_1 that consists of n exact replicas of containers that run version V_1 of the service. The load balancer balances the traffic to the service in a Round Robin-manner between its n containers. One of the main benefits of a `Docker` service is that the service appears as a single unit to other parts of the system, regardless of the number of replicated containers it consists of. Hence, other parts of the system need not be aware of the

[1]https://www.docker.com/

load balancing.

One downside of `Docker` services is that all the containers of a service are exact replicas. In the next section, we present a motivating example in which it would be beneficial to have a service that consists of containers that run different versions of the service.

3.4 A Motivating Example

Erica is a developer who works for an e-commerce start-up company that sells products online. The start-up company has migrated all of its software systems to containerized ones. As the start-up has limited financial resources, it is important to run their systems in a cost-effective manner. Erica is responsible for designing the algorithm that recommends new products to the customers based on the customer's shopping cart, the customer's order history, or the popularity of the items. Erica suggested several algorithms for the recommendation system, each with their own strengths and weaknesses. While the algorithm needs to be fast, it should provide high quality recommendations as well.

Unfortunately, Erica noticed that the performance requirements of one the software systems could not be satisfied when the recommendation algorithm was enabled. Erica's first solution was to increase the allocated server resources. However, the start-up company cannot afford these extra costs. Instead, Erica decided to implement two versions of the algorithm; one resource-heavy version that provides high quality recommendations, and one lightweight version that provides lower quality but still acceptable recommendations. Hence, by switching between the algorithms as the availability of resources allows, the system can make the trade-off between resource usage and recommendation quality. For example, when there is a sale event happening on the website the lightweight algorithm can be used, to ensure the recommendation algorithm does not consume too many resources.

Listing 3.1: The original `docker service create` command. We omitted the arguments that are not relevant to our work for clarity.

```
$ docker service create [$OPTIONS] $IMAGE [$REPLICATION]
```

3.5 Our Approach

Software multi-versioning is the concept of developing and running several different versions of a software system or component to improve one or more of the system's quality attributes. Our approach is to apply software multi-versioning to the containers of a service in a containerized software system. To deploy multi-version services, we need to deploy multiple containers, each of which are instantiated from different container images. Our goal is to implement multi-versioning in a transparent manner, i.e., users of the services and/or containers are not aware of the multi-versioning. Hence, our multi-version containers should form a unified service which can be treated like a regular single-version service.

3.5.1 Implementation

To implement our approach, we extended the `Docker` framework into the `DockerMV` framework. To create a service with the original `Docker` framework, the `docker service create` command in Listing 3.1 is used. The original command takes the following parameters:

- `$OPTIONS`: Optional parameters that can be used to configure container-specific parameters, such as the environment variables and the memory limit.
- `$IMAGE`: The image from which the container should be created.
- `$REPLICATIONS`: The number of replications of the container that should be created.

The command in Listing 3.1 will create a service that consists of `$REPLICATIONS` exact copies of the container that is created from the `$IMAGE` image with the configuration options specified in `$OPTIONS`. The `docker service create` command in Listing 3.1 does not support multi-versioning. Therefore, we extended the command's implementation to accept multiple images with different replication and configuration parameter values. Listing 3.2 shows the extended command, which allows the creation of multi-version `Docker` services. In particular, the extended command allows the creation of a `Docker` service that consists of $\$REPLICATIONS_1 + ... + \$REPLICATIONS_n$ containers, that were created from n images. In addition, the extended `docker service create` command supports the following parameters:

- `Network`: The name of the overlay network to connect the containers to each other (fixed for all containers in the `Docker` service) (Required).
- `Name`: The `Docker` service name (fixed for all containers in the `Docker` service) (Required).
- `Environment variables`: The environment variables (fixed for all containers in the `Docker` service) (Optional).
- `Memory`: The memory limit for a container (Optional).
- `Swap memory`: The swap memory limit for a container (Optional).
- `CPU`: The number of CPUs for a container (Optional).
- `Container port`: The port that the containers of the `Docker` service will listen on (fixed for all containers in the `Docker` service) (Required).
- `Rule-set`: The location of the user-defined rule-set.

In our extended command, the `network, name, environment variables,` and `container port` parameters have the same value across all containers of the service. However, the `memory, swap memory`, and `CPU` can be configured differently for each container in the service. Listing 3.3 shows an example invocation of the extended

Listing 3.2: The extended `docker service create` command

```
$ docker service create [$OPTIONS]
    $IMAGE_1 $REPLICATIONS_1
    ...
    $IMAGE_n $REPLICATIONS_n
```

Listing 3.3: An example invocation of the extended `docker service create` command that deploys two versions of the `teastore-recommender` service.

```
$ docker service create
    e REGISTRY_HOST=host_IP e REGISTRY_PORT=10000
    e HOST_NAME=host_IP e SERVICE_PORT=3333
    10.2.5.26 Network recommender 8080 1g 1g 0.2 rules.txt
    sgholami/teastore-recommender:HeavyWeight 1
    sgholami/teastore-recommender:LightWeight 1
```

command (which is part of our `DockerMV` extension). In particular, two versions of the `teastore-recommender` service are started (one replication of each), that are connected to the `recommender` network. Each of the containers initializes four environment variables (`REGISTRY_HOST`, `REGISTRY_PORT`, `HOST_NAME` and `SERVICE_PORT`).

3.5.2 Load Balancing

Figure 3.2 shows an example architecture of a `Docker` service. As shown in Figure 3.2, a `Docker` service has a load balancer that distributes the incoming requests between the service's containers. As these containers are created from the same image, the load balancer usually distributes the incoming traffic in a round-robin manner (i.e., an equal amount of traffic to each container) [22]. Figure 3.3 shows the architecture of a service that consists of containers made from different images. As these containers are created from different images, they may perform a similar task at different quality of service levels, e.g., comparable to our motivating example in Section 3.4. Hence, it may no longer be desirable to distribute the traffic in a round-robin manner. Instead, we would like to balance the load based on performance metrics of the service, such as median response times or CPU utilization. Therefore, we implemented a rule-based

Listing 3.4: Format of the rules for the load balancer

```
$METRIC $OPERATOR $THRESHOLD ,
    (version $VERSION_NAME perc=$PERCENTAGE;)+
```

Listing 3.5: Example rule for the load balancer

```
RT > 0.4 ,
    version recommender:HeavyWeight perc=40;
    version recommender:LightWeight perc=60;
```

load balancer in our services. We used a customized version of NGINX[2] as the load balancer amongst the different replications of a service's containers. Our customized load balancer has a user-defined rule-set which defines how to balance the incoming traffic to satisfy a system's performance requirements. Listing 3.4 shows the format of the rules for the load balancer.

The parameters in the rule in Listing 3.4 are as follows:

- `$METRIC`: The metric that is used to check whether a rule should fire. Currently only `RT` (median response time) is supported.
- `$OPERATOR`: The relational operator ($<$, $\leq$, $>$, $\geq$ or $==$) that is used in the condition to check whether a rule should fire.
- `$THRESHOLD`: The threshold for the metric that is used in the condition to check whether a rule should fire.
- `$VERSION_NAME`: The name of one of the versions of the service.
- `$PERCENTAGE`: The percentage of requests to be directed to the container (between 1 and 100).

Listing 3.5 shows an example rule, in which 40% of the requests are directed to the first container (i.e., the HeavyWeight version of the service), and the second

[2]https://www.nginx.com

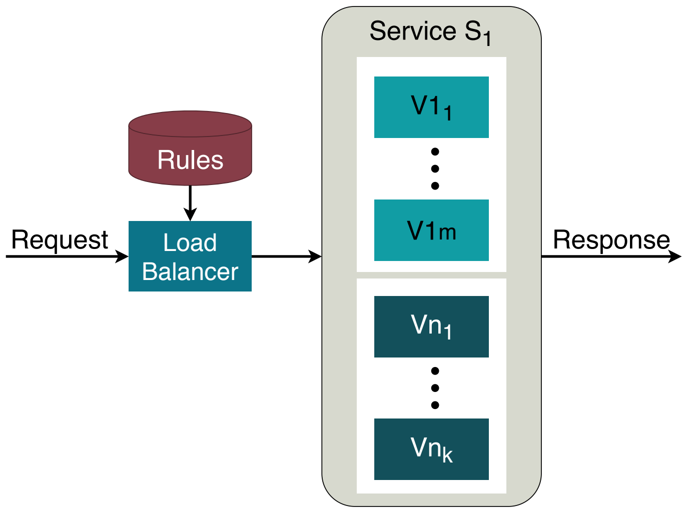


Figure 3.3: High-level architecture of a service with multi-versioning where requests are balanced based on a rule-set.

(LightWeight) container handles the other 60% of the requests. We recalculate the median response time every five seconds from NGINX's log file. NGINX uses this median value to decide which rule should be used when balancing the incoming traffic. NGINX saves the `$time_local`, and `$request_time` for each of the incoming requests. The `$time_local` returns the local time of the machine, and we use that time to identify the requests which were received in the last n seconds. The `$request_time` is the elapsed time since the first bytes were read from the client.

3.5.3 On the Necessity of Our Approach

One could argue that software multi-versioning could easily be achieved using `if`-statements inside a service's source code, or by simply starting multiple services (i.e., one for each version). However, source code-based solutions have the disadvantage that they clutter the source code, making maintenance and understanding of the code more challenging. In addition, starting multiple services causes software multi-versioning to no longer be transparent, which has obvious (negative) consequences for the other parts of the system. For example, the system now needs to be aware of more complex load balancing requirements. Hence, our approach is necessary to provide

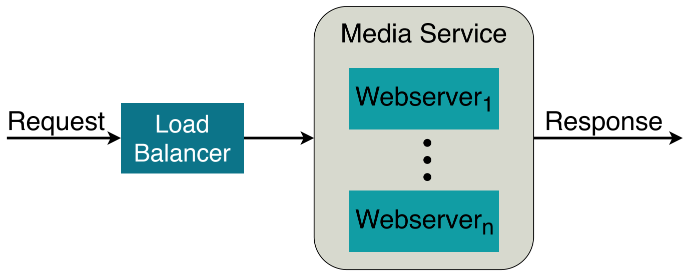


Figure 3.4: High-level architecture of the `Znn` application

multi-versioning in containerized systems in a transparent, non-cluttered manner.

3.6 Experimental Setup

In this section, we elaborate on our experimental setup. The goal of our experiments is to study the benefits of software multi-versioning for satisfying the performance requirements of containerized software systems.

3.6.1 Subject Systems

In our experiments, we study the `TeaStore` [39] and the `Znn` applications. The `TeaStore` application is a reference microservice application that can be used for performance testing and benchmarking. The `TeaStore` application simulates an online store that is composed of six microservices (see Figure 3.1). Every microservice runs inside its own container. In addition, the database runs inside its own container.

The `Znn` application [13] is a three-tier web-based news portal that can be used for testing and benchmarking of self-adaptive applications. The `Znn` application contains a pool of web servers, a MySQL database with news-related text and multimedia contents, and a load balancer that receives requests from clients and distributes them among the web servers in a Round Robin manner. The high-level architecture of the `Znn` application is shown in Figure 3.4. The source code of the `TeaStore`[3] and the `Znn`[4] applications are both publicly available.

[3]https://github.com/DescartesResearch/TeaStore
[4]https://github.com/cmu-able/znn

3.6.2 Introducing Multi-Versioning in the Subject Systems

To introduce multi-versioning in the `TeaStore` application, we adapted the `Recommender` service, which is designed to return recommendations based on the user's history and items in their shopping cart (similar to our motivating example in Section 3.4). The `TeaStore` application provides several algorithms and trains them once the service is first launched. To conduct our experiment, we selected one of the algorithms, which is the SlopeOne algorithm, and forced retraining every two minutes. The multiple retraining is applied to simulate a higher load and pressure on the system in which the `Recommender` service is replicated (and hence retrained) on several containers. The retraining causes slower response times of the `Recommender`. As a result, we use two versions of the `Recommender` in our experiment, one with regular retraining (`HeavyWeight`) and another with a single training (`LightWeight`). Figure 3.1 shows the architecture of the `TeaStore` application with the multi-versioned `Recommender` service.

The `Znn` application returns news articles that contain multimedia contents (such as a video that is sent by the web server). Therefore, when the load of the system increases, the system's median response time rises (as the network bandwidth becomes a bottleneck). Hence, we created two different versions of the web servers. The first version provides the original news article along with its multimedia contents, while the other version of the service returns only the text contents of the news. Figure 3.5 shows the high-level architecture of the containerized version of the `Znn` application with multi-versioning.

3.6.3 Experiments

We conducted three experiments for each of the subject systems:

- **Ideal Case Experiment**: In this experiment, we tested the "ideal case" for each of the systems, i.e., the case in which all requests are served by the heavy-

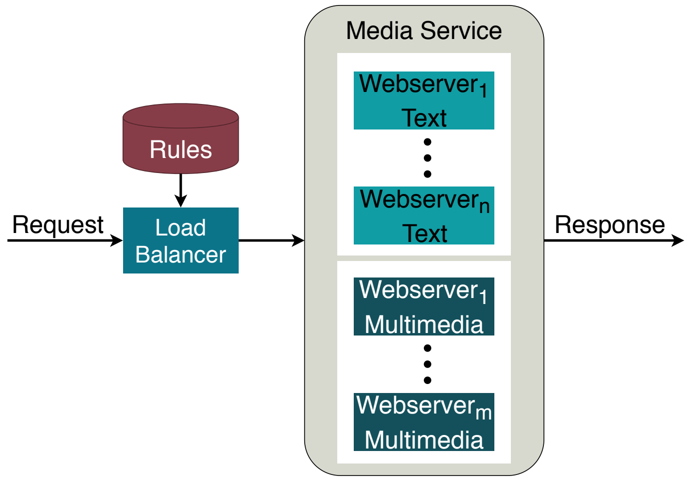

Media Service
Rules
Request
Load Balancer
Webserver_1 Text
Webserver_n Text
Webserver_1 Multimedia
Webserver_m Multimedia
Response

Figure 3.5: Containerized deployment of `Znn` in which we have two different versions of the `Media` service.

weight versions of the services. Hence, for the `TeaStore` application all requests are served by the `Recommender` that is constantly retrained, and for `Znn` all requests receive a multimedia response.

- **Worst Case Experiment**: In the second experiment, we tested the worst-case setup (in terms of quality of service, i.e., we only used the lightweight versions of the services) for each of the subject systems. For the `TeaStore` application, the worst case is to use only the `Recommender` service with a single training. In the `Znn` application, the worst case is to return only the text responses.

- **Adaptive Experiment**: Finally, we studied how multi-versioning together with an adaptive balancing of the workload can help to satisfy the performance requirements. For each of the subject systems, we deployed both of the versions of the services and balance the load based on a customized rule-set. In this setup, we used our extended version of `Docker` (`DockerMV`) along with our customized NGINX load balancer.

These experiments are summarized in Table 3.1. To demonstrate our approach, we defined the performance requirements as follows:

- For the `TeaStore` application, we set 450 milliseconds as the upper limit for the median response time.
- For the `Znn` application, we set 1 second as the upper limit for the median response time.

Both of these performance requirements were defined empirically based on the ideal and worst case experiments. Please note that the exact choice of performance requirements does not matter much—our sole purpose in this chapter is to demonstrate the efficacy of software multi-versioning to satisfy performance requirements.

Table 3.1: A description of the experiments that we conducted for the `TeaStore` and `Znn` applications

	`TeaStore` Description	`Znn` Description
Ideal case experiment	`Recommender` with multiple training	Multimedia responses only
Worst case experiment	`Recommender` with single training	Text responses only
Adaptive experiment	Adaptive load distribution	Adaptive load distribution

3.6.4 Workload

We used Apache JMeter,[5] a tool for load testing web applications to generate workloads for our experiments.

For the `TeaStore` application, we generated the workload using the JMeter script that is provided by the `TeaStore` developers and modified it to add more items to the shopping cart to put more pressure on the `Recommender` service. We generated a workload of 100 users who concurrently send HTTP requests to the `TeaStore` application for different purposes such as opening the home page, logging in, or adding items to the cart. This workload continued for 1,000 seconds. Each user sends an

[5]https://jmeter.apache.org

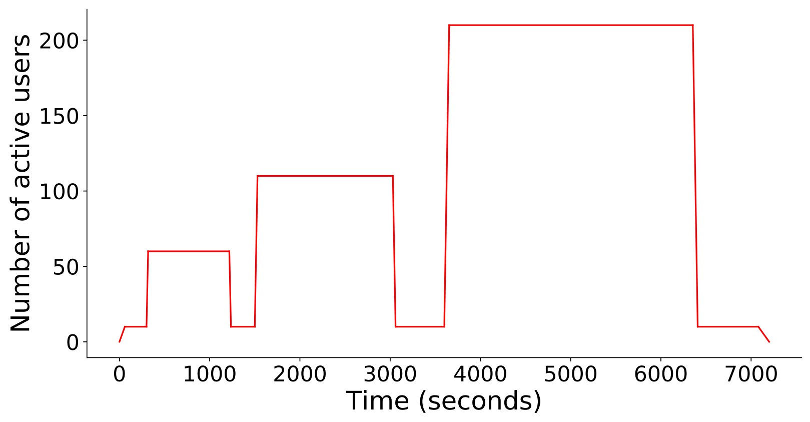


Figure 3.6: Shape of the `Znn` application workload

HTTP request to the server and receives an HTML page, and as soon as the user receives a response, they send the next request.

For the `Znn` application, we generated a workload that sends HTTP requests and simulates multiple users sending requests to the `Znn` application concurrently. Figure 3.6 shows the shape of the workload and the number of active users during each of the two-hour experiments. For instance, at the highest peak where the number of active users is 200, it means that 200 threads concurrently send requests to the servers, and when they get a response, they send another request. We use the same workload across the experiments for each of the subject systems.

3.6.5 Deployment of the Subject Systems

Table 3.2 shows the description of the containers that we used for the experiments. We limited the containers' memory, swap memory, and CPU to stop them from growing too much and allocating all of the available resources. These limits were defined based on our experience with the subject systems.

We provisioned one virtual machine in the Compute Canada cloud[6] and one virtual machine in the Cybera Rapid Access Cloud[7] to run our containers for both of the

[6]https://www.computecanada.ca/research-portal/national-services/compute-canada-cloud
[7]https://www.cybera.ca/services/rapid-access-cloud/

Table 3.2: Description of the containers in the experiments

Name	Docker Image	Memory	Swap Memory	CPU
HeavyWeight-Recommender	sgholami/teastore-recommender:HeavyWeight	1G	1G	0.4
LightWeight-Recommender	sgholami/teastore-recommender:LightWeight	1G	1G	0.4
Multimedia	alirezagoli/znn-multimedia:v1	1G	1G	0.4
Text	alirezagoli/znn-text:v1	1G	1G	0.4
NGINX	sgholami/nginx-monitoring	unlimited	unlimited	unlimited
NGINX_official	NGINX	unlimited	unlimited	unlimited
MySQL	alirezagoli/znn-mysql:v1	unlimited	unlimited	unlimited

experiments. In particular, we ran the JMeter script on the Compute Canada cloud and the subject systems on the Cybera cloud. Table 3.3 summarizes the configurations of our virtual machines.

Table 3.3: Description of the virtual machines

Cloud	Instance	VCPUs	Memory	OS
Cybera	Experiment	4	8GB	Ubuntu-18.04
Compute Canada	JMeter	4	15GB	Ubuntu-18.04

The source code of the `DockerMV` and more details about our experiments can be found on the project's GitHub repository [27].

3.6.6 Load Balancing

For the `TeaStore` application, we set the rules presented in Listing 3.6 for NGINX to balance the load between the versions of the `Recommender` service. Listing 3.7 shows the set of rules for the `Znn` application. Both rule sets were defined empirically based on observations during preliminary runs of the experiments.

Listing 3.6: NGINX rule set for the `TeaStore` application

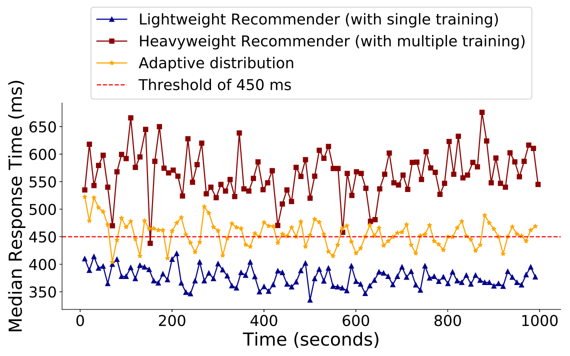
```
RT < 0.1 ,
      version recommender:HeavyWeight perc=99;
      version recommender:LightWeight perc=1;
RT < 0.25 ,
      version recommender:HeavyWeight perc=90;
      version recommender:LightWeight perc=10;
RT < 0.4 ,
      version recommender:HeavyWeight perc=80;
      version recommender:LightWeight perc=20;
RT >= 0.4 ,
      version recommender:HeavyWeight perc=70;
      version recommender:LightWeight perc=30;
```

Listing 3.7: NGINX rule set for the `Znn` application

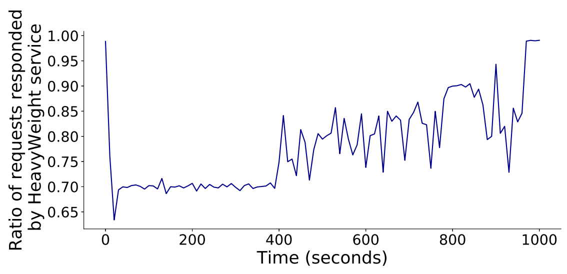
```
RT < 0.1 ,
      version znn-multimedia:v1 perc=99;
      version znn-text:v1 perc=1;
RT < 0.2 ,
      version znn-multimedia:v1 perc=80;
      version znn-text:v1 perc=20;
RT < 0.3 ,
      version znn-multimedia:v1 perc=70;
      version znn-text:v1 perc=30;
RT < 0.6 ,
      version znn-multimedia:v1 perc=40;
      version znn-text:v1 perc=60;
RT < 0.8 ,
      version znn-multimedia:v1 perc=30;
      version znn-text:v1 perc=70;
RT >= 0.8 ,
      version znn-multimedia:v1 perc=20;
      version znn-text:v1 perc=80;
```

3.7 Experimental Evaluation

In this section, we discuss the results of our experiments for each subject system.



(a) Median response time of the `TeaStore` application



(b) The ratio of requests responded by the HeavyWeight version of the service

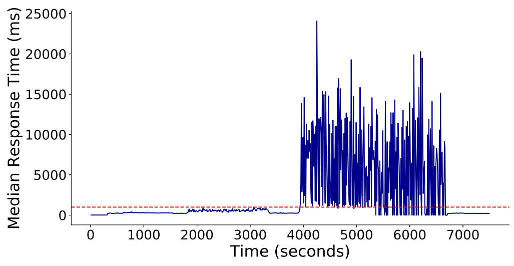
Figure 3.7: The `TeaStore` application experiments and the distribution ratio of traffic using software multi-versioning and adaptive load balancing

3.7.1 Experiments with the `TeaStore` Application

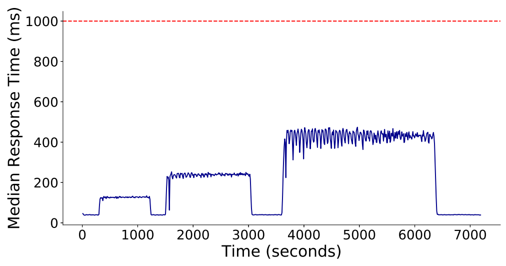
Figure 3.7a shows the median response times of the `TeaStore` application in our experiments. We illustrated the result of all tests in one plot as the range of their values is close, and it is possible to observe the changes in all of them together. Figure 3.7b shows the ratio of requests that were responded to by the HeavyWeight version of the `Recommender` service. We observe that the median response time fluctuates around our performance requirement threshold as the load balancer distributes the load between the HeavyWeight and LightWeight versions of the service.

3.7.2 Experiments with the `Znn` Application

Figure 3.8a shows the median response time of the ideal case for the `Znn` application when we are using only the multimedia version of the service. During this experiment, the median response time of the application goes up to around 25 seconds, which indicates that the resources for the application are severely under provisioned. Figure 3.8b shows the median response time of the worst case experiment, which shows that the available resources can easily handle this type of traffic. However, the quality of service is considerably reduced since all requests are handled by the text version of the service. Figure 3.9a shows the median response time when using

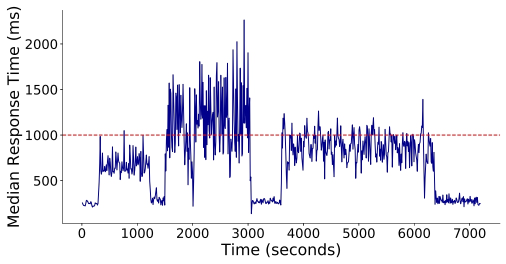


(a) The median response time when running only the multimedia-version of the service

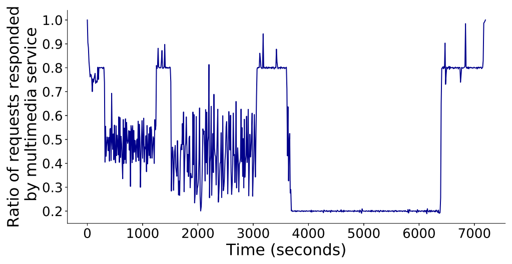


(b) The median response time when running only the text version of the service

Figure 3.8: The `Znn` application experiments using only the multimedia vs only the text version of the service. Note that the scales on the y-axes are different.



(a) The median response time when running the services with multi-versioning and using the rules defined in Listing 3.7



(b) The ratio of requests responded by the multimedia version of the service

Figure 3.9: The `Znn` application experiment using software multi-versioning and adaptive load balancing

software multi-versioning to balance between the multimedia and text version of the service. In addition, Figure 3.9b shows the ratio of the requests which were responded to by the multimedia server in the adaptive experiment. Figures 3.9a and 3.9b show that the system deals with the increases in workload by balancing the majority of the requests (first approximately 50-70% and then approximately 80%) to the text version of the service.

Summary: *Multi-versioning allows us to satisfy the performance requirements of subject systems while maintaining a level of Quality of Service that is as high as possible using the given resources.*

3.8 Related Work

In this section, we discuss prior work that is related to ours. In particular, we discuss related work on software multi-versioning, software multi-versioning for containerized systems, and performance engineering of containerized systems.

3.8.1 Software Multi-Versioning

Until now, software multi-versioning has been used for several purposes, such as improving a system's security [10, 25, 40, 51], safety [30], reliability [14], and availability [29]. In these cases, software multi-versioning is often used as a means to achieve software redundancy, i.e., to have several different versions of the software that are functionally equivalent, yet different in terms of e.g., implementation or used implementation language.

Larsen et al. [40] studied the effect of automated software redundancy on a system's security. Franz et al. [25] used software multi-versioning as a defense mechanism for a system. The idea of their approach is that as the system has several versions, it is harder to figure out for the attackers which version they are attacking. Therefore, they have less chance to succeed in their attack, which increases the system's security. Persaud et al. [51] used software redundancy by using Genetic Algorithms to enhance the security of the system. Cigsar et al. [14] considered software multi-versioning as an approach to improve the reliability of repairable systems. Gracie et al. [30] stated that there have been designs for using redundancy for safety purposes. Gorbenko et al. [29] used software multi-versioning for a web service to extend its functionality and improve its attributes such as availability and reliability.

3.8.2 Software Multi-Versioning for Containerized Systems

In containerized systems, software multi-versioning has been used mostly for purposes such as enhancing the fault tolerance. [61, 67], security, and reliability [66] of the systems. For example, Wang et al. [61] suggested the idea of applying multi-versioning to critical components of cloud-based software to improve the fault tolerance of the system. Wang et al. proposed an approach to find the critical components of the system and apply software multi-versioning only to those critical components to reduce the cost and complexity of software multi-versioning while improving the system's fault tolerance. Also, Zheng et al. showed that software multi-versioning can be used to improve the reliability [66] and fault tolerance [67] of service-oriented systems. However, the choice of using multi-versioning can affect the quality of service of the system. Therefore, Zheng et al. formulated the reliability requirements as an optimization problem and proposed a heuristic algorithm to maintain the quality of the system by solving this optimization problem.

All prior studies on applying software multi-versioning to containerized software systems focused on improving the reliability of a system. We are the first to study the benefits of software multi-versioning for satisfying the performance requirements of a system.

3.8.3 Performance Engineering of Containerized Systems

Recently, performance engineering researchers have started to study performance engineering for microservices. For example, Heinrich et al. gave an overview of the challenges of performance engineering microservices [31]. They identified performance testing, monitoring and modeling of microservices as the main performance engineering challenges. Performance testing microservices is challenging, as the services are developed and maintained independently. Therefore, Camargo et al. [12] presented an approach to automate the performance testing for microservices. In this approach, each microservice provides a test specification that was used for performing the tests. Jindal et al. [34] addressed performance modeling of microservices by capacity planning. They identified a microservice's capacity to find the appropriate resource needed for the microservices. As a result, the system would not violate the performance requirements. Amaral et al. [3] studied two models for microservices architecture using containers. They compared the performance of CPU and network for the master-slave and nested-container models to provide a guide for system designers.

A large body of the existing performance engineering work for containerized systems is about the performance of cloud systems. As performance engineering for cloud systems is a very broad topic, a thorough discussion of this body of work is outside the scope of this chapter, and we refer the reader to one of the excellent surveys on this topic, e.g., the ones by Xu et al. [63] or Nuaimi et al. [49]. Also, Ruan et al. [56] studied the performance of cloud systems by using containers from different perspectives.

3.9 Threats to Validity

In this chapter, we studied how software multi-versioning can help to satisfy the performance requirements of containerized software systems. In this section, we discuss the threats to the validity of this study.

3.9.1 External Validity

The Choice of Subject Systems. We studied two open source applications, one of which is a microservices application (the `TeaStore` application), and the other is a more traditional three-tier application (the `Znn` application). The `Znn` application is not originally a containerized application, although based on our experience, many three-tier applications are containerized in a similar manner as we did in this study. While we aimed to select systems that are representative for larger groups of systems, future studies should investigate how well software multi-versioning works for a wider range of systems, such as industrial systems.

3.9.2 Internal Validity

The Choice of Performance Requirements. The performance requirements that we used in our experiments were defined empirically based on the ideal and worst case experiments. As our purpose in this chapter is to demonstrate the efficacy of software multi-versioning to satisfy performance requirements, the exact values of the requirements do not matter much. There could exist characteristics that make some requirements more difficult to satisfy than others. For example, if a light weight version of a service already has difficulties to satisfy a performance requirement given the available resources, software multi-versioning will not help much (since the load balancer will simply divert all traffic to the light weight version). Hence, future studies should further investigate how the choice of performance requirements impacts the efficacy of software multi-versioning to satisfy those requirements.

The Choice of Load Balancing Rules. As the focus of our work is to demonstrate the efficacy of software multi-versioning for satisfying performance requirements, and not to present a novel load balancing technique, in our experiments the load balancing is done by a simple static approach. Users of `DockerMV` can easily adapt the load balancing rules to implement more advanced load balancing techniques for their own systems, such as those proposed by Niu et al. [48], Radojevic et al. [54] or Dasguptaat al. [15]. While our experiments show that the used simple rule sets can already yield satisfactory results, future studies should investigate how to optimize the rules on a per-system and per-workload basis.

3.9.3 Construct Validity

The Choice of Performance Metric. We chose median response time as our performance metric as it the primary metric that is used for measuring the user-perceived performance. Future studies should consider how software multi-versioning can benefit other performance metrics, such as CPU utilization or memory usage.

The Overhead of Software Multi-Versioning. We did not measure the overhead that is added by introducing software multi-versioning to containerized systems. However, given that the additional load balancing is fairly simple and straightforward, there should not be a significant amount of additional overhead introduced.

3.10 Conclusion

Traditionally, software multi-versioning has been applied only to mission-critical systems due to the high cost of maintaining multiple versions of the software. Recently the increase in popularity of containerized software systems has opened many new opportunities for the application of software multi-versioning, as the technique can be applied to smaller parts of these systems.

In this chapter, we study how software multi-versioning can help to satisfy the performance requirements of containerized software systems. In summary, our chapter makes the following contributions:

- **A demonstration that software multi-versioning can effectively be applied to satisfy the performance requirements of containerized software systems.** We show through experiments on two open source applications that software multi-versioning can effectively be applied to containerized systems to satisfy performance requirements while maintaining a quality of service-level that is still acceptable given the available resources.

- **A framework to deploy services with software multi-versioning.** We extended the `Docker` container platform to allow the creation of multi-version services. Our `DockerMV` platform supports custom rule-based load balancing between the versions of a service that can be controlled by the service developer, and hence is transparent to the other parts of the system. Our `DockerMV` implementation is publicly available [27].

We are one of the first ones to study the benefits of software multi-versioning for containerized systems. In particular, we are the first to demonstrate how software multi-versioning can help to satisfy the performance requirements of such systems. Our expectation is that our `DockerMV` platform can help to satisfy other nonfunctional requirements of containerized software system, such as dependability, reliability, availability and security requirements. Hence, future studies can leverage our platform to investigate how software multi-versioning can be applied to further help satisfy the nonfunctional requirements of containerized software systems.

Chapter 4

Conclusions & Future Work

4.1 Conclusions

In this section, we highlight our contributions and findings in each of our studies.

In Chapter 2, we investigated the updates in the official `Docker` repositories on `Docker Hub` and focused on how native, *Node*, and *Python* packages are changing in different types of applications. We found that all studied applications had changing packages. In addition, we found that the type of application affects the median number of package changes. For instance, the *Operating Systems* and *Base Images* repositories had a low median number of changes, while *Application Services* and *Analytics* repositories had the highest median number of changes. However, practitioners need to take caution when doing in-place updates on images from different applications, as all of them have some package changes in their image updates.

In Chapter 3, we proposed our solution to mitigate some of the identified threats in our previous study. Our approach to preventing having poor performance as a result of updating system packages is to maintain the previous version of the system along with the updated version and gradually move to the updated version. Therefore, we implemented `DockerMV`, an extension of `Docker` framework with multi-versioning, to deploy multiple versions of an image under one service. In this study, we focused on the performance point of view and demonstrated that containerized software systems' performance requirements could be satisfied using software multi-versioning. We conducted two sets of experiments on two open-source applications as our subject systems and showed that the performance requirements could be met while maintaining an acceptable level of quality of service.

4.2 Future Work

The following is a list of possible future directions for research that follow from the work presented in this thesis:

- **Future direction 1: Automatically identifying development branches.** In our first study, we manually divided the images of each repository into branches. Further work is required to automate the branching of the images in each repository to be able to analyze more repositories.

- **Future direction 2: Quantifying the risk of package changes through other metrics (e.g., performance).** Until now, prior work only quantified the risk of package changes from a security point of view.

- **Future direction 3: Using `DockerMV` to satisfy other nonfunctional requirements.** We used `DockerMV` to satisfy the performance requirements of containerized software systems. However, `DockerMV` can be used for satisfying other nonfunctional requirements such as availability, reliability, dependability, and security.

- **Future direction 4: Improving `DockerMV`'s load balancing algorithm.** Currently, `DockerMV` uses a user-defined set of rules to distribute the traffic. The process of selecting these rules and the thresholds can be tedious, hence it should be automated. Machine learning approaches can be used to learn the workload and predict it to make a better load distribution.

Bibliography

[1] S. Adler, "The Slashdot effect: An analysis of three internet publications," *Linux Gazette*, vol. 38, no. 2, 1999.

[2] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2017.

[3] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *Proceedings of the 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34.

[4] C. Anderson, "Docker [software engineering]," *IEEE Software*, vol. 32, no. 3, pp. 102–c3, 2015.

[5] A. Avizienis and J. P. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *Computer*, no. 8, pp. 67–80, 1984.

[6] A. Avizienis and J. C. Laprie, "Dependable computing: From concepts to design diversity," *Proceedings of the IEEE*, vol. 74, no. 5, pp. 629–638, 1986.

[7] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[8] C.-P. Bezemer, S. Eismann, V. Ferme, J. Grohmann, R. Heinrich, P. Jamshidi, W. Shang, A. van Hoorn, M. Villavicencio, J. Walter, *et al.*, "How is performance addressed in DevOps?" In *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering*, ACM, 2019, pp. 45–50.

[9] C.-P. Bezemer and A. Zaidman, "Multi-tenant SaaS applications: Maintenance dream or nightmare?" In *Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ACM, 2010, pp. 88–92.

[10] H. Borck, M. Boddy, I. J. De Silva, S. Harp, K. Hoyme, S. Johnston, A. Schwerdfeger, and M. Southern, "Frankencode: Creating diverse programs using code clones," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 604–608.

[11] T. Bui, "Analysis of docker security," *arXiv preprint arXiv:1501.02967*, 2015.

[12] A. de Camargo, I. Salvadori, R. d. S. Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," in *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*, ACM, 2016, pp. 422–429.

[13] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the Rainbow self-adaptive system," in *Proceedings of the 9th Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, IEEE, 2009, pp. 132–141.

[14] C. Cigsar and Y. Lim, "Modeling and analysis of cluster of failures in redundant systems," in *Proceedings of the 2nd International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2017, pp. 119–124.

[15] K. Dasgupta, B. Mandal, P. Dutta, J. K. Mandal, and S. Dam, "A genetic algorithm (GA) based load balancing strategy for cloud computing," *Procedia Technology*, vol. 10, pp. 340–347, 2013.

[16] Docker Inc., *Docker Cloud*, https://www.docker.com/sites/default/files/Docker%20Cloud.pdf, Accessed: 2020-05-27.

[17] Docker Inc., *Docker container*, https://docs.docker.com/glossary/?term=container, Accessed: 2020-05-27.

[18] Docker Inc., *Docker overview*, https://docs.docker.com/get-started/overview/, Accessed: 2020-05-27.

[19] Docker Inc., *DockerFile reference*, https://docs.docker.com/engine/reference/builder/, Accessed: 2020-05-27.

[20] Docker Inc., *dotCloud, Inc. is now Docker, Inc.* https://www.docker.com/docker-news-and-press/dotcloud-inc-now-docker-inc, Accessed: 2020-05-27.

[21] Docker Inc., *How services work?* https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/, Accessed: 2020-05-27.

[22] Docker Inc., *Swarm mode key concepts*, https://docs.docker.com/engine/swarm/key-concepts/#load-balancing, Accessed: 2020-05-27.

[23] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and ulterior software engineering*, Springer, 2017, pp. 195–216.

[24] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 692–702, 1991.

[25] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the New Security Paradigms Workshop*, ACM, 2010, pp. 7–16.

[26] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei, "A framework for satisfying the performance requirements of containerized software systems through multi-versioning," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 150–160.

[27] S. Gholami, A. Goli, C.-P. Bezemer, and H. Khazaei, *DockerMV*, https://github.com/pacslab/DockerMV, Accessed: 2020-05-27.

[28] Gmail Help Center, *Gmail help*, https://support.google.com/mail/answer/15049?hl=en, Accessed: 2020-05-27.

[29] A. Gorbenko, V. Kharchenko, and A. Romanovsky, "Using inherent service redundancy and diversity to ensure web services dependability," in *Methods, Models and Tools for Fault Tolerance*, Springer, 2009, pp. 324–341.

[30] E. Gracie, A. Hayek, and J. Borcsok, "Evaluation of FPGA design tools for safety systems with on-chip redundancy referring to the standard IEC 61508," in *Proceedings of the 2nd International Conference on System Reliability and Safety (ICSRS)*, IEEE, 2017, pp. 386–390.

[31] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: Research challenges and directions," in *Companion of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, ACM, 2017.

[32] M. Heusser, *30 essential container technology tools and resources*, https://techbeacon.com/enterprise-it/30-essential-container-technology-tools-resources-0, Accessed: 2020-05-27.

[33] IBM Cloud Education, *Containerization*, https://www.ibm.com/cloud/learn/containerization, Accessed: 2020-05-08.

[34] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 10th ACM/SPEC on International Conference on Performance Engineering*, ACM, 2019, pp. 25–32.

[35] J. Kabbedijk, C.-P. Bezemer, A. Zaidman, and S. Jansen, "Defining multi-tenancy: A structured mapping study on the academic and industrial perspective," *Journal of Systems and Software (JSS)*, vol. 100, pp. 139–148, 2015.

[36] J. P. J. Kelly, T. I. McVittie, and W. I. Yamamoto, "Implementing design diversity to achieve fault tolerance," *IEEE Software*, vol. 8, no. 4, pp. 61–71, 1991.

[37] N. Kerzazi and B. Adams, "Botched releases: Do we need to roll back? Empirical study on a commercial web app," in *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, vol. 1, 2016, pp. 574–583.

[38] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, IBM Corp., 2017, pp. 234–240.

[39] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '18, IEEE, 2018.

[40] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, " SoK: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, IEEE, 2014, pp. 276–291.

[41] P. Leitner and C.-P. Bezemer, "An exploratory study of the state of practice of performance testing in Java-based open source projects," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ACM, 2017, pp. 373–384.

[42] J. Luu, "A deep dive into Docker Hub's security landscape- A story of inheritance?" Master's thesis, 2019.

[43] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[44] G. Mezzetti, A. Moller, and M. T. Torp, "Type regression testing to detect breaking changes in Node.js libraries," in *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[45] J. Morgan, *Introducing the new Docker Hub*, https://www.docker.com/blog/the-new-docker-hub/, Accessed: 2020-04-17.

[46] D. Namiot and M. Sneps-Sneppe, "On microservices architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[47] P. Nguyen, *Docker Store program and policies*, https://success.docker.com/article/store, Accessed: 2020-05-27.

[48] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *IEEE Conference on Computer Communications (INFOCOM)*, 2018, pp. 198–206.

[49] K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, and J. Al-Jaroodi, "A survey of load balancing in cloud computing: Challenges and algorithms," in *Proceedings of the 2nd Symposium on Network Cloud Computing and Applications*, 2012, pp. 137–142.

[50] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[51] B. Persaud, B. Obada-Obieh, N. Mansourzadeh, A. Moni, and A. Somayaji, "Frankenssl: Recombining cryptographic libraries for software diversity," in *Proceedings of the 11th Annual Symposium On Information Assurance. NYS Cyber Security Conference*, 2016, pp. 19–25.

[52] T. Preston-Werner, *Semantic versioning 2.0.0*, https://semver.org/, Accessed: 2020-05-27.

[53] T. Preston-Werner, *Semantic versioning 2.0.0*, https://semver.org/, Accessed: 2020-04-24.

[54] B. Radojevic and M. Zagar, "Analysis of issues with load balancing algorithms in hosted (cloud) environments," in *Proceedings of the 34th International Convention MIPRO*, IEEE, 2011, pp. 416–420.

[55] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[56] B. Ruan, H. Huang, S. Wu, and H. Jin, "A performance study of containers in cloud environment," in *Asia-Pacific Services Computing Conference*, Springer, 2016, pp. 343–356.

[57] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on Docker Hub," in *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 269–280.

[58] J. Turnbull, *The Docker book: Containerization is the new virtualization.* James Turnbull, 2014.

[59] VMWare Inc., *Hypervisor*, https://www.vmware.com/topics/glossary/content/hypervisor, Accessed: 2020-05-27.

[60] R. Wadsworth, *Beyond Docker: Other types of containers*, https://www.contino.io/insights/beyond-docker-other-types-of-containers, Accessed: 2020-05-27.

[61] L. Wang and K. S. Trivedi, "Architecture-based reliability-sensitive criticality measure for fault-tolerance cloud applications," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[62] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2017, pp. 138–147.

[63] F. Xu, F. Liu, H. Jin, and A. V. Vasilakos, "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions," *Proceedings of the IEEE*, vol. 102, no. 1, pp. 11–31, 2014.

[64] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the impact of outdated and vulnerable JavaScript packages in Docker images," in *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019, pp. 619–623.

[65] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the relation between outdated Docker containers, severity vulnerabilities, and bugs," in *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019, pp. 491–501.

[66] Z. Zheng and M. R. Lyu, “Selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints,” *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 219–232, 2013.

[67] Z. Zheng, M. R. T. Lyu, and H. Wang, “Service fault tolerance for highly reliable service-oriented systems: An overview,” *Science China Information Sciences*, vol. 58, no. 5, pp. 1–12, 2015.