Bachelor Thesis

March 22, 2017

An Empirical Analysis of the Docker Container Ecosystem on GitHub

Mining Software Repositories to Identify the State and Evolution of Docker Containers

Sali Zumberi

of Zürich, Switzerland (13-790-951)

supervised by

Prof. Dr. Harald C. Gall

Jürgen Cito and Gerald Schermann

Bachelor Thesis

An Empirical Analysis of the Docker Container Ecosystem on GitHub

Mining Software Repositories to Identify the State and Evolution of Docker Containers

Sali Zumberi

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis
Author: Sali Zumberi, sali.zumberi@uzh.ch
Project period: 22.09.2016 - 22.03.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Dr. Harald C. Gall, the software evolution and architecture lab of the University of Zurich for the possibility to delve into a versatile topic, leading to interesting insights. Furthermore, I would like to thank my supervisors Jürgen Cito and Gerald Schermann for their constructive inputs and feedback during the process of writing the bachelorthesis.

Abstract

Docker is a leading software container platform which has become increasingly popular in recent years. It allows to package an application with its dependencies into a standardized, self-contained unit called a container. Containers thus simplify the provision of applications because they contain all the necessary packages and are easy to transport and install. They can be used in software development to automate repetitive tasks, such as deploying and configuring development environments, or simply launching software on any system. Docker provides his own declarative scripting language (written in Dockerfiles) to create images for such containers. It consists of declarative definitions and refers to the practice of scripting the provisioning of hardware and operating system requirements concurrently with the development of the software itself. With a Dockerfile, a container image can be created repetitively without manual installation or configuration steps. Therefore, they can often be found in respective source code repositories. This Bachelor thesis presents an exploratory empirical study with the goal to characterize the Docker ecosystem, analyze the evolution of Dockerfiles, and lastly identify prevalent quality issues. It is based on a data set of 97'571 Dockerfiles, which are divided into three different categories to contrast them: the entire data set, the top 100 and the top 1000 most popular Docker-using projects. Most of the quality issues (28.6%) arise from missing version pinning (i.e., specifying a concrete version for dependencies). The most popular projects change more often than the rest of the Docker population, with 4.95 revisions per year and 5 lines of code changed on average. Most changes deal with dependencies which are currently stored in a rather unstructured manner. Furthermore, a rising trend of using lightweight images, for instance Alpine, has been observed.

Zusammenfassung

Docker ist eine führende Software-Container-Plattform, die in den letzten Jahren stark an Popularität gewonnen hat. Sie erlaubt, eine Anwendung mit ihren Abhängigkeiten in eine standardisierte, in sich geschlossene Einheit zu verpacken, die als Container bezeichnet wird. Container vereinfachen somit die Bereitstellung von Applikationen, da sie alle notwendigen Pakete enthalten und einfach zu transportieren und zu installieren sind. Sie können in der Softwareentwicklung eingesetzt werden, um repetitive Aufgaben zu automatisieren, wie z. B. das Implementieren und Konfigurieren von Entwicklungsumgebungen oder simpel das Starten von Software auf jedem System. Docker stellt seine eigene Skriptsprache (geschrieben in Dockerfiles) zur Verfügung, um Images für solche Container zu erstellen. Es besteht aus deklarativen Definitionen und bezieht sich auf das Scripting der Bereitstellung von Hardware- und Betriebssystemanforderungen parallel zur Softwareentwicklung selbst. Mit einem Dockerfile kann ein Containerimage ohne manuelle Installations- oder Konfigurationsschritte wiederholt erstellt werden. Daher können sie oft in entsprechenden Quellcode-Repositories gefunden werden. Diese Bachelor-Arbeit präsentiert eine explorative empirische Studie mit dem Ziel, das Docker-Ökosystem zu charakterisieren, die Evolution von Dockerfiles zu analysieren und letztlich vorherrschende Qualitätsprobleme zu identifizieren. Es basiert auf einem Datensatz von 97'571 Dockerfiles, welcher in drei verschiedene Kategorien unterteilt wird, um diese gegenüberzustellen: den gesamten Datensatz, die Top 100 sowie die Top 1000 der meist verbreiteten Docker-basierten Projekte. Die meisten Qualitätsprobleme (28,6%) ergeben sich aus fehlender Versionsvereinigung (d.h. Angabe einer konkreten Version für Abhängigkeiten). Die populärsten Projekte ändern sich öfter als der Rest der Docker-Bevölkerung, mit 4,95 Revisionen pro Jahr und 5 Zeilen Code, die im Durchschnitt geändert werden. Die meisten Änderungen befassen sich mit Abhängigkeiten, die derzeit eher unstrukturiert gespeichert sind. Darüber hinaus wurde ein steigender Trend der Verwendung von leichten Images, wie zum Beispiel Alpine, beobachtet.

Contents

List of Figures

List of Tables

List of Instructions

Chapter 1

Introduction

Containerization, also known as container-based virtualization is an OS-level virtualization method [70] for packaging, deploying and running applications without launching an entire Virtual Machine for each application. Instead, multiple isolated systems, called containers, are run on a single control host and access a single kernel [40]. Containerization has gained interest as a light-weight virtualization technology to define software infrastructure [41] [25]. Containers are easily packaged, lightweight and designed to run anywhere. Containers encapsulate discrete components of application logic and are easily packaged, lightweight and designed to run anywhere. There is only one underlying OS which takes care of the incoming hardware calls. The more agile environment facilitates new approaches, such as microservices and continuous integration and delivery. Due to their rapid gained prominence in the software development community, the Docker container format and runtime will form the basis of the new standard of Containers [48] [49]. Docker allows, in addition to the Command Line Tool, to build Containers automatically by reading the instructions from a Dockerfile. The content of the *Dockerfile* is declaratively defined and includes *Dockerfile* instructions as layers to reach a certain infrastructure state [30], following the notion of Infrastructure-as-Code (IaC) [32]. Docker supports fully automated builds for both public and private repositories as well as for Github and Bitbucket [13].The whole Dockerfile context can be executed with only one command: *docker build* and it will immediately build the isolated infrastructure. Since its imposition in 2013, repositories on Github have added 108560 Dockerfiles to their projects (until Mars 2017).

1.1 Contribution

Containers are changing the way of how software developers build, maintain and monitor applications [1] and last but not least, its surrounding claim of enabling reproducibility [4]. This thesis analyses the Docker ecosystem in terms of quality and their change and evolution behavior within software repositories. In order to get a clear overview of the Dockerfiles it was crucial to develop a tool chain that translates Dockerfiles and their evolution in Git repositories into a relational database model. Instead of selecting a sample of projects with Dockerfiles, the mentioned framework mined the entire population of Dockerfiles on Github as of March 2017.

The results of this thesis can help standard bodies around containers and tool developers to develop better support in order to improve the quality and drive ecosystem change, yet to adjust existing tools based to the provided information and data of this thesis.

This thesis makes the following contributions with the empirical analysis:

Ecosystem Overview. Characterization of the ecosystem of Docker containers on Github by analyzing the mined data. There section is divided into two parts.

First, it examines the meta-information such as distribution of projects using Docker, broken down by primary programming language, project size, and the base infrastructure (*base image*) they inherited from.
Then, a drill down into the instructions and closer look at the commands which were used such as the exposed ports, used packages, most commented instruction and many more. The results show, among other things, that most inherited base images are well-established, but heavy-weight operating systems, while light-weight alternatives are in the minority.
The reason may lay in the convenience to use already existing, known systems and their established package managers. Additionally,it is easier to move legacy applications into containers without re-constructing them.
Moreover, the analysis shows that there is a difference regarding the distribution of the followed instructions depending on the base image.

Evolution Behaviour. Investigation of evolutionary aspects such as rate of changes, size of changes, dependencies along the evolution of commits and more. There was a need to refine the existing Diff-Tool of Git to increase the accuracy between two Commits. Finally, the Diff Processor was able to perform classification of different kinds of changes between consecutive versions of Dockerfiles. On average, Dockerfiles only changed 3.11 times per year, with a mean 3.98 lines of code changed per revision. Nonetheless, there is evidence that more popular projects (higher popularity, measured by metadata attributes) revise up to 5.81 per year with 5 lines changed.

Quality Assessment. Static quality Assessment of Dockerfiles on Github by classifying results of a Dockerfile Linter [37], which performs static code analysis by parsing the Dockerfile into an AST and performing rules on top of the AST [38]. Most of the issues encountered considered version pinning (i.e., specifying a concrete version for either base images or dependencies), accounting for 28.6% of quality issues.

This thesis makes following additional contributions:

1. It contains the tooling, data model, and mined data for 100k Dockerfiles and their evolution as of October 2016
2. It provides a tool chain that parses Dockerfiles and stores it in a relational model. Furthermore it (*dockalyzer*) captures structured evolution between subsequent versions in the version control history
3. It provides the entire database (*37 GB*), queries and analysis scripts.

1.2 Outline

The remainder of this paper is divided into five chapters.

Chapter 2 *Background* provides the basic background on containerization technologies, Docker and the associated Dockerfile as well as a brief insight in git and repository mining.

Chapter 3 *Related Work* gives a short and comprehensive view over previous research on this field.

Chapter 4 *Approach* describes the tool chain, implementation details and the resulting data set.

Chapter 5 *Results* shows the main results of the analysis, seperated in four sections. Firstly an overview of the Ecosystem, secondly the dependencies along the Packages, followed by an evolutionary insight. The last part describes a quality assessment.

Chapter 6 *Threats to Validity* presents a discussion of the threats of validity of this thesis.

Chapter 2

Background

2.1 Virtual machines vs. Containers

2.2 Containers and Docker

In computing, a virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer [62]. Containers are based on an OS-level virtualization technique that provides virtual environments that enable process and network isolation. LXC provides operating system-level virtualization through a virtual environment that has its own process and network space, instead of creating a full-fledged virtual machine. LXC relies on the Linux kernel cgroups functionality that was released in version 2.6.24. It also relies on other variations of namespace isolation functionality, which were developed and integrated into the mainline Linux kernel.

2.3 Docker

Docker is built on top of LXC and adds image management and deployment assistance to virtualize applications. Docker provides an automation and rapid provisioning of LXC cgroups without requiring a VM. Docker provides an API that extends the functionality of LXC for building Platform as a Service (PaaS) offerings [56]. It aims to improve reproducibility of applications by enabling bundling of container contents into a single object which can be deployed across machines [9]. Docker containers allow you to put the application and dependencies together and isolate from other containers and make it easily portable from one environment to another. The Docker image is the building block for these application containers - these images are built, shared, updated and then deployed as containers [29].

2.4 Dockerhub

Docker Hub is a cloud-based registry service which allows users to link code repositories, build images and test them, stores manually pushed images and links them Docker Cloud so can deploy images to their hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline. The Docker Official Repositories are a curated set of Docker repositories that are promoted on Docker Hub [14].

2.5 Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. It is a similar concept as the recipes and manifests found in infrastructure automation (IA) tools like Chef[1] or Puppet[2]. Using docker build users can create an automated build that executes several command-line instructions in succession [14].

2.1 Listing (INSTRUCTION)

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily [14]. This subsection provides a brief explanation

of some of the available instructions to provide a basic understanding of the format for the remaining thesis. **FROM** The first instruction must be 'FROM' in order to specify the Base Image

from which Docker is building.

2.2 Listing (FROM Instruction)

```
# Example
FROM ubuntu
```

MAINTAINER(deprecated) Before: This is a non-executable instruction used to indicate the author of the Dockerfile. The LABEL instruction should be used in favor, it is a much more flexible version.

2.3 Listing (MAINTAINER Instruction)

```
Before
MAINTAINER <name>
Now
LABEL maintainer "sali.zumberi@uzh.ch"
```

RUN This instruction allows to execute a command on top of an existing layer and create a new layer with the results of command execution. It is often used to retrieve dependencies and install and compile software.
For example, if there is a pre-condition to install JAVA before running an application, then an appropriate command to install JAVA on top of base image can be executed like this:

2.4 Listing (RUN Instruction)

```
FROM ubuntu
RUN apt-get update && apt-get install java
```

[1] https://www.chef.io/chef/
[2] https://puppet.com/

CMD The major difference between CMD and RUN is that CMD doesn't execute anything during build time. It just specifies the the intended process for the image. Whereas RUN actually executes the command during the build time.

There can be only one CMD instruction in a Dockerfile, if there are more, only the last one is affective.

2.5 Listing (CMD Instruction)

```
CMD "echo" "Hello World!"
```

LABEL It's possible to assign metadata in the form of key-value pairs to the image using this instruction. It is important to notice that each LABEL instruction creates a new layer in the image, so it is best to use as few LABEL instructions as possible.

2.6 Listing (LABEL Instruction)

```
LABEL version="1.0" description="This is a sample desc"
```

EXPOSE A Container may need to listen on specified ports, while a service is running in the container. The EXPOSE instruction opens these ports.

2.7 Listing (EXPOSE Instruction)

```
EXPOSE 8080
```

ENV This instruction can be used to set environment variables in the container.

2.8 Listing (ENV Instruction)

```
ENV var_home="/var/etc"
```

COPY This instruction is used to copy files and directories from a specified source to a destination (in the file system of the container).

2.9 Listing (COPY Instruction)

```
COPY ubuntu
```

ADD Although ADD and COPY are functionally similar, COPY is preferred from an objective point of view. That is because it is more transparent than ADD. COPY only supports the basic copying of local files into the container, while ADD has some features (like local-only tar extraction and remote URL support) that are not immediately obvious. Consequently, the best use for ADD is local tar file auto-extraction into the image, as in ADD rootfs.tar.xz /.. [12].

2.10 Listing (ADD Instruction)

```
ADD http://www.site.com/downloads/sample.tar.xz /usr/src
```

ENTRYPOINT The ENTRYPOINT Instruction specifies a command that will always be executed when the container starts.

For example, if there is only one application installed within an image that should run whenever the image is executed.
Besides this, all the elements specified using CMD except the arguments, will be overridden. They will be passed to the command specified in ENTRYPOINT.

2.11 Listing (ENTRYPOINT Instruction)

```
CMD "Hello World!"
ENTRYPOINT echo
```

VOLUME The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

2.12 Listing (VOLUME Instruction)

```
FROM ubuntu
RUN mkdir /myvol
RUN echo "hello world" > /myvol/greeting
VOLUME  /myvol
```

This Dockerfile results in an image that causes 'docker run', to create a new mount point at /myvol and copy the greeting file into the newly created volume.

USER This is used to set the UID (or username) to use when running the image.

2.13 Listing (USER Instruction)

```
USER daemon
```

ARG The ARG instruction defines a variable that users can pass at build-time to the builder.

2.14 Listing (ARG Instruction)

```
ARG buildno
```

WORKDIR This instruction is used to set the currently active directory for other instructions such as RUN, CMD, ENTRYPOINT, COPY and ADD. Note that if a relative path is provided, the next WORKDIR instruction will take it as relative to the path of previous WORKDIR instruction.

2.15 Listing (WORKDIR Instruction)

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

This will output the path as /a/b/c.

ONBUILD This instruction adds a trigger instruction to be executed when the image is used as the base another image. It behaves as if a RUN instruction is inserted immediately after the FROM instruction of the downstream Dockerfile. This is typically helpful in cases where a static base image is needed, with a dynamic config value that changes whenever a new image has to be built (on top of the base image).

2.16 Listing (ONBUILD Instruction)

```
# Example
ONBUILD RUN rm -rf /usr/temp
```

Dockerfiles provide a simple and easy way to create Docker images with very minimal programming effort required.

2.6 Version control system and Repository Mining

2.6.1 Git and Github

The Java based repository miner and Analyser used algorithms based on Git and loaded Software Repositories from Github.

Git is a version control system (VCS) for tracking changes in computer files and coordinating work on those files among teams. It is especially used for software development [55], but it can also be used to keep track of changes in any text files. As a distributed revision control system it is aimed at speed [64], data integrity [66] and providing support for distributed workflows [65].

Github is a web-based Git or version control repository and Internet hosting service. It offers all of the distributed version control and source code management (SCM) functionality of Git. It provides features like access control and several collaboration features such as bug tracking, feature requests, task management, comments and wikis for every project [23].

GitHub offers both plans for private and free repositories on the same account [24] which are commonly used to host open-source software projects [36]. As of April 2016, GitHub reports having more than 14 million users and more than 35 million repositories, thus making it the largest host of source code in the world [21].

2.6.2 Repository Mining

Mining software repositories (MSR) is an important area of research.The main purpose is to analyze the rich data available in software repositories to uncover interesting and actionable information about software systems, projects and behaviours in software engineering. There is also an international workshop on MSR, which has been established under the umbrella of an/the international conference on software engineering (ICSE) in year 2004 [8]. Github makes it possible to clone thousands of projects, therefore it is well suited for repository mining, even if it brings some perlis with it [35].

2.6.3 Automated Deployment Workflows

As mentioned above, Docker Hub offers the ability to connect to Continuous Integration (CI), Delivery (CD) or Git systems to automate the build to test and deploy workflow. Automated builds can be setup from source code management systems such as Github and Bitbucket to build a new image to Docker Hub every time a newly committed code passes the integration test. It ensures that the resulting image is built exactly as specified in the Dockerfile and that the image is up to date, meaning that in each stage of the application development process all your application configurations remain constant. Webhooks allows to connect Hub to CI/CD tools that can automatically run integration tests or deploy every time a new image is pushed to Hub, drastically speeding up the application development, test and production.

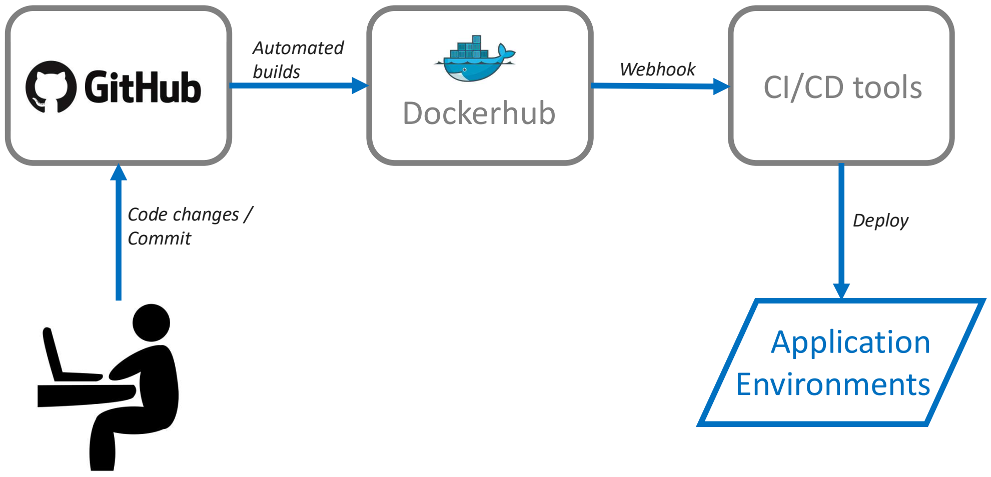


Figure 2.1: Graphical Illustration of an automated deployment workflow

Chapter 3

Related Work

3.1 Ecosystem Analyses

Since the work by Messerschmitt and Szyperski in 2003 [44], research on software ecosystems in a software engineering setting has been around for more than a decade. Since then, empirical analysis of software ecosystems grew in popularity and got an important aspect of software ecosystem research [58]. Serebrenik et al. conducted a meta-analysis by surveying 26 authors of the research field of software ecosystems in which they list the challenges of different topics. Statistics, Visualization and Comparison are the main challenges of an Analysis of Ecosystem [58]. Related work addressed challenges on aggregating data [47], evolution of ecosystems [10] [10],visualization [39] [51] [68], characterizing ecosystem maturity [2] and on how to aggregate software quality metrics [47]. Some works did comparisons between software ecosystems [18] [16] [67]. Jansen et al. looked at ecosystems from different point of view. Alongside technical aspects their work also considered the business facets of software ecosystems, thus they defined software ecosystems as a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. Their findings are collected in an research agenda for software ecosystems [31]. Blincoe et al. analyzed ecosystems in Github and found that most ecosystems are centered around one project and are interconnected with other ecosystems. Following Blincoe et al. the predominant types of ecosystems are those that develop tools to support software development [3].

3.2 Empirical Analyses over specific Ecosystems

There are also many Empirical Analysis of software ecosystems that involve around a specific framework or programming languages. Wittern et al. conducted an empirical analysis of the JavaScript package ecosystem, one of the largest software ecosystems. Their work confirmed npm[1] to be a striving ecosystem with ongoing and even accelerating growth of packages and increasing dependencies between them [69].

A work from Raemaekers et al. present a dataset containing basic metrics, dependencies, and changes with some aggregate statistics about Maven[2], a popular package manager for Java [52].

Another work regarding Maven runs *FindBugs*, a tool that examines Java bytecode to detect numerous types of bugs and identifies bugs in the source code of libraries shared in the same ecosystem. The dataset was obtained from the Mavencentral repository ecosystem [45].

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

[2]https://maven.apache.org/

Daniel M Germán et al. did an analysis of the statistical computing project R[3] [22] which shows a super-linear growth with a strong set of core packages.

An other work enhanced this analysis and tried to find out how an ecosystem evolves. Their finding show that the success of the R ecosystem relies on a strong commitment by a small core of users who support a large and fast growing community [50] .

Another paper presents an overview of the open source Ruby[4] ecosystem and lists its elements, characteristics, descriptives, roles and relationships. They gathered the needed data using the Git decentralized source code management system and applied social network and statistical analysis techniques for their analysis [34]. The paper provides a graph visualization of the whole ecosystem as well as some more descriptive graphical representations about selected characteristics of packages, including downloads and package size. Their analysis shows that the Ruby ecosystem exists out of a couple very distinctive roles developers fulfill. It also shows that within the Ruby ecosystem only a small 'core' of approximately 10% of all developers and gems (Ruby packages) are dominant within the ecosystem, similar to the finding of R ecosystem [50].

Grechanik et al. [27] did research on the structural characteristics of the source code of 2080 randomly chosen Java open source projects. To do so he answered 32 research questions related to classes and packages, constructors and methods, fields, statements, exceptions, variables and basic types, and evolution/maintenance activities occurring on the projects.

3.3 Analysis of IaC Ecosystems

The Internet-speed of things has changed the way software companies deliver value to their customers. The boundless adoption of lean principles and agile methodologies [54] has provided evidence for the need of continuous activities which are important for software development in today's context [20].

Related work on modern software engineering concepts such as continuous deployment [53], cloud computing [7] and containerization increasingly more significance.

McIntosh et al. conducted work on build systems such as the Java-based systems. They studied the evolution of build systems based on two popular Java build languages (i.e., ANT and Maven) from two different perspectives: (1) a static perspective, where they examined the complexity of build system specifications using software metrics adopted from the source code domain; and (2) a dynamic perspective, where the complexity and coverage of representative build runs are measured [42].

Later, they published a paper where they studied low-level, abstraction-based, and framework-driven build technologies as well as tools that automatically manage external dependencies. Their findings are that modern and framework-driven build technologies need to be maintained more often and these build changes are more tightly coupled with the source code than low-level or abstraction-based ones. However, build technology migrations tend to coincide with a shift from build maintenance work to a build-focused team, deferring the cost of build maintenance to them [43].

Jiang and Adams have analyzed the co-evolution of the IaC code with the production and test code, and compared them with build files [32]. Their findings are that infrastructure files are large and churn frequently, which could be an essential factor for introducing bugs. Furthermore, they found out that the infrastructure code files are coupled tightly with the other files in a software project, especially test files, which implies that testers often need to change infrastructure specifications when making changes to the test framework and tests.

[3]https://www.r-project.org/
[4]https://www.ruby-lang.org/

So far, there is only little research on how to develop and maintain IaC code. Related work in this field has been shown by Hummer et al.. They proposed and evaluated a model-based testing framework for IaC. An abstracted system model will be utilized to derive state transition graphs, based on systematically generated test cases for the automation. The test cases are executed in light-weight virtual machine environments. Their prototype targets the popular IaC tool (Chef[5]), but the proposed approach is general.

Existing empirical studies related especially to Docker typically focus more on performance aspects, in most of the cases there is a comparison of container performance and overhead with traditional virtualization techniques [19,46,59,60].

Also a very interesting usage of Docker is mentioned by Cito et al. and Boettiger et al.. They have concurrently proposed that containerization technology may be an important game changer in making systems and software engineering research more reproducible by using Docker containers to distribute reproducible research. This should be seen as an approach that is synergistic with, rather than an alternative to, other technical tools for ensuring computational reproducibility [4,9].

In industry, the usage of containers in PaaS is becoming mainstream, and the Linux Containers are becoming a defacto standard but still have a long way to go before widespread adoption. Some of the Cloud Providers are already using Containers to provide a fine-grained control over resource sharing [15]. Following Dua et al. containers have a bright future especially in the PaaS use case, provided there is more standardization and abstraction from the underlying kernel and Host OS.

Cito et al. conducted research on the quality aspects of Docker IaC code. The dataset and tooling for these analyses is based on this thesis. A randomly selected representative sample of Dockerfiles of 560 repositories was built. According to their measurements 66% of Dockerfiles could be built successfully with an average build time of 145.9 seconds, while 34% of builds failed in 90.5 seconds. Following Cito et al. integration of quality checks into the "docker build" process to warn developers early about build-breaking issues, such as version pinning, which can lead to more reproducible builds [33].

No existing research has investigated in an empirical Analysis on the Docker Ecosystem, the evolution of Dockerfiles, dependecies within the containers or empirical studied about the open source ecosystem of Docker. The present thesis tries to show an overview of the ecosystem, dependencies, maintenance and evolution and quality aspects and wants to provide useful information and data for further research.

3.4 Mining Software Repository

Providing or offering software repository data is not new field and can be seen as related to "Data as a Service" or "Information as a Service" [11]. The data being provided was usually limited to the meta-data or elements of the repository such as files [26]. Github is a repository of repositories [23], therefore its elements are repositories themselves. As opposed to existing meta repositories such as OHLOH or FLOSSMOLE [28], lean GHTorrent[6] allows researchers to request a specific slice of the full GHTorrent dataset, on a per repository basis. All a researcher has to do is compile a list of repository names with the support of the provided MySQL query interface, rather than being forced to analyse the entire collection searching for the proverbial needle [26]. Services like BOA integrate the repository analysis tasks in the web-based interface [17]. This thesis uses an approach, which is not cited or mentioned in related work yet. It uses Google BigQuery[7] to

[5]`https://www.chef.io`
[6]http://ghtorrent.org/lean.html
[7]https://cloud.google.com/bigquery/what-is-bigquery

get the specific repositories.

Chapter 4

Approach

This chapter describes the end-to-end process of the whole application, which supports the mining, processing, analysing and extracting phase of this thesis. After a short overview of the overall system architecture, it shows the resulting data model together the data processing section, where it presents the conceptual logic of the various important computations, such as the parser, which transforms an given Dockerfile to a structural object and outputs it as a json File. Furthermore, it examines the evolution parser, which takes an software project as input and processes through the commit history and finally outputs all information regarding a Dockerfile to a database. This section also includes a self written Diff Processing Tool, which has been a necessary requirement for achieving a very high data quality. The last chapter containts some brief information about the implementation.

One goal of this thesis is to compare Docker usage in the general population of GitHub projects to how the tool is used in outstanding projects. Therefore, this thesis introduce two additional samplings in the database: *Top-100* and *Top-1000* repositories containing Dockerfiles. To retrieve these two samples, the repositories were ordered by the number of star gazers (a measure for popularity) and selected 100 and 1000 unique repositories respectively. In order to foster reproducibility and follow-up studies, this thesis provides a comprehensive reproducibility package, containing tool chain, database, queries and analysis scripts. The tool chain consists of two separate Java projects. Docker Parser [72] is responsible for parsing and storing Dockerfiles in a relational database (Postgres[1]), and Dockolution [71] computes structural changes between all revisions in a repository. The Entire database(*37 GB*) is exported and available on our university's archive server[2]. All of this analyses (SQL queries to the database, R and Python scripts to produce plots) are also available on GitHub[3] for inspection and reproduction.

4.1 Dataset

To enable this research, it was necessary to retrieve a list of repositories that contain Dockerfiles from the public GitHub archive on BigQuery[4] in January 2017. BigQuery hosts more than 2.8 million open source projects on Github since the middle of 2016 [61]. GitHub's BigQuery dataset is based on the GitHub Archive Project, a project that aims in taking snapshots of GitHub at specific points in time, storing and making them accessible for everyone. It contains about 1.5TB of data. The observation period for revisions and changes that was mined to analyze evolution behavior

[1]https://www.postgresql.org/

[2]Database Export Link omitted for double blind review

[3]Queries and Scripts GitHub Link omitted for double blind review

[4]`https://cloud.google.com/bigquery/public-data/github`

was from the first Dockerfile commit that appeared in the repository until January 2017. The initial list contained over 324255 Dockerfiles in 78723 GitHub projects. It was required to query the needed files from the public github archive. The provided csv File consisted of tuples, each tuple being associated with two attributes: Repository Name and path to the Dockerfile. Instead of rebuilding the .git url from the existing repository name, it was necessary to send request to the GitHub API to get additional metadata for each project of the list, such as the owner type, owner name, used programming languages, project size, number of forks, issues, or the number of watchers. Due the restriction of 5000 requests/h, the Github API Miner changed the User after the limited rate was exhausted. After a first assessment of the consolidated data, it was crucial to remove repositories that were forks from other repositories to avoid biasing our analysis with duplicate entries. Almost two-thirds of the initial project list were concerned. This would have been particularly problematic, as especially large, popular projects such as Kubernetes[5] nginx[6] or docker/docker itself are forked very frequently. Therefore the first behavioural pattern was identified. Developers apt to fork big and popular projects. The reason is that big projects offer a lot of predefined basic and also advanced Dockerfile templates. The resulting study population for this analysis consisted of 97571 unique Dockerfiles originating from 48102 GitHub projects.

4.2 Overall System Architecture

This section presents the overall system architecture of the Java-based tool chain (see Figure 4.1.It consists of three different kinds of components and supports and facilitates the entire process from the raw data to the final data visualizations. The three different kinds are the following:

1. Data Mining from Google BigQuery and expansion with GitHub API assistance
2. Multi-threaded Data extraction and Processing
3. Output to Database and Data analysis

It starts with enhancing the data from BigQuery with additional information, by using the Github API, as described in the section above. With this meta-data it was possible to inspect the first insights of the docker ecosystem, which are related to the given project meta-data. The second part dives into the project content itself by feeding the consolidated data into main application. In addition to that, it was also necessary to set a number of threads as parameter. Out of performance reasons the University of Zurich provided a Linux Server to increase the performance, which allowed to start 50 threads at once. Each thread cloned a Project Repository first and then continued with searching the Blob[7] for the Dockerfile, which represents the content of a file in a tree. Then it iterates above the entire tree and gathers the associated Revisions to a Collection, which finally represent the History of that File. Index 0 represents the first Revision.

It continues by iterating though this collection and reproducing for each Revision the corresponding Dockerfile. Each Dockerfile is parsed to an Object. Furthermore it adds information about the used base image in the From Instruction by using the Dockerhub Api [8] such as rating stars, or whether it is a official or automated build. Then it starts enhancing the main Object, which contains the mentioned Collection of Dockerfile Revision. It simply adds the project metadata to it and also performs a statical code analysis of the latest Revisions by parsing the Dockerfile into an AST and performing rules on top o the AST. Finnally, the Evolution Tracker goes through the list of Revisions and gathers for each of them the changed files within that revision.

[5]https://github.com/kubernetes/
[6]https://github.com/nginx
[7]https://git-scm.com/book/en/v2/Git-Internals-Git-Objects
[8]https://docs.docker.com/v1.4/reference/api/docker-io_api

It uses a certain range before and after the revision. The range in this thesis is set to three. That means it gathers three revisions before and after the corresponding Dockerfile revision. Lastly the Diff Processor, compares two Revisions and identifies the differences between them. This thesis categorized the different types of differences (*Change Type*): ADD, MODIFY and DELETE with subcategories for each instruction, which enables a much simpler and more fine-grained evolution analysis. The final Object is then persisted with a synchronized method to the database. Figure 4.1 visualizes the declared process.

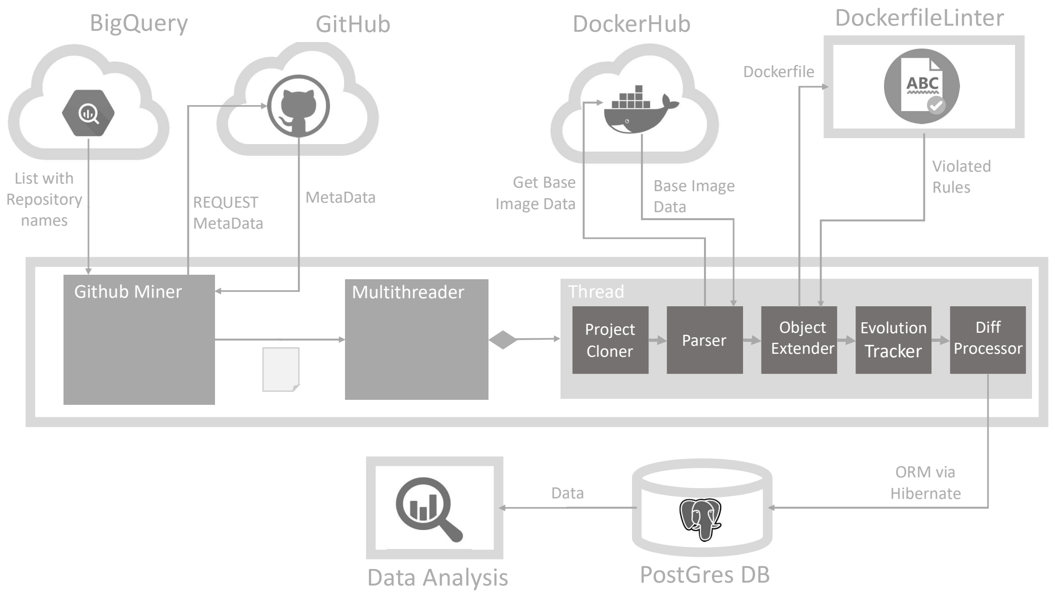


Figure 4.1: Overall System Architecture

4.3 Datamodel

Figure 4.2 shows the Entity Relationship of the data model. Each project on GitHub stores one to many *Dockerfile* entities that contain metadata about the repository and the file such as number of forks, network counts, open issues, owner type, project size in KB etc. Our database consists of (*97'571 Dockerfiles* distributed across *48'102 projects*). Each Dockerfile has one to many violated rules*381'439*. Each *Dockerfile* contains one to many *Snapshots* (Revision with additional information) (*347'904 revisions*), which reflect every commit on this Dockerfile. The initial commit that adds a Dockerfile to a GitHub project is also modeled as a revision, hence every Dockerfile has at least one revision. Each Snapshot needs at least one Instruction to be valid. There are two types of Instructions: Single Instructions with a one to one relation and Instructions, which can appear multiple times in a Dockerfile as one to many relation. Instructions like CMD, ENTRYPOINT and RUN also have one to many parameters (*7'598'453*). For two consecutive snapshots (before and after a change) of Dockerfiles, the Diff processor (see Section 4.4.3) compute structural differences, and stores the entity *Diff* (*351'389 diff objects*) with one to many *Diff Type* entities for each instruction (*2'557'047 changes*). A Diff Type is a structured change.

Table 4.3 presents a short description for the non-trivial attributes:

table	attribute	description
FROM	CURRENT	boolean: if snapshot is the lastest one
	FULL_NAME	full image name (e.g. python:2.7)
	IMAGENAME	only image name (e.g. python)
	IMAGEVERSION	only image version (number) (e.g.2.7)
	IMAGEVERSIONSTRING	only image version (string)
	DIGGEST	only version tag
COMMENT	INSTRUCTION	4 different cases: 1. standalone: does not belong to an instruction 2. before: name of instruction, which this comment belongs to 3. commented out: name of instruction, which is commented out 4. header: header of dockerfile
	COMMENT	content
RUN	EXECUTABLE	exec (e.g. java)
	RUN_PARAMS	summary of all parameters (e.g. -jar -test)
DOCKERFILE	REPO_PATH	repository name
	DOCKER_PATH	path to Dockerfile
	CREATED_AT	project creation date
	FIRST_DOCKER_COMMIT	date when dockerfile was added the project (extracted from corresponding revision
	COMMITS	number of revisions
SNAPSHOT	FROM_DATE	publishing date of the revision
	TO_DATE	date of next revision
	IMAGEISAUTOMATED	boolean: if base image is build automatically from github
	IMAGEISOFFICIAL	boolean: if base image is an official dockerhub build
	STARCOUNT	rating starts of the base image in dockerhub
CHANGED_ FILES	CHANGETYPE	3 types: - ADD - DELETE - MODIFY
	INSERTIONS	insertions made in this file
	DELETIONS	deletions made in this file
	RANGE_INDEX	actual range index (commit before or after dockerfile revision. - 0 is the revsion, where the dockerfile was changed; - 1 is one revision after the dockerfile commit; - 1 is one revision before)
	RANGE_SIZE	inspection range (e.g. 6: three revisions before and thre after the dockerfile revision, which is assigned to 0)

Table 4.1: Description of non-trivial attributes

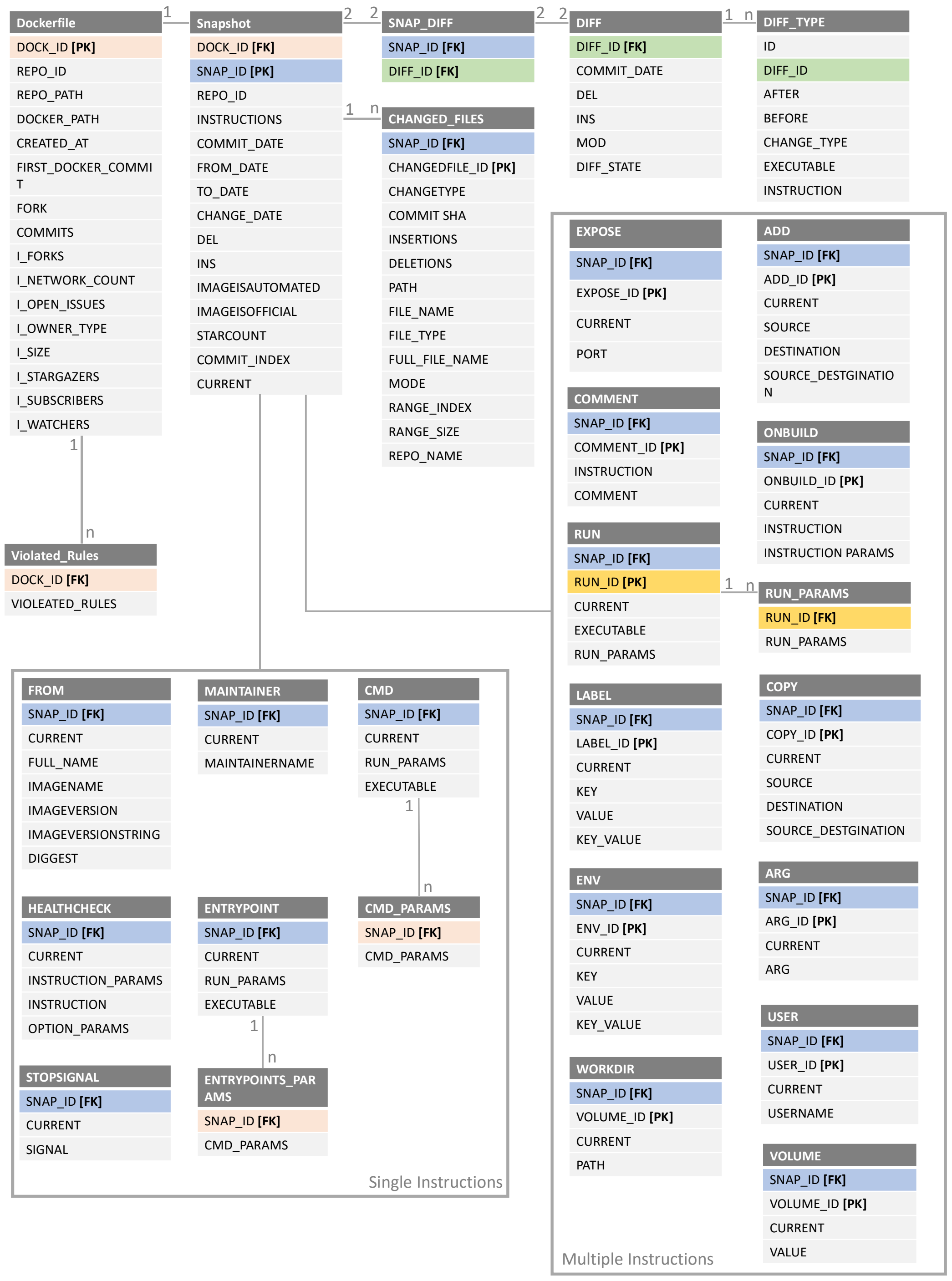


Figure 4.2: Relational data model to support queries for data analysis.

4.4 Data Processing

This section presents the basic concepts behind the tool chain and can be used for similar projects.

4.4.1 Dockerfile Parser

Dockerfiles is very simply constructed and consists of declarative instructions. An Instruction starts with a new line and should be the first letter (see more information in section 2.5. The instruction is not case-sensitive. Therefore, it is very well suited for parsing it to an POJO[9]. The decision to parse the files is based on the fact that it facilitates the mapping to a relational database also as well as an higher data accuracy. Furthermore, it allows to do more precise data analysis due the advanced level of details (see Figure4.2). Before the parsing process starts, the file is "flattened" by transforming multiple instructions, which claim multiple lines, to one line. This Example shows how this looks like:

4.1 Listing (Flattening of an Instruction)

```
RUN service postgresql start & & \
su postgres -c "createuser -d -r -s root"  # Can also be written as:

RUN service postgresql start & & su postgres -c "createuser -d -r -s root"
```

Then the parsing process starts from top to bottom by reading the first word of each line and recognizing the corresponding instruction. Each instruction has a method which includes the parsing logic. There are two different kind of Instructions, single instructions occur once and other multiple times, thus, it was necessary to use Collections for them. Another minor difficulty was the form of the declared attributes. There are two possible syntax forms. Shell form and Exec form. This increases the complexity by adding another condition on top of the parsing method.

4.2 Listing (Shell and exec syntax)

```
# shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd
/S /C on Windows
 RUN <command>

# exec form
RUN ["executable", "param1", "param2"]
```

Lastly, another special case were instructions with parameters such as CMD, ENTRYPOINT and RUN. Therefore, it was necessary to collect them in lists with a many to one relation to the instruction. Finally the parsed object was used in the further processes. The Dockerfile Parser is also available as an independent application, which creates a json file.

4.4.2 Evolution Tracker

Each Snapshot contains all changed files within a specified range size. Figure 4.3 presents a visualization of range index and size. This thesis uses range index six, which leads to *144'938'718* database entries. The application takes the range size as a input argument.

[9]https://en.wikipedia.org/wiki/Plain_old _Java _object

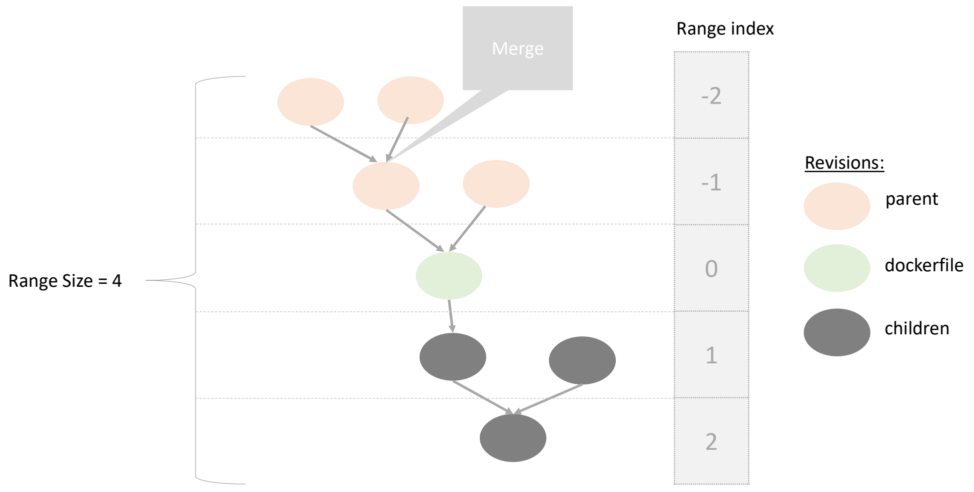


Figure 4.3: Example of Range index with Range size=4

In order to collect the files, it was required to walk through the commit tree. This Algorithm 1 shows the implemented concept of this tree-walker:

Algorithm 1 Collect changed files for $rangeIndex = x$

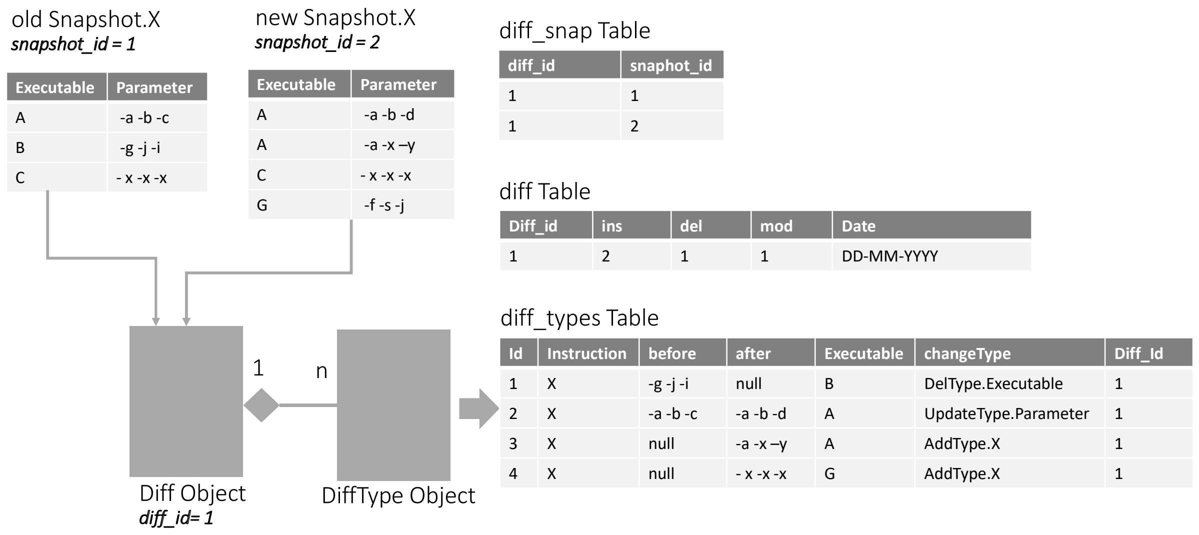
Input: dockerCommit c, rangeIndex x
Output: list of changed files

```
  $list\ tempCommits \leftarrow empty$
  $list\ allCommits \leftarrow git.repository.getAllCommits()$
  $list\ changedFiles \leftarrow empty$
  $boolean\ foundCommit \leftarrow false$
  finder:
  if $x < 0$ then
    $index \leftarrow 0$
    for $commit$ of $all\_commits$ do
      if $commit$.getID = $c$.getID then
        add $commit$ to $tempCommits$
        while not $foundCommit$ do
          if $index$ = x then
            for $tempcommit$ of $tempCommits$ do
              $changedFiles \leftarrow getFilesFromCommit(tempcommit)$
            end for
            $foundCommit \leftarrow true$
            break finder
          else
            $list\ parentCommits \leftarrow empty$
            for $tempcommit$ of $tempCommits$ do
              for $tempcommit$.getParents() do
                add $parent$ to $parentCommits$
              end for
            end for
            $list\ tempCommits \leftarrow parentCommits$
          end if
          $decrement\ index$
        end while
      end if
    end for
  else if $x > 0$ then
    list childrenCommits $\leftarrow$ empty
    walker:
    for $childofallCommits$ do
      for parent of child.getParents() do
        for $tempcommit$ of $tempcommits$ do
          if parent.getId = $tempcommit$.getID then
            add $child$ to $childrenCommits$
            $increment\ index$
            break walker
          end if
        end for
      end for
    end for
    $list\ tempCommits \leftarrow childrenCommits$
  else
    $changedFiles \leftarrow$ getFilesFromCommit($commit$)
  end if
```

4.4.3 Diff Processor

After retrieving all changed files within a Revision, the key processing step begins. The idea behind the implemented Diff Processor is that the provided version control system of git is not precise enough for the applied use cases, for instance inaccurate and wrong diffs due the white spaces, new lines etc. All the information is structured, since all Versions of a specific dockerfile are transformed into Snapshot objects. Therefore, it was straightforward to compare the values of two adjacent snapshots. This means, that two snapshot will have one common Diff Object. Again, it has to be considered that there are two kinds of instructions. Instructions, which can appear multiple times, have to be matched before the comparison. First, it removes the intersection between two lists, then it recognizes the new state, then it has to be checked either if its new or has been deleted. If that is the case, then the corresponding change Type is assigned to it. The remaining changes are assigned as update change types and those types are mapped to two value pairs, where the first value is the old and the second is the new one. There are three different Change Types: ADD, DELETE, and UPDATE, and each of them has further distinctions. In total there are more than 20 different change types, for instance if a version number has been updated to the FROM instruction, the database entry will be: `UpdateType.imageVersionNumber`. Therefore, almost every type of change is recorded. The most difficult part was the mapping of the parameters in instructions like RUN, CMD or an Entrypoint. Thus, a scoring model decides the mapping by counting the identical parameters, for instance if the old snapshot has an instruction: `RUN -a -b -c java` and the new snapshot include two different instructions: `RUN -a -b -d java` and `RUN -a -x -y java`, it would select the first one as an `UpdateType.params` and the second one as a `AddType.run`. It is not certain that the author had exactly applied this changes, but we are sure the result is more confident than the version control system of git, which would show them both as two new added lines.

Figure 4.4 shows a visualized example for instruction X, which can have parameters. X can be a RUN, CMD or an Entrypoint instruction. The right side depicts the corresponding database tables.



Executable	Parameter
A	-a -b -c
B	-g -j -i
C	- x -x -x

Executable	Parameter
A	-a -b -d
A	-a -x –y
C	- x -x -x
G	-f -s -j

diff_id	snaphot_id
1	1
1	2

Diff_id	ins	del	mod	Date
1	2	1	1	DD-MM-YYYY

Id	Instruction	before	after	Executable	changeType	Diff_Id
1	X	-g -j -i	null	B	DelType.Executable	1
2	X	-a -b -c	-a -b -d	A	UpdateType.Parameter	1
3	X	null	-a -x –y	A	AddType.X	1
4	X	null	- x -x -x	G	AddType.X	1

Figure 4.4: Example of an instruction with parameter handled by the Diff Processor

4.5 Implementation

This subsection lists the relevant notes of the implementation. First the project is implemented in **Java**[10] and based on **Maven**, a build automation tool which is used primaly for Java projects. To process this large (1.524 terabytes) amount of data it was necessary to develop a multi-threading approach where each thread clones the project, mines the data and saves the extracted information to a postgres database as described in section 4.1. **Postgres**[11] is an open source relational database. It was necessary to optimize the application to satisfy defined requirements of min. 1000 project/h. In the end it was able to process 2000 projects/h, which makes it more flexible detecting mistakes in the collected data. The other side of multi-threading in Java led to Heap size problems, which is a well known issue [63] . First the entire list with all Dockerfiles was feed into the application, and after 5000 to 6000 finished projects the application heap size was full, this was because jvm used more than 98% for garbage collection and only 2% of the heap was recovering. Therefore, it was crucial to split the project-list to chunks and optimize the chunk size until the best results were achieved. The final size was 1000 projects/chunk. One of the most important libraries used by this application is **JGIT**[12], a pure java library implementing the GIT version control system, but also some methods from **Gitblit**[13] were used. Gitblit is also based on

[10]https://java.com/de/download/
[11]https://www.postgresql.org/
[12]https://eclipse.org/jgit/
[13]https://github.com/gitblit/gitblit

JGIT. **GSON**[14],a Java library that can be used to convert Java Objects into their JSON representation, is used for transforming the parsed Dockerfile Objects into json files. To access the API of Dockerhub it makes use of a fully Java implementation of docker, provided by **Spotify**[15]. This application also uses **super-csv**[16] to assure fast reading and writing of big CSV Files. Finally, it makes also use of **Hibernate** [17], an object-relational mapping (ORM) tool for the Java programming language. The final size of the database is 37 GB. !TEX root = ../thesis.tex

[14]https://github.com/google/gson
[15]https://github.com/spotify/docker-client
[16]http://super-csv.github.io/super-csv/index.html
[17]http://hibernate.org/

Chapter 5

Results

This section discusses the main findings along four dimensions. Firstly the Docker ecosystem is characterized, then a brief insight into the dependencies within a dockerfile will is presented, followed by an overview of the maintenance and evolution of Dockerfiles and to finish the chapter an assessment of quality issues of projects using Docker. At the end of each section, there is a short summary of the conducted findings.

5.1 The Docker Ecosystem

A way to classify the Docker ecosystem is to identify the types of project, which use Docker and to analyze what Docker is used for in these projects. Each sub-section inspects the characteristics of the projects containing Dockerfiles.

5.1.1 Project size Distribution

Figure 5.1 shows histograms of the size of projects in the recorded data set, as reported by the GitHub REST API.[1] As described in Chapter 4, there is a consideration of the overall data set as well as the Top-100 and Top-1000 projects as described. Evidently, projects in the Top-1000 and even more in the Top-100 are of larger size. This fact likely a general effect and must not be necessarily true only for projects using Docker. Of all repositories, an unusual high amount of projects has a size between 130 and 150 KB. Upon manual inspection, many of these repositories seem to consist of solely a readme a license file, and a Dockerfile. This suggests a clear separating of IaC code and software project code. It is very interesting that popular packages have another intended purpose of using Docker, in this case they include the Dockerfile within the software source code whereas as not so popular project use the repository for dedicated purpose. The median project size for all three categories are:

- Top-100 = 19718 KB
- Top-1000 = 5685 KB
- all = 300 KB

There seems to be a correlation between popularity and project size. An explanation for this could be that developers try to separate packaging or deployment concerns from an application.

[1] `https://developer.github.com/v3/`

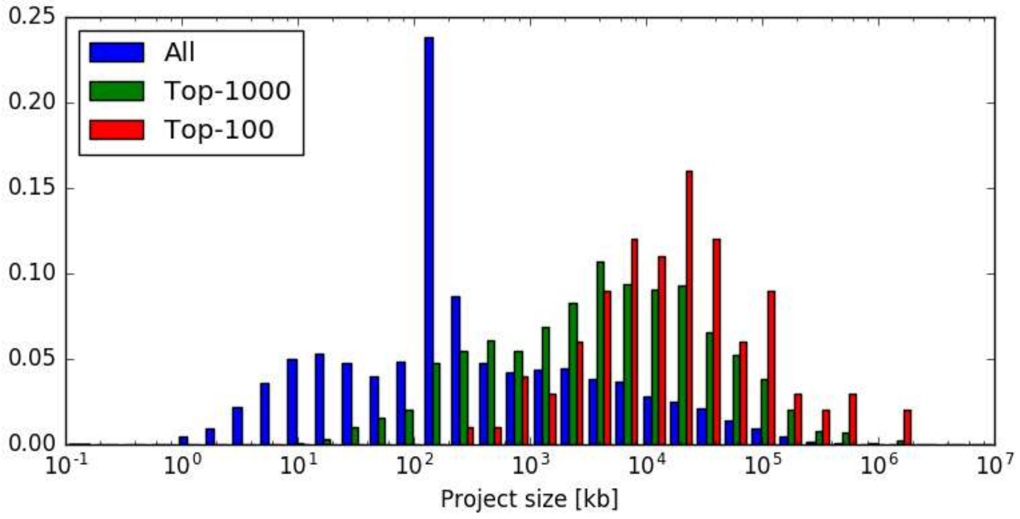


Figure 5.1: Distribution of project sizes

5.1.2 Programming Language Distribution

Figure 5.2 presents the distribution of *primary* programming languages across the checked projects. As almost every project includes more than one language, these statistics refers to the language that accounts for the majority of source code (as measured in file sizes) into consideration. As mentioned in Section 4.1, the data was collected from GitHub's public REST API. In addition to the core data set, Figure 5.2 presents excerpts from the GitHub data available on Google's Big-Query service.[2] The primary programming languages of 2.8 million GitHub repositories have been queried from this public data set. Figure 5.2 shows that Docker projects are frequently dominated by shell scripts, while such repositories in general are less usual on GitHub. A common way to run multiple processes inside a Container is to call a bash script from an instruction like CMD, RUN or ENTRYPPOINT, which is an anti-pattern according to the dockers best practices guide. This could explain why `Shell` is so present.

Languages like `Go` are often used for Docker projects. This can be explained by the fact that a lot of open source tools for Docker are written in `Go`, for instance Kubernetes. This was one of the reasons for removing fork-projects because one fifth of all dockerfiles where forked from kubernetes [3]. Other popular open source tools for docker are dockersh [4], prometheus[5] and weave[6]. All of them are being used in the TOP-100 projects. Most of the tools are only usable as containers in applicable docker environment, therefore they include a Dockerfile as well.

On the other hand, there are relatively underrepresented languages like `PHP` or `Java`. The most used language in GitHub is `Javascript`, which is also very often used by projects with dockerfiles.

[2]`https://cloud.google.com/bigquery/public-data/github`
[3]https://github.com/kubernetes/kubernetes
[4]https://github.com/Yelp/dockersh
[5]https://github.com/prometheus/prometheus/
[6]https://github.com/weaveworks/weave

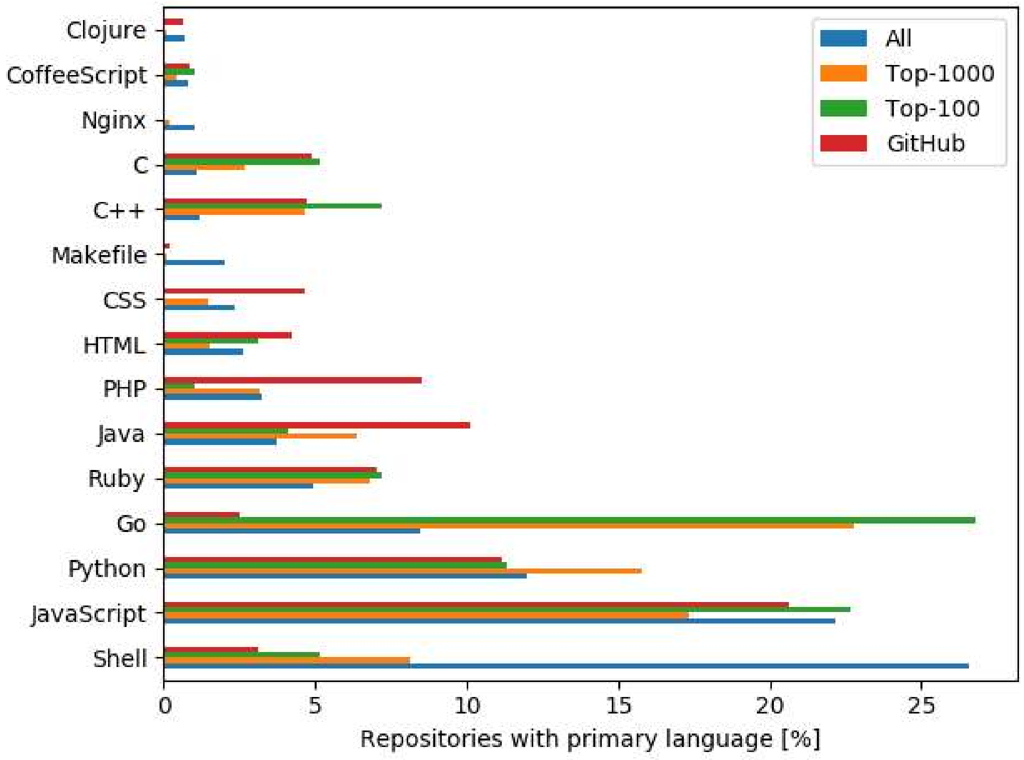


Figure 5.2: Distribution of top 15 languages in our dataset

5.1.3 Base Images

The mandatory base image which is specified as the first instruction (cf. Section 2.5) helps to do a high-level classification and identification of Docker usages. A base image specification is a tuple of the form `(namespace/)name(:version)`. In every case, a name is used to identify an image, and often to indicate the content of the image. For so-called "official" images, such as `ubuntu` or `node`, the name is the sole identifier of an image. Non-official images – or in other words automated builds – further depend on a namespace. It consists of two parts: The first part is usually the name of the organization or user who maintains the image, as it is the case in GitHub, and the second one is the name of the image.Additionally, a base image specification can contain a version string which can be a specific version number (like `1.0.0`) or a more flexible version query (like `latest`), (see Section 2.1 for more information). Overall, the Dockerfiles assessed contained 9′298 unique base image specifications. Figure 5.3 shows the 15 most commonly used base images in our dataset overall and how often they are used in the Top-1000 and Top-100 projects. The figure shows a strong representation of Linux distributions, including `ubuntu`, `debian`, `centos`, `alpine`, and `fedora`. Notably 14 out of the 15 most common base images are official ones – among all 12′001 images, only 148 (1.4%) are official. Furthermore, it is remarkable that of the overall assessed projects, 43.2% of the thousand most used images have a version number attached to the base image, whereas this figure is 36% for the TOP-100 and only 12.3% for the TOP-1000. The result shows a negative association between popularity and whether a version number is attached to the base image. Software developers might be interested in the version they are going to use.

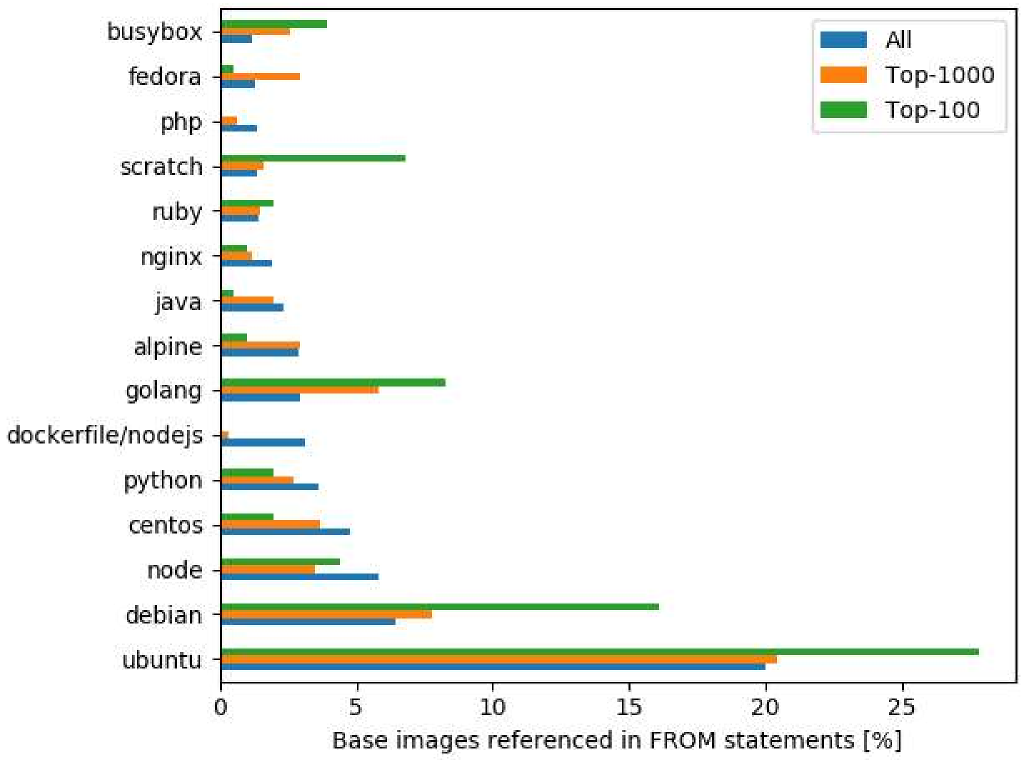


Figure 5.3: Percentage of usage of 15 most commonly used base images

So far, the use of Docker is still unclear. To create more clarity a manual classification of the top 25 images used in the overall data, the Top-1000, and the Top-100 into 5 types will help to understand its usage. This method allows to cover significant parts of the overall data, as the 25 most commonly used images account for 71% of usage across all images. "OS" are base images that contain a blank operating system, without further software being installed. "Language runtime" images additionally contain a runtime needed to execute applications written in a specific programming language. "Application" base images come bundled with an application, a database or a web server. Base images specifications that contain placeholders for parameters to be filled out at runtime are marked as "variable", e.g., `{{namespace}}/{{image_prefix}}base:{{tag}}`. Finally "other" denotes images that do not fit clearly into any of the above categories, for example `scratch`, an empty image, `busybox` or a collection of UNIX utilities. Figure 5.4 shows that the operating systems dominate in all three data sets. Within the OS section, ubuntu is the most frequent one followed by debian and centos.Those three are followed by the language runtime base images section. As mentioned above, node is ranked first place ahead of python and golang. The low number of application base images may be, on first sight, surprising. However, the here analyzed Dockerfiles likely define application images themselves. The presented numbers do not necessarily reflect the actual usage amounts of base images, it is a only a rough subdivision.

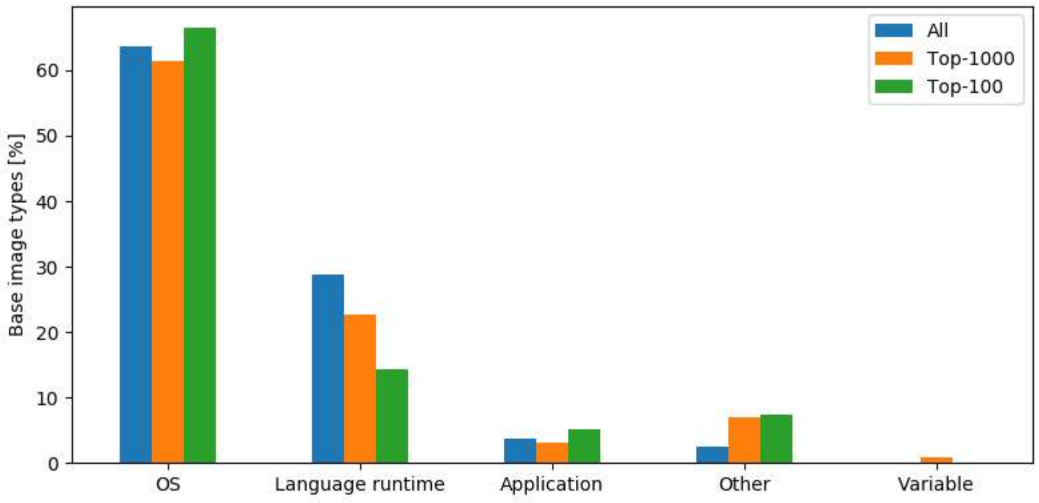


Figure 5.4: Percentage of base image types across top 25 images

5.1.4 Instructions

Each instruction creates a layer within a container, starting with the base image as the lowest layer. Therefore, instructions are a good indication of how projects are built. Table 5.1 lists a percentage breakdown of the instructions used in the assessed Dockerfiles, as well as the fraction of repositories that use instructions at least once, across possibly multiple Dockerfiles associated with the repository. `RUN` is the instruction that is clearly used the most in Dockerfiles, thus it is not surprising that it is ranked first. Notably, `RUN` and `COMMENT` instructions make up 56% of a typical Dockerfile. The remaining instructions (`ADD`, `CMD`, `COPY`, ...) are used in many Dockerfiles, but only represent small fractions of code. Instructions, which support or act as a placeholder, such as `Label`, `USER` and `ARG` are used very infrequently in practice (1% and less than 1% of all Dockerfiles, respectively).

Most instructions are used relatively evenly across the data sets (entire population, Top-1000, Top-100). One exception is the significant lower usage of the `MAINTAINER` instruction in the top views, which is used to set the author of a Dockerfile. The instruction has recently been deprecated and it may be harder to clearly assign a single maintainer in larger projects. Another exception is the `WORKDIR` instruction, that is used to set the directory for `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions to run in. This instruction is more often used in the top repositories. To this regard, top repositories follow prescribed best practices to use `WORKDIR` over `cd` in `RUN` instructions.[7]. A interesting observation is that the instruction `ENV` is ranked as number three, whereas `ARG` took the last place, although there is only one small difference between them,namely the execution time. The `ARG` instruction defines a variable that users can pass at build time to the builder with the docker build command while the `ENV` instruction sets the environment variable

[7]`https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#/workdir`

Table 5.1: Percentages of instructions and repositories using instructions at least once

	Instructions			Repos using instructions (at least once)		
Instruction	All	T1000	T100	All	T1000	T100
RUN	0.40	0.41	0.48	0.87	0.87	0.86
COMMENT	0.16	0.14	0.15	0.54	0.51	0.55
ENV	0.06	0.07	0.09	0.42	0.44	0.47
FROM	0.07	0.08	0.07	0.97	0.96	0.94
ADD	0.06	0.05	0.02	0.46	0.48	0.37
CMD	0.04	0.04	0.03	0.56	0.53	0.47
COPY	0.03	0.04	0.03	0.29	0.32	0.32
EXPOSE	0.04	0.04	0.03	0.45	0.43	0.42
MAINTAINER	0.04	0.04	0.03	0.55	0.42	0.45
WORKDIR	0.03	0.03	0.03	0.45	0.53	0.57
ENTRYPOINT	0.02	0.02	0.01	0.23	0.31	0.27
VOLUME	0.02	0.02	0.01	0.17	0.19	0.16
LABEL	0.01	0.01	0.00	0.01	0.02	0.02
USER	0.01	0.01	0.01	0.10	0.10	0.08
ARG	0.00	0.00	0.00	0.00	0.00	0.00

and persist the value when a container is running from the resulting image. The analysis might indicate that developers prefer to define everything needed inside a container instead of passing values during the build. Probably IaC Code is seen as a kind of deployment-package, that should not need any interaction during the build.

The following sub-section takes a closer look at the `RUN`, `COMMENTS`, `CMD` and `HEALTHCHECK` instructions.

RUN

Given that `RUN` instructions are used so prevalently to implement Dockerfiles, this sub-section tries to identify the kind of commands that are being issued by `RUN` instructions within containers. A categorization of the commands executed in `RUN` will be presented. First, by sorting by usage and manually classifying six categories from the top 100 results:

dependencies	package management or build commands, such as `apt-get` or `pip`
filesystem	UNIX utilities used to interact with the file system, such as `mkdir` or `cd`
permissions	UNIX utilities and commands used for permission management, such as `chmod`
build/execute	build tools such as `make`
environment	UNIX commands that set up the correct environment, such as `set` or `source`
other	remaining commands

The majority of commands (>70%) can be classified as belonging to either *dependencies* (∼45%) or *filesystem* (∼30%). There are not many major differences between the overall population and the top projects on GitHub. Interestingly, Top-100 projects use 13.5% *build or execute* commands, compared to only 5.3% in all projects. A potential explanation is that popular projects themselves are often used as a base for other Dockerfiles and thus need more commands to build their own source. This effect is visible in the permissions categorization, where the overall population uses more permissions than the popular projects. From an architectural perspective it makes sense to use permissions above a fully functional container. Thus, it might be possible that users of popular projects try to add their customizations as additional layer on top of the stack.

Table 5.2: Breakdown of `RUN` instructions in six categories

		Instructions		
Category	**Examples**	All	T1000	T100
Dependencies	apt-get, yum, npm, pip, mvn	0.452	0.447	0.452
Filesystem	mkdir, rm, cd, cp, touch, ln	0.304	0.293	0.294
Permissions	chmod, chown, useradd	0.073	0.052	0.023
Build/Execute	make, python, service, install	0.053	0.083	0.135
Environment	set, export, source, virtualenv	0.006	0.01	0.002
Other		0.113	0.115	0.094

COMMENTS

Table 5.1 clearly shows the usage of the functionless instruction `COMMENT`. This sub-section tries to attain a better understanding of the comments inside a Dockerfile.

Table 5.3: Percentages of Comments before an Instruction

	Instructions		
Instruction	All	T1000	T100
`RUN`	0.48	0.54	0.57
`ENV`	0.09	0.14	0.10
`FROM`	0.07	0.07	0.04
`COPY`	0.05	0.05	0.07
`ADD`	0.11	0.04	0.03
`EXPOSE`	0.06	0.03	0.03
`CMD`	0.03	0.03	0.06
`WORKDIR`	0.02	0.02	0.03
`VOLUME`	0.02	0.01	0.01
`ENTRYPOINT`	0.01	0.01	0.0
`USER`	0.01	0.01	0.0
`LABEL`	0.00	0.00	0.0
`MAINTAINER`	0.00	0.00	0.0
`ONBUILD`	0.00	0.0	0.00
`ARG`	0.00	0.0	0.00

The `RUN` Instruction is placed first in all three data sets. An explanation could be the nature of it, which allows a broad oppertunity of use. According to the data set, a lot of the comments on the RUN instruction where description of the execution itself, which might indicate a need of additional information for other developers, which supports the finding, that dependencies trigger most of ambiguities in Dockerfiles. Notably, the `ADD` instruction is more commented in

Table 5.4: List of most frequent words or all Instructions

Instruction	most frequent words sorted in descending order
RUN	install, dependencies, packages, add, prerequisites, create, update,build, directory, file
ENV	development, only, currently, environment, install, variables, git, setting, version, install
FROM	image, base, build, Dockerfile, pull, ubuntu, base, version, file,start
COPY	copy, to, and, app, install, source, add, files, bundle, dependencies
ADD	install, start, packages, available, work, npm, manually, add, script, files
EXPOSE	port, server, 3000, livereload, default, http, container, web, 80, api
CMD	default, command, run, container, start, system, server, script, application, execution
WORKDIR	working, directory, define, set, install, application, build, dir, change, app
VOLUME	to, add, for, allow, volume, define, logs, backups, databases, config
ENTRYPOINT	command, script, container, default, run, commands, allow, containers, nested, wrap
USER	user, created, rest, package, 'postgres', 'apt-get', root, run, change, switch
LABEL	labels, consumed, build, Red, service, hat, used, info, searching, tenet
MAINTAINER	maintainer, author, file, /, the, imgae, details, name, thx, info
ONBUILD	build, install, and, project, copy, directory, user, dependencies, instructions, change
ARG	user, setup, permissions, superuser, match, host, user, build, arguments, into

overall packages than in the popular ones. A closer examination of the 10 most frequent words for each instruction in the overall dataset show the following:

There are also discrepancies between the instructions, for instance words from the `RUN` instruction are describing more or less the use cases of this instruction, whereas `EXPOSE` uses more specific words such as ports, protocols (http). As expected, frequent words from `ONBUILD` instructions indicate a mix of different instructions. It reflects exactly the nature of it. The conducted dataset also includes instructions, which have been commented out. The information content of those not high. The `RUN` instruction is the most out commented instruction, followed by `ENV`, `ADD`. Commenting out a piece of code rather than removing it from the file could point out that it is important to keep it. The information value keeping it is higher compared to removing it completely. A closer look at some `RUN` instructions, which have been commented out, shows that most of them are commands where something is installed(apt-get install), printed out (echo) , updated or removed.

EXPOSE

`EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. It has to be considered that exposing a port does not make the port accessible to other containers. Actually 33% of all Dockerfiles use this instruction. Table 5.5 shows the 20 most used ports, which account for 70% of usage across all used ports. To get a better understanding of the most used ports, a mapping to the underlying image helped to classify the corresponding container and its purpose. Port 80 is ranked first, followed by port 8080, which is also used for HTTP. The main difference between these two is, that it is necessary to have superuser rights on the server to use the standardized HTTP port 80. It is possible that the specified running Container has limited rights for security reasons. Notably, three database providers are listened, which shows that the best practice-rule "one-process" per Container is respected in this case. Many language-specific webservers are listened, such as pyhton, php, ruby and definitively javascript and shows the dominance of images of the category `Language Runtime`.

Table 5.5: 20 most used ports

rank	port	%	usage
1	80	19.49842	HTTP
2	8080	15.82209	HTTP
3	3000	15.43477	Javascript Server
4	35729	7.64315	Livereload
5	22	6.32303	Secure Shell
6	443	5.29340	HTTPS
7	5000	4.84152	Python Server
8	8000	4.27345	Python Server
9	9000	3.35356	PHP Server
10	3306	2.68866	MySQL
11	5432	2.66929	PostgreSQL
12	8888	2.42722	HyperVM HTTPS
13	6379	1.57511	Redis
14	9200	1.51701	Elasticsearch
15	8081	1.46214	HTTP/Alternativ
16	27017	1.28139	mongoDB
17	9300	1.18779	Elasticsearch
18	53	1.10064	Domain Name System
19	4567	0.83920	Ruby
20	8302	0.76819	Supervisor

CMD and ENTRYPOINT

The CMD Instruction defines the default arguments or an Entrypoint instruction. However, a user can overwrite these commands or pass in and bypass the default parameters from the command line when the container is running, whereas the ENTRYPOINT instruction is the program to run the given command. It is recommended to use ENTRYPOINT when a container should act as an executable and when a command always needs to be executed. If the default commands and/or argument should be alterable, CMD should be preferred. A manual classification of six categories from the top 100 of both instruction resulted in:

Server official server software node, nginx, apache, gunicorn, redis-server

Database also server, but with specialization in databases servers postgres, iinfluxdb, mysql, mongodb

Tools supervisord, grunt, Unix utilities, npm

Language Runtime java, python, dotnet, rails, php

Other shell scripts like: run.sh, start.sh and other self written scripts, Unix commands, openssh ssh daemons

Table 5.6 presents the results, which account for 70% of usage across all used CMD and RUN executables. Both of them have the same rankings, but with some differences. A closer look into the 'Other' Category shows that more than 30% bash scripts are executed in the CMD instruction and more than 40% in the Entrypoint instruction, which is quite high. A possible explanation for this fact is that there might be instructions which always have to be executed and this could be exactly the nature of an executable. Executing a bash script is a way to circumvent the "one process per container"-rule (see 5.3.2) for more informations). Tools like e.g. npm, grunt etc. need arguments to run, and thus it fits exactly in the nature of CMD instruction. The amount of used tools in the Entrypoint is less than half of CMD. Lastly, specific Server applications such as databases are also less used in the Entrypoint session, whereas Server applicatins and Language Runtime based applications seems to be used identically.

Table 5.6: Camparison of CMD and ENTRYPOINT Instruction

Category	CMD	ENTRYPOINT
Other	42%	65%
Tool	29%	12%
Language Runtime	15%	14%
Server	9%	8%
Database	5%	1%

5.2 Maintenance and Evolution

This section analyzes the dataset in terms of maintenance and evolution of Dockerfiles. By definition a revision represents a change to a document's contents or a modification to a part such that the part remains interchangeable within its previous variation [57]. In this thesis every revision is formed by a commit that added, removed or modified at least one line in a Dockerfile. For every Dockerfile, this section uses revisions per year as a metric to define how often it is updated. The first commit of a Dockerfile on Github is also a revision, so every Dockerfile has at least one revision. The transition from a revision to a newer revision is defined by a diff. A Diff consists of changes. This thesis tries to categorize the changes, which belong to a Diff. This section starts with the rate of change, followed by the magnitude of change and finally the nature of change.

5.2.1 Rate of Change

Figure 5.5 visualizes yearly revisions for all studied Dockerfiles. Evidently, Dockerfiles generally are not changed often. The arithmetic average of the number of yearly revisions over all files is 3.13. However, this mean is biased, as the distribution appears to closely follow a power law, with 67.15% of Dockerfiles being revised 0 to 1 times per year, 9.69% 3 times and 5.76% 4 times after the initial commit. Section 5.1.3 concluded that there is some evidence, that users build on top of popular projects, which are public and free to use. Interestingly, Dockerfiles belonging to the Top-100 and Top-1000 projects are in fact updated substantially more often, with an arithmetic mean of 5.72 and 4.35 revisions per year respectively. Taken together, with the results discussed in Section 5.1, we can conclude that more popular projects are not only larger, but also maintained more actively.Thus users, who build on top of popular projects, do that consciously, because they expect more maintenance indirectly from them. From a software development perspective, this is a very good use of the module pattern. A big disadvantage of this behavior is that it could be a problem, if users build to many dependencies on top of the popular project. For the Top-100 and Top-1000 samples, no power law can be observed. Furthermore, the slope of the distribution is flatter than predicted by a power law, i.e., more Dockerfiles are revised more frequently. A reason for that could be the increase of dependencies inside the container.

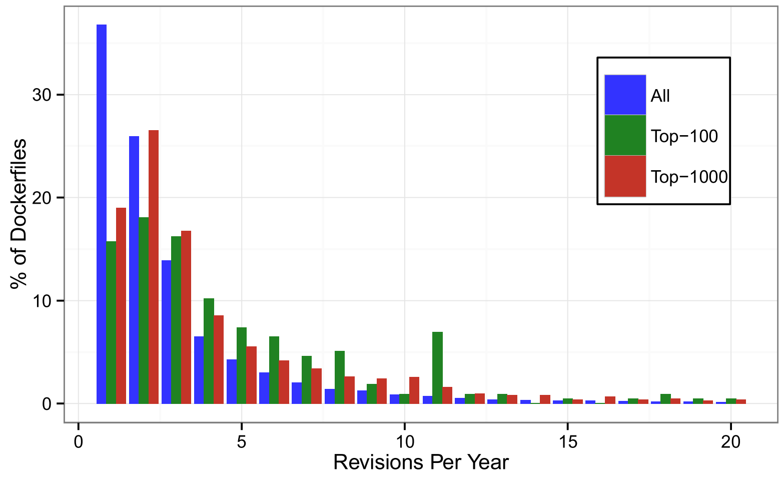


Figure 5.5: Distribution of yearly revisions

5.2.2 Magnitude of Change

This subsection takes a closer look at the size of revisions in terms of addition, modifications and removal of instructions. All initial commits have been filtered out for this analysis, thus only revisions that actually modify an already existing Dockerfile are considered. Evidently, revisions on Dockerfiles are typically small, with an average of only 3.48 lines of code in Dockerfiles being changed. 80.39% of all revisions consist of 5 lines of Docker code changed or less. Figure 5.6 depicts the number of total, added, removed or modified lines of Docker code per revision. Plot 5.6 shows the data of the entire set, because no significant difference for the Top-100 and Top-1000 projects as compared to the entire data set have been observed. A classification of the revisions into additions, modifications and removals shows that most changes to Dockerfiles consist of instruction additions (47.44%), followed by the removal (33.17%) and modification (19.40%) of instructions.

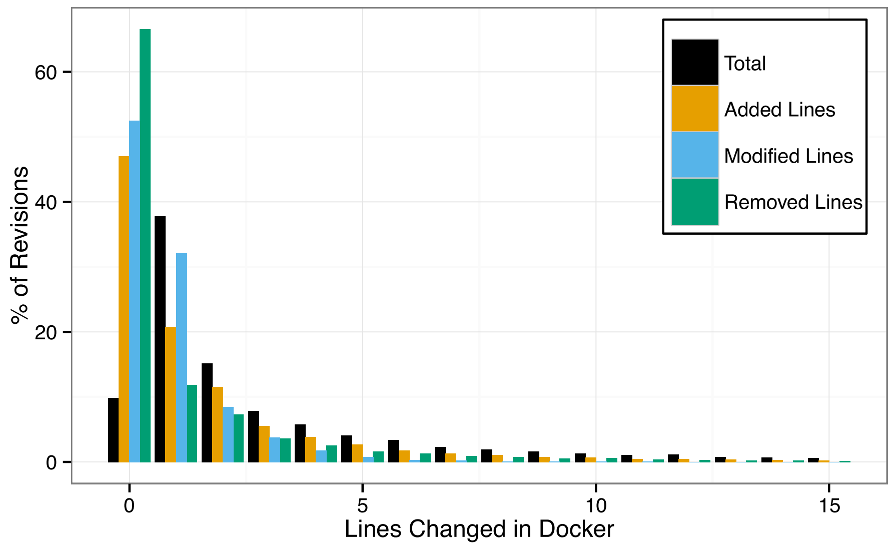


Figure 5.6: Distribution of revision sizes measured in lines changed

Docker started in 2013 and has on average released 3 new versions per month. Table 5.7 shows the Evolution of Version/per month starting Mars 2013 on the left side. Next to it, the number of releases (without RC) and also the three kind of changes on a Dockerfile: del, ins, mod. The last column finally shows the amount of new Dockerfiles per month. First, it is very obvious that Docker has a clear Roadmap of its software. Interestingly, the average of its changes is very linear, thus Dockerfiles have not been much affected from new releases of the main application, since it actually builds the files. There are only a few outliers. The first one occur in November 2013, where many lines were deleted but also many new lines where added. A closer look into to these time ranges shows that a project called 'docker-scientific-python', which has a Dockerfile with more than 50 lines of code for for each python version, added and deleted many dependencies during that time. Its Dockerfiles consists of 20-30 RUN instructions, which is quit high. After these changes have been made public, the projects which used their Dockerfiles, also started updating their files, which increased the del, adds and ins in November 2013. Again, it shows that maintaining dependencies can cause a wave of changes, if the causer is a popular project. Another result, which stands out is: May 2014. A brief look at the change types shows that 25% of all Dockerfiles have changed the underlying image. Most of the images have been ubuntu images. It is very interesting that less than one month before the change, Ubuntu 14.04 has been released. There is a high certainty that the layer above the base images have to be modified, in particular the RUN instructions. In conclusion, new releases of Docker does not affect the maintenance of Dockerfile from a software development perspective.

Table 5.7: Docker Releases mapped to monthly insertions, deletions and modifications on Dockerfiles

		Date		del			ins			mod			
Docker Releases	#	YYYY	MM	sum	counts	**avg**	sum	counts	**avg**	sum	counts	**avg**	**new Dokerfiles**
v0.4.8 - v0.5.1	3	2013	7	84	49	**1.71**	86	49	**1.76**	26	49	**0.53**	**46**
v0.5.2 - v0.6.1	4	2013	8	371	394	**0.94**	750	394	**1.9**	134	394	**0.34**	**154**
v0.6.2 - v0.6.3	2	2013	9	317	291	**1.09**	556	291	**1.91**	105	291	**0.36**	**106**
v0.6.4 - v0.6.5	2	2013	10	375	299	**1.25**	685	299	**2.29**	153	299	**0.51**	**228**
v0.6.6 - v0.6.7, v0.7.0-rc1 - rc7, v0.7.0	4	2013	11	1380	457	**3.02**	1810	457	**3.96**	287	457	**0.63**	**253**
v0.7.1, v0.7.2	2	2013	12	842	495	**1.7**	1395	495	**2.82**	275	495	**0.56**	**294**
v0.7.3 - v0.7.6	4	2014	1	1863	939	**1.98**	2267	939	**2.41**	605	939	**0.64**	**390**
v0.8.0, v0.8.1	2	2014	2	1646	1028	**1.6**	2510	1028	**2.44**	700	1028	**0.68**	**552**
v0.9.0, v0.9.1	2	2014	3	1838	1179	**1.56**	2628	1179	**2.23**	808	1179	**0.69**	**525**
v0.10.0	1	2014	4	2336	1395	**1.67**	3154	1395	**2.26**	1042	1395	**0.75**	**488**
v0.11.0, v0.11.1	2	2014	5	2500	1458	**1.71**	3573	1458	**2.45**	1281	1458	**0.88**	**762**
v0.12.0, v1.0.0, v1.0.1	3	2014	6	5495	3120	**1.76**	6990	3120	**2.24**	1627	3120	**0.52**	**1066**
v1.1.0 - v1.1.2	3	2014	7	3365	2305	**1.46**	4989	2305	**2.16**	1869	2305	**0.81**	**1181**
v1.2.0	1	2014	8	3927	2791	**1.41**	5616	2791	**2.01**	2116	2791	**0.76**	**1764**
	0	2014	9	5725	3220	**1.78**	7297	3220	**2.27**	2312	3220	**0.72**	**1422**
v1.3.0,v1.3.1	2	2014	10	5574	3619	**1.54**	8273	3619	**2.29**	2826	3619	**0.78**	**1704**
v1.3.2	1	2014	11	4921	3322	**1.48**	7805	3322	**2.35**	2534	3322	**0.76**	**1790**
v1.3.3, v1.4.0 -v1.4.1	5	2014	12	4302	3868	**1.11**	7416	3868	**1.92**	2661	3868	**0.69**	**2032**
v1.5.0-rc1 - v1.5.0-rc4	4	2015	1	7668	4274	**1.79**	10125	4274	**2.37**	3257	4274	**0.76**	**2108**
v1.5.0	1	2015	2	8624	5277	**1.63**	9155	5277	**1.73**	4424	5277	**0.84**	**2285**
v1.6.0-rc1 -rc2	1	2015	3	7153	5188	**1.38**	11053	5188	**2.13**	3603	5188	**0.69**	**2659**
v1.6.0-rc3-rc7,v1.6.0	2	2015	4	8628	5291	**1.63**	11861	5291	**2.24**	3863	5291	**0.73**	**2707**
v1.6.1, v1.6.2, v1.7.0-rc1	3	2015	5	9035	5786	**1.56**	13544	5786	**2.34**	4567	5786	**0.79**	**2974**
v1.7.0-rc2 --rc5, v1.7.0	2	2015	6	8055	6222	**1.29**	12305	6222	**1.98**	4607	6222	**0.74**	**2884**
v1.7.1-rc1 --rc3, v1.7.1, v1.8.0-rc1	3	2015	7	8804	6278	**1.4**	13925	6278	**2.22**	5011	6278	**0.8**	**3400**
v1.8.0-rc2-rc3,v1.8.0, v1.8.1	3	2015	8	7726	6009	**1.29**	12658	6009	**2.11**	4115	6009	**0.68**	**3331**
v1.8.2-rc1, v1.8.2	2	2015	9	8684	6692	**1.3**	13192	6692	**1.97**	5211	6692	**0.78**	**3266**
v1.8.3 ,v1.9.0-rc1-rc4	2	2015	10	10459	6838	**1.53**	14306	6838	**2.09**	5296	6838	**0.77**	**4030**
v1.9.0-rc5, v1.9.0, v1.9.1-rc1, v1.9.1	4	2015	11	12098	7849	**1.54**	16364	7849	**2.08**	6532	7849	**0.83**	**3713**
	0	2015	12	9914	7376	**1.34**	15404	7376	**2.09**	5595	7376	**0.76**	**3961**
v1.10.0-rc1-rc2	1	2016	1	10215	7589	**1.35**	14660	7589	**1.93**	5922	7589	**0.78**	**3539**
v1.10.0-rc3-rc4, v1.10.0 , v1.10.1-rc1, v1.10.1, v1.10.2-rc1, v1.10.2	6	2016	2	13068	8667	**1.51**	17753	8667	**2.05**	6782	8667	**0.78**	**3725**
v1.10.3-rc1-rc2, v1.10.3, v1.11.0-rc1-rc3,	3	2016	3	11897	8550	**1.39**	16881	8550	**1.97**	6718	8550	**0.79**	**3698**
v1.11.0-rc3-rc5,v1.11.0, v1.11.1-rc1, v1.11.1	4	2016	4	11427	8467	**1.35**	15673	8467	**1.85**	6302	8467	**0.74**	**3403**
v1.11.2-rc1	1	2016	5	11931	8806	**1.35**	16817	8806	**1.91**	6808	8806	**0.77**	**3222**
v1.11.2, v1.12.0-rc1 -rc2	2	2016	6	9163	7598	**1.21**	13380	7598	**1.76**	6217	7598	**0.82**	**3379**
v1.12.0-rc3 - rc5, v1.12.0, docs-v1.12.0	3	2016	7	9923	7183	**1.38**	13776	7183	**1.92**	5611	7183	**0.78**	**2807**
docs-v1.12.0, v1.12.1-rc1-rc2, v1.12.1	3	2016	8	9159	7053	**1.3**	12746	7053	**1.81**	5287	7053	**0.75**	**5014**
v1.12.2-rc1-rc3, v1.12.2, v1.12.3-rc1, v1.12.3	4	2016	9	8973	7325	**1.22**	13244	7325	**1.81**	5578	7325	**0.76**	**3833**
v1.13.0-rc1-rc3	1	2016	10	11510	8212	**1.4**	16068	8212	**1.96**	6698	8212	**0.82**	**3732**
v1.13.0-rc3, v1.12.4-rc1-rc2	2	2016	11	11005	8341	**1.32**	16077	8341	**1.93**	6327	8341	**0.76**	**3697**
v1.13.0-rc3, v1.12.4-rc1, v1.12.4, v1.12.5-rc1, v1.12.5, v1.13.0-rc4	6	2016	12	11231	8648	**1.3**	16842	8648	**1.95**	6687	8648	**0.77**	**3551**

5.2.3 Nature of Change

Table 5.8 goes deeper into the nature of changes to Dockerfiles. It lists all currently supported instruction types as discussed in Chapter 2.2, and specifies how many changes refer to an instruction of the respective type. The table does not include results for `LABEL`, `USER`, and `ARG` because their changes accounted for less than 0.2% in all categories. 55% of all changes relate to `RUN` instructions.
RUN instructions make up 40% of all instructions (cf. Table 5.1), this type of instruction appears to be changed more frequently than others. Table 5.8 also lists the subcategories of it (e.g., dependency management, file system) and reveals a similar distribution for Dockerfiles in general (cf.Table 5.2). Interestingly, `RUN` instructions are often added or removed (59% and 61% respectively), but less frequent modified (29%).
FROM instructions are frequently modified (29%), but almost never added or removed. The reason behind this is that, the base image is mandatory and the foundation of each Container. Everything is built up on it. Section 5.2.2 presented a rare case, where a base image was updated to the latest ubuntu image, which leads, as a consequence to many changes on the layer above it. Consequently, Dockerfiles typically contain exactly one `FROM` instruction and if there are more than one, then the last one is selected. An update to this instruction may indicate a switch to a different base image, or a version change.

Table 5.8 presents the same data for the Top-1000 and Top-100 projects. Largely, no significant differences can be observed for these projects. However, the Top-100 projects appear to be modifying the `FROM` instruction less frequently than other projects (23% for the Top-100 versus 29% in the entire population). This might which may be explained by projects with a large user base being more conscious of significant changes such as updating the base image.

Table 5.8: Relative changes of all Docker instruction types.

	All				**Top-1000**				**Top-100**			
	All	**Add**	**Mod**	**Rem**	**All**	**Add**	**Mod**	**Rem**	**All**	**Add**	**Mod**	**Rem**
`RUN`	0.53	0.59	0.29	0.61	0.55	0.64	0.25	0.64	0.58	0.66	0.31	0.62
Dependencies	0.48	0.5	0.34	0.49	0.49	0.51	0.35	0.5	0.51	0.52	0.44	0.51
Filesystem	0.26	0.24	0.42	0.25	0.24	0.23	0.41	0.22	0.24	0.24	0.37	0.2
Permissions	0.06	0.06	0.07	0.06	0.05	0.06	0.04	0.05	0.03	0.03	0.0	0.04
Build/Execute	0.04	0.04	0.03	0.05	0.06	0.05	0.05	0.07	0.1	0.09	0.06	0.14
Environment	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.0	0.01	0.01	0.01	0.0
`COMMENT`	0.12	0.12	0.11	0.12	0.09	0.09	0.10	0.09	0.10	0.10	0.10	0.09
`ENV`	0.07	0.06	0.13	0.05	0.07	0.05	0.16	0.05	0.10	0.06	0.16	0.12
`FROM`	0.06	0.00	0.29	0.00	0.07	0.00	0.31	0.00	0.00	0.04	0.26	0.00
`ADD`	0.05	0.05	0.05	0.06	0.06	0.05	0.06	0.07	0.02	0.02	0.01	0.03
`CMD`	0.05	0.04	0.06	0.05	0.04	0.04	0.04	0.04	0.03	0.03	0.04	0.02
`COPY`	0.03	0.04	0.03	0.03	0.03	0.04	0.03	0.03	0.04	0.05	0.03	0.04
`EXPOSE`	0.02	0.02	0.01	0.02	0.01	0.02	0.01	0.01	0.02	0.02	0.01	0.01
`MAINTAINER`	0.01	0.01	0.02	0.00	0.01	0.00	0.04	0.00	0.01	0.00	0.02	0.01
`WORKDIR`	0.01	0.02	0.00	0.02	0.01	0.01	0.00	0.01	0.01	0.02	0.00	0.02
`ENTRYPOINT`	0.02	0.01	0.02	0.02	0.01	0.01	0.02	0.01	0.03	0.01	0.04	0.03
`VOLUME`	0.01	0.01	0.01	0.01	0.01	0.01	0.00	0.01	0.01	0.02	0.00	0.01

Distribution of Images over time

Section 5.1.3 presents a distribution of images and also categorized them into five types. The following subsection presents a distribution over the time for the two big categories: OS and

Language Runtime. Figure 5.7 shows the dominance and the fast growth of ubuntu, but also the fact that there is space for new images. For instance, alpine started growing in October 2010 and had the strongest growth after ubuntu. If the trend continues, it will catch up with centos and maybe debian as well. Alpine[8] contains a light-weight operating system. It actually has a size of only *4MB* [6], in comparison to ubuntu *180MB* [6]. It is clear that users prefer smaller base images, since they can be downloaded, built and started faster. Figure 5.8 shows the distribution of Language Runtime Images. The green line shows the automated build of Node, whereas the red one shows the official build. This reinforces the fact that, user use more likely an official image, and then build their layers on top of it, instead of creating everything from the scratch. At the beginning of their containerization in 2015, all (with exception of the green one) had a fast growing time. Java lost speed, whereas the rest is still growing.

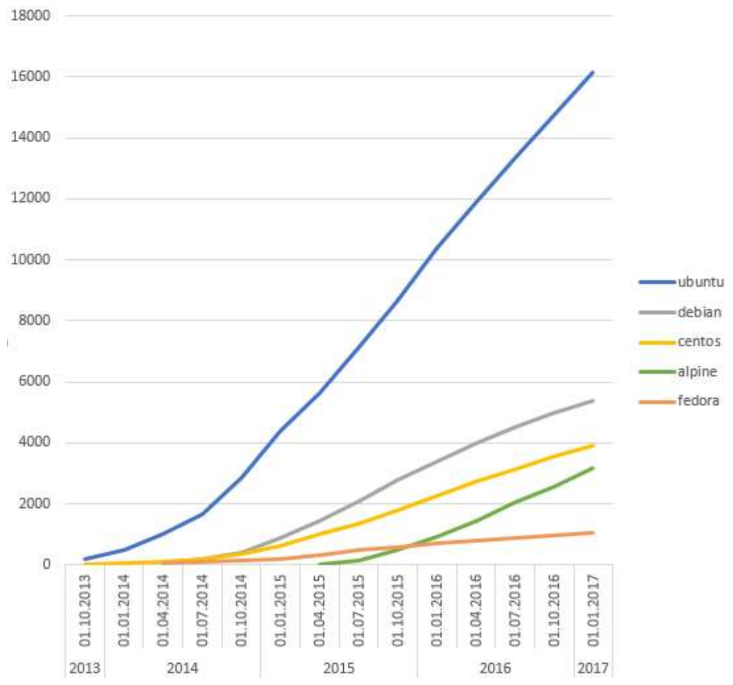


Figure 5.7: Distribution of OS images over time

[8] `https://alpinelinux.org/`

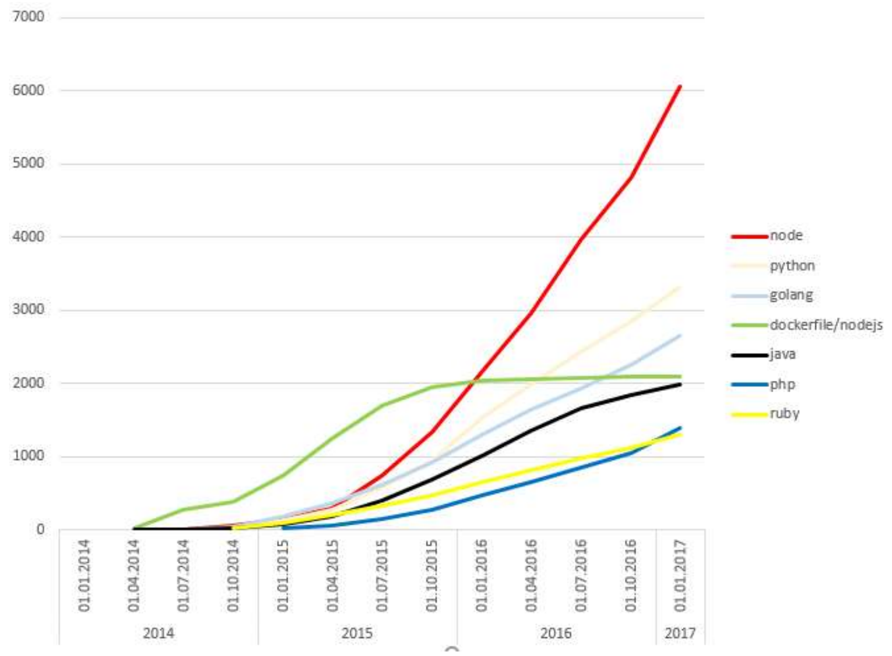


Figure 5.8: Distribution of Language Runtime images over time

Change of Dependencies

Dependencies are changed a lot, corresponding Table 5.8. A closer look at the change types shows that in most of the cases a whole RUN instruction is added, followed by adding parameters to an existing RUN instruction. Less than 10% of the parameters are maintained.

Table 5.9: Change types for RUN Instruction

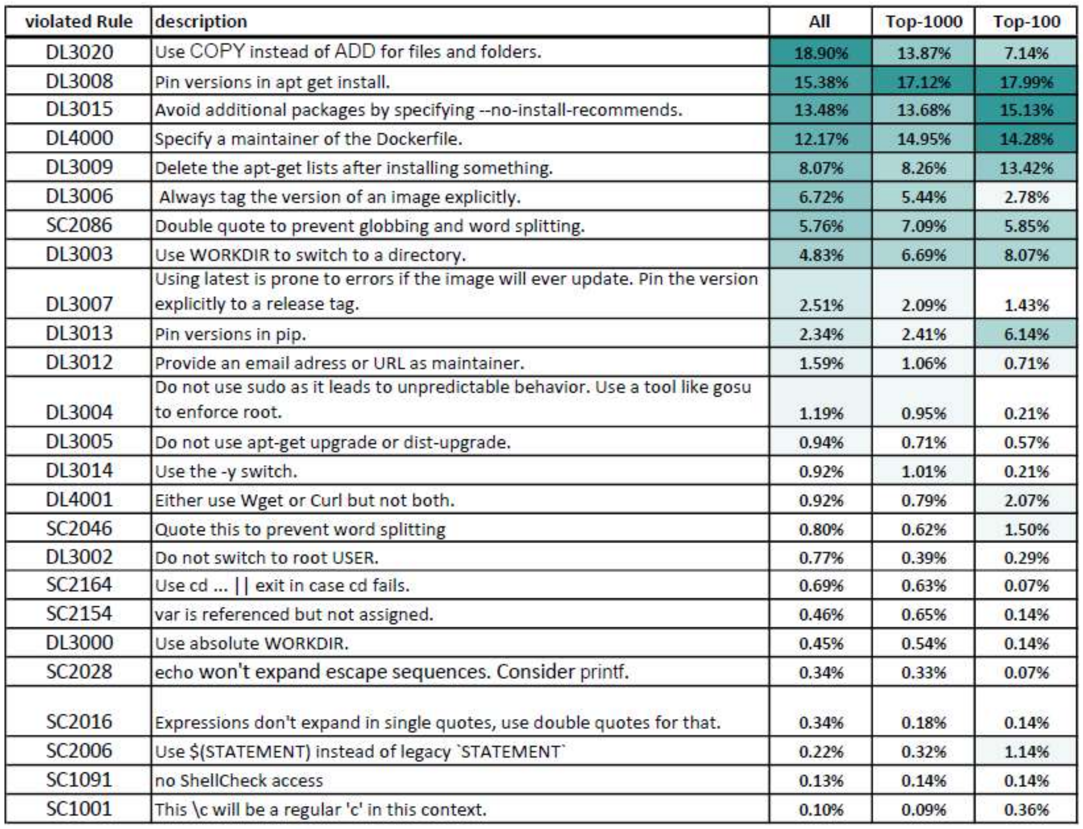
Change Type	%	description
AddType_RUN	30.4	new RUN Instruction
AddType_PARAMTER	22.6	new parameter in existing RUN Instruction
DelType_RUN	22.6	removal of whole RUN instruction
DelType_PARAMTER	15.1	removal of parameter in RUN Instruction
UpdateType_PARAMTER	8.6	update of parameter
UpdateType_EXECUTABLE_PARAMETER	0.5	update of executable and its parameters
UpdateType_EXECUTABLE	0.2	update of executable

5.3 Quality of Dockerfiles

This section focuses on the quality aspects of Dockefiles and is based on a set of best practices evolved [12] from Docker itself for describing images in Docker's declarative language. Best practices are used to maintain quality as an alternative to mandatory legislated standards [5].Thus, best practices are helpful to avoid common mistakes (e.g., using `ADD` instead of `COPY`) and they also help to increase the quality of static code by giving hints to the formation of code. Best practices also exist for various programming languages or other IaC languages (e.g., Puppet, Chef).

In order to identify how Docker repositories on GitHub comply to those best practices this thesis relies on a Dockerfile linter [37], which is based on the aforementioned best practices and contributions by the GitHub community. This linter, which runs also in a Docker container, parses a given Dockerfile and checks adherence to a set of rules representing the best practices on top of the resulting AST. Every Dockerfile in our data set is executed by the linter and the reported rule violations are added to the database.

5.3.1 Most Violated Rules



violated Rule	description	All	Top-1000	Top-100
DL3020	Use COPY instead of ADD for files and folders.	18.90%	13.87%	7.14%
DL3008	Pin versions in apt get install.	15.38%	17.12%	17.99%
DL3015	Avoid additional packages by specifying --no-install-recommends.	13.48%	13.68%	15.13%
DL4000	Specify a maintainer of the Dockerfile.	12.17%	14.95%	14.28%
DL3009	Delete the apt-get lists after installing something.	8.07%	8.26%	13.42%
DL3006	Always tag the version of an image explicitly.	6.72%	5.44%	2.78%
SC2086	Double quote to prevent globbing and word splitting.	5.76%	7.09%	5.85%
DL3003	Use WORKDIR to switch to a directory.	4.83%	6.69%	8.07%
DL3007	Using latest is prone to errors if the image will ever update. Pin the version explicitly to a release tag.	2.51%	2.09%	1.43%
DL3013	Pin versions in pip.	2.34%	2.41%	6.14%
DL3012	Provide an email adress or URL as maintainer.	1.59%	1.06%	0.71%
DL3004	Do not use sudo as it leads to unpredictable behavior. Use a tool like gosu to enforce root.	1.19%	0.95%	0.21%
DL3005	Do not use apt-get upgrade or dist-upgrade.	0.94%	0.71%	0.57%
DL3014	Use the -y switch.	0.92%	1.01%	0.21%
DL4001	Either use Wget or Curl but not both.	0.92%	0.79%	2.07%
SC2046	Quote this to prevent word splitting	0.80%	0.62%	1.50%
DL3002	Do not switch to root USER.	0.77%	0.39%	0.29%
SC2164	Use cd ... || exit in case cd fails.	0.69%	0.63%	0.07%
SC2154	var is referenced but not assigned.	0.46%	0.65%	0.14%
DL3000	Use absolute WORKDIR.	0.45%	0.54%	0.14%
SC2028	echo won't expand escape sequences. Consider printf.	0.34%	0.33%	0.07%
SC2016	Expressions don't expand in single quotes, use double quotes for that.	0.34%	0.18%	0.14%
SC2006	Use $(STATEMENT) instead of legacy \`STATEMENT\`	0.22%	0.32%	1.14%
SC1091	no ShellCheck access	0.13%	0.14%	0.14%
SC1001	This \c will be a regular 'c' in this context.	0.10%	0.09%	0.36%

Table 5.10: 25 most violated rules reported by the linter

Table 5.10 shows the 25 most violated rules reported by the linter. After providing a short overview about the quantity of rule violations, the following subsections focus more on specific rules and the differences, which have been identified among the considered segments of the population.

Overview

Dockerfiles violate 3.2 rules on average across the entire population, while Dockerfiles of the Top-100 most prominent projects violate 3.4 rules and Dockerfiles of the Top-1000 projects 3.7 rules on average. One potential reason is that more popular projects tend to have more sophisticated Dockerfiles, thus the chances for violating rules is higher than for simple Dockerfiles with only a few instructions. Focusing on the Top-100 and their Dockerfiles, 18 projects out of the Top-100 do not violate a single rule. This is similar to the Top-1000, in which 201 projects (i.e., 20%) conform to all of the rules checked by the linter. Across the entire population, this is the case for 14% of the repositories. There is a clear correlation between the amount of RUN instructions and the amount of violated Rules. Dockerfile with less than 4 RUN instructions concern only 21% of all violated rules, Dockerfiles with less than 7 RUN instructions concern half of all violated rules and files with 16 or less instructions concern almost 78%. Therefore, an increase in number of used dependencies expenses in quality.

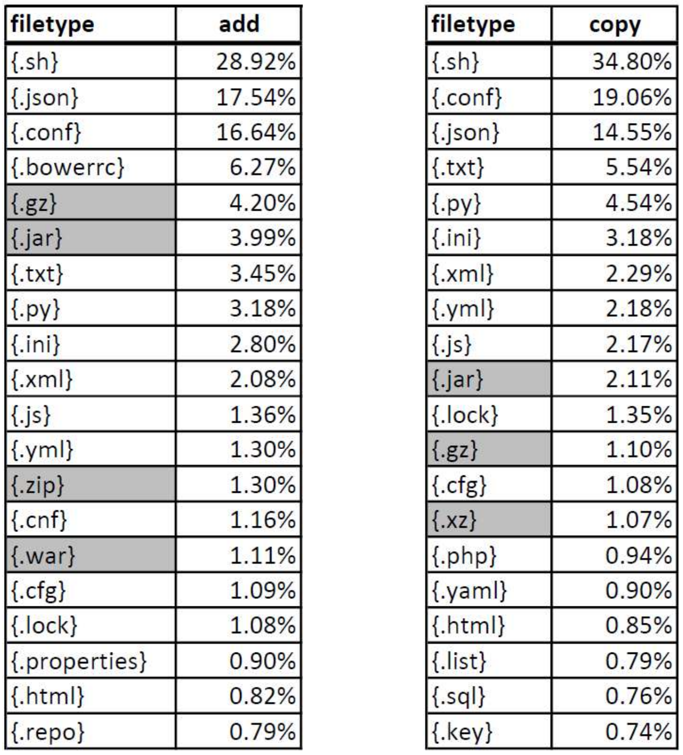
Version Pinning

The linter has rules for four types of version pinning violations: image version pinning (DL3006), apt-get version pinning (DL3008), pip version pinning (DL3013), and usage of ":latest" (DL3007). All of them report the absence of a concrete version, either for the base image (i.e., `FROM`), or for a concrete package to be installed (i.e., `RUN`). If no concrete version is declared, then a scenario can enter where a version is used, which is not compatible with the other components of the container, especially for the base image, which acts as a foundation of the Container. Therefore, best practices suggest to specify concrete versions, because version pinning forces the build to retrieve a particular version regardless of what is in the cache [12]. 6.72% of all Dockerfiles across the entire population violate the rule of image version pinning. Interestingly, when looking at the Top-100 repositories, just 2.78% of the considered Dockerfiles violated this rule. Therefore, more popular repositories might be more aware of bad practices associated with build failures.

Regarding the installation of specific packages (i.e., pip and apt-get version pinning), the most popular repositories perform worse than the entire population. There are two possible explanations for this. Firstly, this thesis makes an assumption that more popular projects have more sophisticated Dockerfiles, hence more dependencies and therefore more `RUN` instructions to violate these rules. This assumption is supported by the finding that the Top-100 projects on average have more `RUN` instructions. Secondly, the purpose of popular projects is more general. Dockerfiles, which use popular images as a base image use more customized packages, where specific versions of packages are installed for an individual software project.

Copy/Add.

One of the most prominent bad practices is to use the instruction `ADD` instead of `COPY` (DL3020). Basically, both instructions provide similar functionality to add resources to an image. However, the `ADD` instruction supports additional functionality, for instance it allows downloading resources from a URL and automatically unpacks compressed local files (e.g., tar, zip, etc). This additional "magic" is considered as dangerous and can lead to accidental failures as long as developers are not aware of it, e.g., when the developer wants to add a zipped folder to the image and `ADD` automatically unpacks it. Across all Dockerfiles, 18.9% violated the practice to prefer `COPY`. Again, this is not the case for the Top-100 repositories with only 7.14%. This assumption is supported by an inspection of the dataset. Table 5.9 presents the most used filetypes in a ADD and COPY instructions. These 20 filetypes cover almost 81% of all filetypes and ADD Instructions and 83% of all filetypes in the COPY Instruction. According to the nature of the ADD Instruction, there should be only compressed filetypes on the list. Despite this fact, in reality the four filetypes which appear in the list in gray sum together to 10.6%, which proves the wrong use of this Instruction. The COPY instruction has only less than 5% of compressed filetypes.



filetype	add
{.sh}	28.92%
{.json}	17.54%
{.conf}	16.64%
{.bowerrc}	6.27%
{.gz}	4.20%
{.jar}	3.99%
{.txt}	3.45%
{.py}	3.18%
{.ini}	2.80%
{.xml}	2.08%
{.js}	1.36%
{.yml}	1.30%
{.zip}	1.30%
{.cnf}	1.16%
{.war}	1.11%
{.cfg}	1.09%
{.lock}	1.08%
{.properties}	0.90%
{.html}	0.82%
{.repo}	0.79%

filetype	copy
{.sh}	34.80%
{.conf}	19.06%
{.json}	14.55%
{.txt}	5.54%
{.py}	4.54%
{.ini}	3.18%
{.xml}	2.29%
{.yml}	2.18%
{.js}	2.17%
{.jar}	2.11%
{.lock}	1.35%
{.gz}	1.10%
{.cfg}	1.08%
{.xz}	1.07%
{.php}	0.94%
{.yaml}	0.90%
{.html}	0.85%
{.list}	0.79%
{.sql}	0.76%
{.key}	0.74%

Figure 5.9: Top 20 used filetypes in ADD and COPY instructions

WORKDIR

By using absolute paths, the problem of running into problems when a previous WORKDIR instruction changes (DL3000). This rule is very less violated and can be also verified by the dataset, which shows that more than 95% of all Instructions observe this rule. Another rule concerning the WORKDIR instruction is DL3003. The rational behind it is that changing a directory with cd should be avoided, because most of commands can work with absolute paths. If it is really necessary to change the current working directory, then WORKDIR should be used. According to the conducted dataset 4.11% of all Run Instructions use 'cd' as command. Interestingly, 8% of the TOP-100 projects violates this rule, which is quite high.

Missing Maintainer Information

15.9% of all Dockerfiles do not use the `MAINTAINER` instruction specifying both name and email address of the developer responsible for the image. Interestingly, for popular projects, this best practice is violated more often with 21.7% for the Top-100 and 19.8% for the Top-1000 repositories. An assumption is that especially for more popular projects, more developers contribute, thus maintenance is not assigned to an individual developer and consequently, the `MAINTAINER` instruction is not used. This Instruction is deprecated since January 2017.

5.3.2 Multi-Process Management

One big advantage of applying the one-process-per-Container Rule is the separation of concerns, which leads to higher uptime, for instance a database should not have to go down every time an image has to be rebuild and it simplifies maintenance of the container due the fact that more processes need more dependencies, which leads to higher maintenance outlay. By definition, Docker container run a single process following the UNIX philosophy "do one thing and do it well".
However, for decades developers were used to run multiple processes in parallel (e.g., application's main process, worker processes for collecting and persisting logs, etc). Thus,the transition from monoliths software architecture to microservices takes time. The Docker design practice is to isolate every single logical component in a separate container: If new functionality is needed, a new container is created for exactly this purpose.
Nevertheless, there exist cases where it is practical to run multiple processes inside a container (e.g., the actual application and a `Apache` daemon). There are two possibilities to make multiple processes run within a container: Either by writing a shell, script which starts all processes by running the script with the CMD or ENTRYPOINT instruction or by using third party tools, which addresses this issue (e.g., *supervisord*, *runit*, *monit*). Doing this manually is known to be error prone. Section 5.3.1 shows that the most copied filetype with an occurrence of almost 35% is shell. It is obvious that shell scripts are a way to bypass the limitation of executing only one executable per Container. 28% of all Entrypoint Instructions execute a shell script or in other relations, 5631 Dockerfiles. It is preferred to use it in the Entrypoint Instruction rather than in the CMD Instruction, because of the nature of it. In Comparison to the third party tools,*Supervisord* is by far the most common tool for multi-process management. It is also recommended from Docker [9]. 1686 repositories called *supervisord* in the `CMD` instruction, and 177 in the `ENTRYPOINT` instruction. *God* is ranked on second rank and used by 217 repositories in the `CMD` instruction, and 208 in the `ENTRYPOINT` instruction.*s6* (i.e., *s6-svscan*) is on the third place and used by 38 repositories in the `CMD` instruction, and 18 in the `ENTRYPOINT` instruction. Other tools including *monit*, *runit*, *system.d* and *upstart* are used by 10 and less repositories.

An analysis of the supplied parameters to `ENTRYPOINT` and `CMD` instructions, shows the actual need of multiple-processes Containers. In total, 2381 projects (i.e., 3.4% of all Docker projects) make use of such tools. One usage was identified within the Top-100 and 27 projects within the Top-1000. This is less than expected due to the prominent coverage of these approaches in on-

[9]`https://docs.docker.com/engine/admin/using_supervisord/`

line resources (e.g., developer blogs).As this section only analyzed the execution of multi-process tools in `ENTRYPOINT` and `CMD` instructions, the results might be lower than the actual population.

Chapter 6

Threats to Validity

This section presents the main threats to the validity of this thesis.

Construct Validity. An essential threat to construct validity is that the relational model might not be an adequate representation of the different data sources we intended to study. This involves capturing the version history of Dockerfiles broken down to a statement-level including the addition, removal, and modification of concrete instructions, and the representation of the violated rules reported by the Docker linter. Moreover, the used parser has to correctly interpret Dockerfiles and write the extracted data into the chosen relational model. The tooling is based on the assumption that Dockerfiles follow the standard naming convention `Dockerfile` without supplied file type. his threat has been mitigated by manually inspecting and validating a small fraction of parsed projects during construction of the parser as well as in the analysis phase.

Internal Validity. Threats to internal validity include potentially missed confounding factors during result interpretation. For example, this thesis assumes that missing image version pinning might be an explanation for a higher build failure rate in our build experiment. However, there could be other factors such as failed test cases when building the actual application that lead to the image build failure. Another threat to internal validity is our segmentation of the population of Docker repositories. This thesis mitigated the effect of a selection bias towards the 100 and 1000 most popular Docker repositories by including the entire Docker population in all our analyses.

External Validity. This thesis consideres the entire population of Docker repositories on GitHub to extend the generalizability of this study. Excluding forked repositories might limit the generalizability to a certain degree. However, forked repositories without any changes compared to their source repository could have led to result misinterpretation. Only Docker repositories on GitHub have been considered. Consequently, it is not assured that the outcomes generalize to projects hosted on other services, such as Bitbucket. Moreover, this thesis analyzed open source software. Docker usage (e.g., languages, tools) and the compliance to best practices regarding the quality of Dockerfiles might be different in closed source environments. The analyses are only based on Docker and do not include other container technologies such as Linux containers (LXC).

The exactly same Threats to Validity have been made in the paper "An Empirical Analysis of the Docker Container Ecosystem on GitHub " by Cito et al. [33] which is based on the used data set .

Chapter 7

Conclusion

This thesis presents an approach of mining repositories with the latest technologies as well as public data providers such as Google BigQuery. It provides a tool chain, which transforms an given Dockerfile to a structural object and outputs it as a json File. Furthermore it examines the evolution parser, which takes an software project as input and processes through the commit history and finally outputs all information regarding a Dockerfile to a database. Basically, these tools make it possible to conduct the first large-scale empirical study to analyze the ecosystem, quality aspects and evolution behavior of Docker containers on Github. This study is based on 97571 Dockerfiles from 48102 projects, which is the entire population of non-forked projects as of January 2017.

In the following, the findings of three main topics are revisited:

Ecosystem Overview. The results show that most of the Dockerfiles use an OS as its base image. However, not all OS images are created equal. More established systems, such as Debian and Ubuntu come bundled with their entire operating filesystem and can be around *180MB* [6]. Other base images contain light-weight operating systems such as Alpine[1], that are as small as *4MB* [6]. Most projects use the larger OS as base images. An insight into the specific instructions showed that the most commonly used instruction is the generic `RUN`. When breaking down the `RUN` statement, 45% are used to define dependencies.

Evolution Behaviour. This thesis observes that Dockerfiles are not changed often, with a mean of 3.11 to 5.81 revisions per year. Dependencies are over-proportionally represented in the change behavior of Dockerfiles. The base image, which is the foundation of a Container does not change a lot. A distribution of images over time showed that there is space for new images which are characterized by their lightness. New Releases of Docker do not cause changes to Dockerfiles.

Quality Assessment. Given the finding that multiprocessing tools, such as *supervisord* are only used by a minority of the Docker repositories (4%), there is no need for multi-process support according to the results of this thesis. The Docker linter, provides valuable feedback concerning the quality of Dockerfiles. Most of the violated rules can be verified with the conducted dataset. The findings show that 28.6% of quality issues (as indicated by the Docker Linter) arise from missing version pinning. Dockerfiles with less than 4 RUN Instructions concern only 21% o all violated rules. Therefore, an increaase in number of used dependencies expenses in quality.

[1]`https://alpinelinux.org/`

Future work. The provided data set can be used for more detailed analysis. For example an inspection of the changed files within the Dockerfile revision could seek out dependencies to IaC code, which might need to be maintained. An analysis in this field would help software developers to create tools, which warn or remind developers to maintain the dependencies.

Another future work which make use of the provided dataset is a tool which is connected to a database that maps all compatible dependencies to the corresponding base image. This tool could be integrated, for instance, in a CI tool such as Jenkins to check the dependencies and give valuable information about the state of the container. With the aid of this tool, a software developer would get immediately feedback about the build without fetching the Dockerfile through the Continuous Delivery and Integration Pipeline. This tool could be also implemented as a plugin to an existing CI tool such as Jenkins. This rapid feedback would help to increase the quality of Dockerfiles with little effort.

Bibliography

[1] Adam Larson. The Life and Times of a Docker Container. `https://blog.newrelic.com/2016/06/16/docker-container-infographic/`, accessed 2017-02-26, 2016.

[2] M. Alves, A. M.and Pessoa, Salviano, and P. . S. Y. . . A. . B. p. . . C. F., TITLE = Towards a Systemic Maturity Model for Public Software Ecosystems. In Software Process Improvement and Capability Determination.

[3] K. Blincoe, F. Harrison, and D. Damian. Ecosystems in github and a method for ecosystem identification using reference coupling. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 202–207, Piscataway, NJ, USA, 2015. IEEE Press.

[4] C. Boettiger. An introduction to docker for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, Jan. 2015.

[5] C. E. Bogan and M. J. English. *Benchmarking for best practices : winning through innovative adaptation*. New York : McGraw-Hill, 1994. Includes bibliographical references (p. 299-304) and indexes.

[6] Brian Christner. Docker Image Base OS Size Comparison. `https://www.brianchristner.io/docker-image-base-os-size-comparison/`, accessed 2017-02-07.

[7] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616, June 2009.

[8] K. K. Chaturvedi, V. B. Sing, and P. Singh. Tools in mining software repositories. In *2013 13th International Conference on Computational Science and Its Applications*, pages 89–98, June 2013.

[9] J. Cito, V. Ferme, and H. C. Gall. *Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research*, pages 609–612. Springer International Publishing, Cham, 2016.

[10] M. Claes, T. Mens, R. D. Cosmo, and J. Vouillon. A historical analysis of debian package incompatibilities. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 212–223, May 2015.

[11] A. Dan, R. Johnson, and A. Arsanjani. Information as a service: Modeling and realization. In *Systems Development in SOA Environments, 2007. SDSOA '07: ICSE Workshops 2007. International Workshop on*, pages 2–2, May 2007.

[12] Docker. Best practices for writing Dockerfiles.
`https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/`, accessed 2017-02-07.

[13] Docker Documentation. Configure automated builds on Docker Hub.
`https://docs.docker.com/docker-hub/builds/`, accessed 2017-02-26, 2017.

[14] D. Documentation. Dockerfile reference.
`https://docs.docker.com/engine/reference/builder/`, accessed 2017-03-04, 2014.

[15] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 610–614, Washington, DC, USA, 2014. IEEE Computer Society.

[16] J. C. Dueñas, H. A. P. G., F. Cuadrado, M. Santillán, and J. L. Ruiz. Apache and eclipse: Comparing open source project incubators. *IEEE Software*, 24(6):90–98, Nov 2007.

[17] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[18] N. Economides and E. Katsamakas. Linux vs. Windows: A comparison of application and platform innovation incentives for open source and proprietary software platforms. Working Papers 05-07, NET Institute, Oct. 2005.

[19] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 00:171–172, 2015.

[20] B. Fitzgerald and K.-J. Stol. Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages 1–9, New York, NY, USA, 2014. ACM.

[21] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik and Andy Zaidman. Lean ghtorrent: Github data on demand.
`http://www.wired.com/2015/06/problem-putting-worlds-code-github/`, accessed 2017-02-26, 2014.

[22] D. M. German, B. Adams, and A. E. Hassan. The evolution of the r software ecosystem. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 243–252, Washington, DC, USA, 2013. IEEE Computer Society.

[23] Github. Github.
`https://github.com/about`, accessed 2017-02-26, 2017.

[24] Github Price plans. Github.
`https://github.com/pricing`, accessed 2017-02-26, 2017.

[25] Google. Search trend for the keyword "docker". `https://trends.google.ch/trends/explore?q=DOCKER`, 2017.

[26] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. Lean ghtorrent: Github data on demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 384–387, New York, NY, USA, 2014. ACM.

[27] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[28] J. Howison, M. Conklin, and K. Crowston. FLOSSMole: a collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, July 2006.

[29] D. Hub. Technical brief: Docker hub. `https://goto.docker.com/rs/929-FJL-178/images/docker-hub_0.pdf`, accessed 2017-03-04, 2016.

[30] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam. *Testing Idempotence for Infrastructure as Code*, pages 368–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[31] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 187–190, May 2009.

[32] Y. Jiang and B. Adams. Co-evolution of infrastructure and source code: An empirical study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 45–55, Piscataway, NJ, USA, 2015. IEEE Press.

[33] E. W. P. L. S. Z. Jürgen Cito, Gerald Schermann and H. Gall. An empirical analysis of the docker container ecosystem on github. In *2017 10th Working Conference on Mining Software Repositories (MSR)*, March 2017.

[34] J. Kabbedijk and S. Jansen. *TSteering Insight: An Exploration of the Ruby Software Ecosystem*, pages 145—-156. In Second International Conference on Software Business (ICSOB), Berlin, Heidelberg, 2011.

[35] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.

[36] Klint Finle. The problem with putting all the world's code in github. `http://www.wired.com/2015/06/problem-putting-worlds-code-github/`, accessed 2017-02-26, 2015.

[37] Lukas Martinelli. Haskell Dockerfile Linter. `https://github.com/lukasmartinelli/hadolint`, accessed 2017-02-07.

[38] Lukas Martinelli. Haskell Dockerfile Linter. `http://hadolint.lukasmartinelli.ch/`, accessed 2017-02-26.

[39] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The small project observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264 – 275, 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[40] Margaret Rouse. What is containerization (container-based virtualization) ? `https://http://searchservervirtualization.techtarget.com/definition/container-based-virtualization-operating-system-level-virtualizati` accessed 2017-02-26, 2016.

[41] Mario Ponticello. Docker Community Passes Two Billion Pulls! `https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/`, accessed 2017-02-25, 2016.

[42] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 141–150, New York, NY, USA, 2011. ACM.

[43] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan. A Large Scale Empirical Study of the Relationship Between Build Technology and Build Maintenance. 2014.

[44] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, MA, USA, 2003.

[45] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis. The bug catalog of the maven ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 372–375, New York, NY, USA, 2014. ACM.

[46] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, IC2E '15, pages 386–393, Washington, DC, USA, 2015. IEEE Computer Society.

[47] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 25(10):1117–1135, 2013.

[48] Open Container Initiative. runC - a lightweight universal runtime container. `http://runc.io/`, accessed 2017-03-07.

[49] Open Container Initiative. Why are all these companies coming together? `https://www.opencontainers.org/faq#n7`, accessed 2017-02-07.

[50] K. Plakidas, S. Stevanetic, D. Schall, T. B. Ionescu, and U. Zdun. How do software ecosystems evolve? a quantitative assessment of the r ecosystem. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 89–98, New York, NY, USA, 2016. ACM.

[51] J. Pérez, R. Deshayes, M. Goeminne, and T. Mens. Seconda: Software ecosystem analysis dashboard. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 527–530, March 2012.

[52] S. Raemaekers, A. van Deursen, and J. Visser. The maven repository dataset of metrics, changes, and dependencies. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 221–224, May 2013.

[53] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo. Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study. *Journal of Systems and Software*, 123:263 – 291, 2017.

[54] P. Rodríguez, J. Markkula, M. Oivo, and K. Turula. Survey on agile and lean usage in finnish software industry. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 139–148, New York, NY, USA, 2012. ACM.

[55] A. Scopatz. *Effective computation in physics : field guide to research with Python*. O'Reilly Media, Sebastopol, CA, 2015.

[56] Scott Hogg. Software Containers: Used More Frequently than Most Realize. `http://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html`, accessed 2017-02-07.

[57] A. Sensing. Terminology: Describing an item's evolution. `http://www.product-lifecycle-management.com/plm-revision-version.htm`, accessed 2017-03-07.

[58] A. Serebrenik and T. Mens. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 40:1–40:6, New York, NY, USA, 2015. ACM.

[59] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 1:1–1:13, New York, NY, USA, 2016. ACM.

[60] T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 189–200, New York, NY, USA, 2016. ACM.

[61] S. D. Simone. Google bigquery now allows to query all open-source projects on github. `https://www.infoq.com/news/2016/07/googlegithubbigquery`, accessed 2017-03-08, 2017.

[62] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

[63] Stackoverflow. Issues related to heap size in stackoverflow. `http://stackoverflow.com/search?q=heap+size`, accessed 2017-03-11, 2017.

[64] L. Torvalds. Re: Kernel SCM saga.. `https://marc.info/?l=linux-kernel&m=111288700902396`, accessed 2017-02-26, 2005.

[65] L. Torvalds. Google tech talk: Linus Torvalds on git. `https://www.youtube.com/watch?v=4XpnKHJAok8`, accessed 2017-02-26, 2007.

[66] L. Torvalds. Re: fatal: serious inflate inconsistency. `https://marc.info/?l=git&m=118143549107708`, accessed 2017-02-26, 2007.

[67] J. van Angeren, V. Blijleven, and S. Jansen. Relationship intimacy in software ecosystems: A survey of the dutch software industry. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, MEDES '11, pages 68–75, New York, NY, USA, 2011. ACM.

[68] R. Wettel and M. Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 921–922, New York, NY, USA, 2008. ACM.

[69] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 351–361, New York, NY, USA, 2016. ACM.

[70] Y. Yu. *Os-level Virtualization and Its Applications*. PhD thesis, Stony Brook, NY, USA, 2007. AAI3337611.

[71] S. Zumberi. Docker-evolution-name. `https://github.com/SaliZumberi/dockolution`.

[72] S. Zumberi. Docker parser. `https://github.com/SaliZumberi/dockerparser`.