2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

An Empirical Analysis of the Docker Container Ecosystem on GitHub

Jürgen Cito*, Gerald Schermann*, John Erik Wittern†, Philipp Leitner*, Sali Zumberi*, Harald C. Gall*

* Software Evolution and Architecture Lab
University of Zurich, Switzerland
{lastname}@ifi.uzh.ch

† IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
witternj@us.ibm.com

***Abstract*—Docker allows packaging an application with its dependencies into a standardized, self-contained unit (a so-called container), which can be used for software development and to run the application on any system. Dockerfiles are declarative definitions of an environment that aim to enable reproducible builds of the container. They can often be found in source code repositories and enable the hosted software to come to life in its execution environment. We conduct an exploratory empirical study with the goal of characterizing the Docker ecosystem, prevalent quality issues, and the evolution of Dockerfiles. We base our study on a data set of over 70000 Dockerfiles, and contrast this general population with samplings that contain the Top-100 and Top-1000 most popular Docker-using projects. We find that most quality issues (28.6%) arise from missing version pinning (i.e., specifying a concrete version for dependencies). Further, we were not able to build 34% of Dockerfiles from a representative sample of 560 projects. Integrating quality checks, e.g., to issue version pinning warnings, into the container build process could result into more reproducible builds. The most popular projects change more often than the rest of the Docker population, with 5.81 revisions per year and 5 lines of code changed on average. Most changes deal with dependencies, that are currently stored in a rather unstructured manner. We propose to introduce an abstraction that, for instance, could deal with the intricacies of different package managers and could improve migration to more light-weight images.**

***Keywords*-empirical software engineering; GitHub; Docker**

I. Introduction

Containerization has recently gained interest as a light-weight virtualization technology to define software infrastructure. Containers allow to package an application with its dependencies and execution environment into a standardized, self-contained unit, which can be used for software development and to run the application on any system. Due to their rapid spread in the software development community, Docker containers have become the de-facto standard format [1]. The contents of a Docker container are declaratively defined in a *Dockerfile* that stores instructions to reach a certain infrastructure state [2], following the notion of Infrastructure-as-Code (IaC) [3]. Source code repositories containing Dockerfiles, thus, potentially enable the execution of program code in an isolated and fast environment with one command. Since its inception in 2013, repositories on GitHub have added 70197 Dockerfiles to their projects (until October 2016).

Given the fast rise in popularity, its ubiquitous nature in industry, and its surrounding claim of enabling reproducibility [4], we study the Docker ecosystem with respect to quality of Dockerfiles and their change and evolution behavior within software repositories. We developed a tool chain that transforms Dockerfiles and their evolution in Git repositories into a relational database model. We mined the entire population of Dockerfiles on GitHub as of October 2016, and summarize our findings on the ecosystem in general, quality aspects, and evolution behavior. The results of our study can inform standard bodies around containers and tool developers to develop better support to improve quality and drive ecosystem change.

We make the following contributions through our exploratory study:

Ecosystem Overview. We characterize the ecosystem of Docker containers on GitHub by analyzing the distribution of projects using Docker, broken down by primary programming language, project size, and the base infrastructure (*base image*) they inherit from. We learn, among other things, that most inherited base images are well-established, but heavy-weight operating systems, while light-weight alternatives are in the minority. However, this defeats the purpose of containers to lower the footprint of virtualization. We envision a recommendation system that analyzes Dockerfiles and transforms its dependency sources to work with light-weight base images.

Quality Assessment. We assess the quality of Dockerfiles on GitHub by classifying results of a Dockerfile Linter [5]. Most of the issues we encountered considered version pinning (i.e., specifying a concrete version for either base images or dependencies), accounting for 28.6% of quality issues. We also built the Dockerfiles for a representative sample of 560 repositories. 66% of Dockerfiles could be built successfully with an average build time of 145.9 seconds. Integrating quality checks into the "docker build" process to warn developers early about build-breaking issues, such as version pinning, can lead to more reproducible builds.

Evolution Behavior. We classify different kinds of changes between consecutive versions of Dockerfiles to characterize their evolution within a repository. On average, Dockerfiles only changed 3.11 times per year, with a mean 3.98 lines of code changed per revision. However, more popular projects revise up to 5.81 per year with 5 lines changed. Dependencies see a high rate of change over time, reinforcing our findings to improve dependency handling from the analysis of the

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.67

ecosystem and quality aspects.
Before discussing an in-depth account of the empirical analyses of these topics in the respective sections IV, V, and VI, a brief introduction to containers and Docker is given.

II. Containers and Docker

Containers are based on an OS-level virtualization technique that provides virtual environments that enable process and network isolation. LXC[1] (Linux Containers) achieve this isolation through chroot, cgroups, and namespaces. As opposed to virtual machines (VMs), containers do not emulate another host operating system, but rather share the underlying kernel with the host. This significantly reduces the overhead imposed through VMs. Docker is an extension of LXC's capabilities that provides higher level APIs and functionality as a portable container engine. It aims to improve reproducibility of applications by enabling bundling of container contents into a single object that can be deployed across machines.

The contents of a container are defined through declarative instructions in a *Dockerfile*. Figure 1 illustrates a Dockerfile with explanations (left in black) and typical quality issues as defined by the Docker Linter [5] (red on the right). We provide brief explanations of some of the available instructions to provide a basic understanding of the format for the remainder of the paper. A Docker container can inherit infrastructure definitions from another container (*`FROM` instruction*). This can either be an operating system container, such as Ubuntu, but also any other existing container (e.g., with a pre-installed JDK installation). For maintenance purposes, a Dockerfile should provide the name and email of an active maintainer (*`MAINTAINER` instruction*). Dockerfiles can also set environment variables (*`ENV` instruction*). Both the *`ADD` and `COPY` instruction* allow to place files into the container. However, `ADD` has more functionality, as it allows the source path to be a URL, and if the source file is an archive (e.g., zip) it automatically unpacks the archive in the container. `RUN` is a rather general and thus powerful instruction, as it allows to execute any possible shell command within the container. It is often used to retrieve dependencies, and install and compile software. `EXPOSE` opens a specific port on the container to enable network communication. By definition, only *one process* can run within a container. This one process is specified with the instruction `CMD`.

III. Dataset

To enable our research, we retrieved a list of repositories that contain Dockerfiles from the public GitHub archive on BigQuery[2] in October 2016. The observation period for revisions and changes that we mined to analyze evolution behavior was from the first Dockerfile commit that appeared in the repository until January 2017. Our initial list contained over 320000 Dockerfiles in 80054 GitHub projects. We decided to remove repositories that were forks from other repositories to avoid biasing our analysis with duplicate entries. This would

[1]LXC: https://linuxcontainers.org/
[2]https://cloud.google.com/bigquery/public-data/github

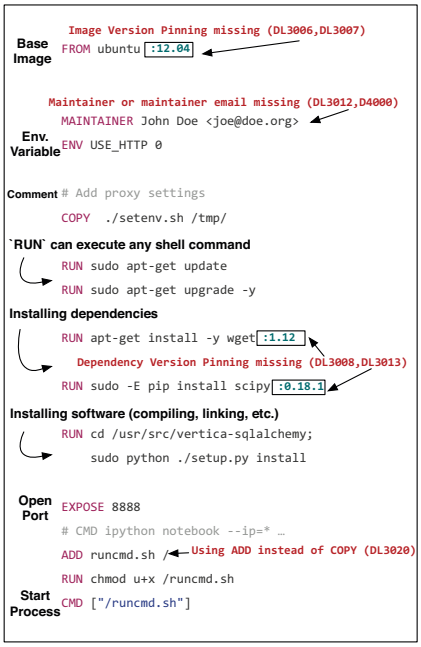

Fig. 1. Example of a fictitious Dockerfile illustrating common instructions and quality issues.

have been particularly problematic, as especially large, popular projects such as Kubernetes or nginx are forked frequently. The resulting study population for our analysis consisted of 70197 unique Dockerfiles originating from 38079 GitHub projects. Further, we queried the GitHub API to get additional metadata to our list, such as the owner type, owner name, used programming languages, project size, number of forks, issues, or the number of watchers.

To facilitate this mining process, we developed a Java-based tool chain that checks out the projects from GitHub and parses each Dockerfile and all its revisions into a relational data model (see Figure 2 for a simplified view of the underlying data model). Each project on GitHub stores one to many *Dockerfile* entities that contain metadata about the repository and the file (*70197 Dockerfiles* distributed across *38079 projects*). A *Dockerfile* contains one to many *Instruction* entities (*897186 instructions* over all Dockerfiles), each of them having one to many *Parameter* entities (*4611272 parameters*). Each *Dockerfile* entity stores one to many *Revision* entities (*218259 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. For two consecutive revisions (before and after a change) of Dockerfiles, we compute structural differences, and store the entity *Diff* with one to many *Structured Change* entities for each instruction (*1483763 changes*). We categorized different types of differences (*Change Type*): ADD, MODIFY, DELETE, with subcategories for each instruction, which enables more fine-grained evolution analysis.

One goal of our study is to compare Docker usage in the general population of GitHub projects with how the tool is

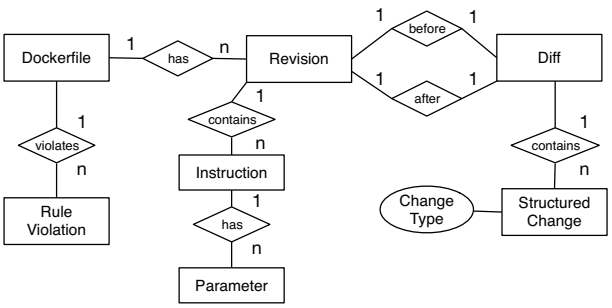

Fig. 2. Relational data model to support queries for data analysis.

used in popular projects. Hence, we introduce two additional samplings in the database: *Top-100* and *Top-1000* repositories containing Dockerfiles. To retrieve these two samples, we order the repositories by the number of star gazers (a measure for popularity) and select 100 and 1000 unique repositories respectively.

In order to foster reproducibility and follow-up studies, we provide a comprehensive reproducibility package, containing tool chain, database, queries and analysis scripts. The tool chain consists of two separate Java projects. **dockerparser**[3] is responsible for parsing and storing Dockerfiles in a relational database (Postgres), and **dockolution**[4] computes structural changes between all revisions in a repository. We exported our entire database (*15 GB*) and make it available together with all of our analyses (SQL queries to the database, R and Python scripts to produce plots) on GitHub[5] for inspection and reproduction.

IV. The Docker Ecosystem

To learn about the Docker ecosystem, we investigate what types of projects use Docker, and what Docker is used for in these projects.

To answer the first question, we analyze characteristics of the projects containing Dockerfiles. Figure 3 shows histograms of the size of projects in our data set, as reported by the GitHub REST API.[6] We consider, on the one hand, the overall data set and on the other hand the Top-100 and Top-1000 projects as described in Section III. Noticeably, projects in the Top-1000 and even more so projects in the Top-100 are of larger size. This effect, however, may be expected for more popular projects in general, and must not necessarily be true only for projects using Docker. Of all repositories, an unusual high amount 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 – possibly being used as a dedicated repository to separate packaging or deployment concerns from an application.

[3]https://github.com/sealuzh/dockerparser
[4]https://github.com/sealuzh/dockolution
[5]https://github.com/sealuzh/docker-ecosystem-paper
[6]https://developer.github.com/v3/

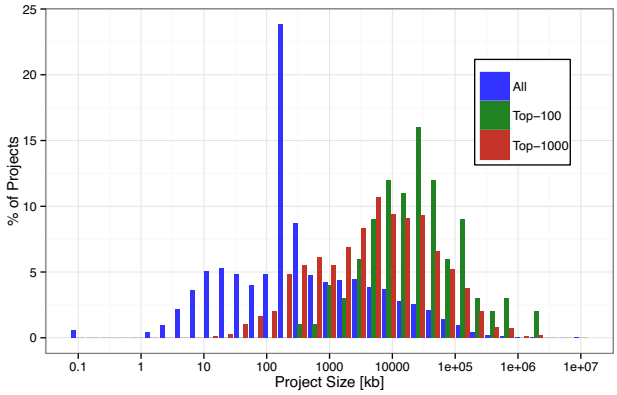

Fig. 3. Distribution of project sizes

A. Programming Language Distribution

Figure 4 shows the distribution of *primary* programming languages across the assessed projects, meaning the one language per project that accounts for the majority of source code (as measured in file sizes). We fetched the distribution of programming languages per project using, again, GitHub's public REST API. In addition to our core data set, Figure 4 presents excerpts from the GitHub data available on Google's BigQuery service.[7] This data set contains over 2.8 million GitHub repositories, from which we queried the distribution of primary programming languages. Figure 4 shows that Docker projects are frequently dominated by shell scripts, while such repositories in general are much less common on GitHub. In addition, we see that specific languages like `Go` are relatively strongly used in Docker projects, while others like `PHP` or `Java` are relatively underrepresented.

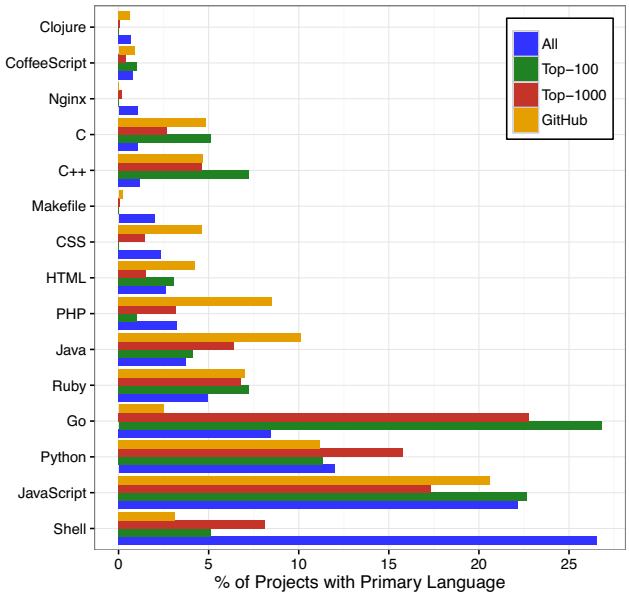

Fig. 4. Distribution of top 15 languages in our dataset

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

B. Base Images

A first indication of what it is that projects use Docker for can be derived from the base images specified in the Dockerfiles (cf. Section II). 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, for example `ubuntu` or `node`, the name is the sole identifier of an image. Non-official images further depend on a namespace, which is often the name of the organization or user who maintains the image. In addition, 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`). In sum, the Dockerfiles assessed contained 9298 unique base image specifications. Figure 5 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, of the 15 most common base images, 14 are official ones – among all 9298 images, only 130 (1.4%) are official.

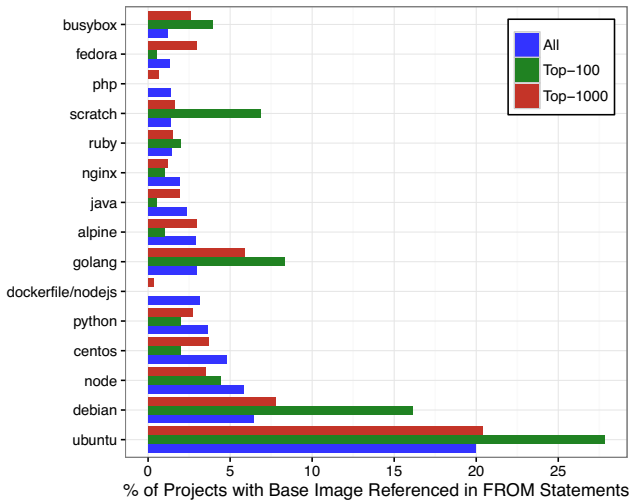

Fig. 5. Percentage of usage of 15 most commonly used base images

To further summarize the use of base images, we manually classified the top 25 ones used in the overall data, the Top-1000, and the Top-100 into 5 types. This approach allows to cover significant parts of the overall data, as the 25 most commonly used images account for 64% of usage across all images. "OS" are base images that contain a blank operating system, without further software being installed. "Language runtime" images, in addition, contain a runtime needed to execute applications written in a specific programming language. "Application" base images come bundled with an application, for example, a database or a Web server. We marked base images specifications that contained placeholders for parameters to be filled out at runtime as "variable", e.g., `{{namespace}}/{{image_prefix}}base:{{tag}}`. Finally, "other" denotes images that do no fit cleanly into any of the above categories, for example `scratch`, an empty image, or `busybox`, a collection of UNIX utilities. Operating system and language runtime base images clearly dominate, in all three data sets. 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 actual usage amounts of base images, which could possibly present a different picture all together. Among the top OS images shown in Figure 5, image sizes differ: while `alpine` is only about 4 MB small, `debian` and `ubuntu` are both around 125 MB, and `centos` and `fedora` are around 195 MB big.

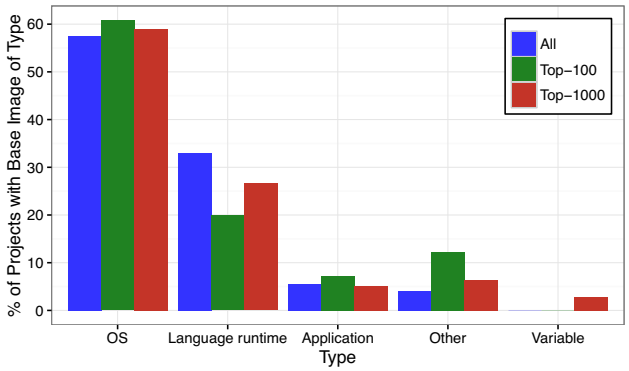

Fig. 6. Percentage of base image types across top 25 images

C. Instructions

Indications of how projects build upon these images can be obtained from the instructions used in the Dockerfiles. Table I 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. We omitted results for `LABEL` and `ARG` because their occurrence in all categories constituted for less than 0.1%. `RUN` is the instruction used by far the most in Dockerfiles. However, this is not surprising considering the generic nature of the instruction that allows to execute any viable (non-interactive) shell command within the container. Taken together, `RUN` and `COMMENT` instructions make up 56% of a typical Dockerfile. Notably, looking at the usage of instructions within repositories, the mandatory `FROM` command is omitted in some projects in all three views on the data, indicating a Dockerfile that will fail to execute. The remaining instructions (`ADD`, `CMD`, `COPY`, ...) are used in many Dockerfiles, but only represent small fractions of Docker code.

Most instructions are used relatively evenly across the three views on the data (entire population, Top-1000, Top-100). One exception is the significantly lower use 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, used

TABLE I
PERCENTAGES OF INSTRUCTIONS AND REPOSITORIES USING INSTRUCTIONS AT LEAST ONCE

Instruction	Instructions			Repos using instructions (at least once)		
	All	T-1000	T-100	All	T-1000	T-100
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
USER	0.01	0.01	0.01	0.10	0.10	0.08

to set the directory for RUN, CMD, ENTRYPOINT, COPY, and ADD instructions to run in, which 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.

Given that RUN instructions are used so prevalently to implement Dockerfiles, we were interested in a narrower breakdown as to what kind of commands are being issued within containers. We categorized the commands executed in RUN by first sorting them 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), and finally *other* (for any remaining commands). We summarize our analysis of RUN instruction breakdown in Table II. The majority of commands (>70%) can be classified as belonging to either *dependencies* (∼45%) or *filesystem* (∼30%). There are not too many large differences between the overall population and the top projects on GitHub. Interestingly, Top-100 projects use 13.5% *build/execute* commands, compared to only 5.3% in all projects. A potential explanation is that popular projects are often themselves used as dependencies for other projects and thus need more commands to build their own source.

TABLE II
BREAKDOWN OF RUN INSTRUCTIONS IN SIX CATEGORIES

Category	Examples	Instructions		
		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

D. Takeaways

Our ecosystem analysis yielded some interesting results around the characteristics of projects that use Docker on GitHub. Based on these results, we derived takeaway messages concerning *image size*, *base image recommendation*, and *instruction abstractions*.

Image Size. Our results show that most 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 *125MB* [6]. Other base images contain light-weight operating systems, such as Alpine, that are as small as *4MB* [6]. Most projects use the larger OS as base images. This is probably due to the convenience of working with a known, established system and their widely used package managers to handle dependencies. However, that defeats the initial purpose of using containers to lower the footprint of virtualization. The container ecosystem would benefit from tools that minimize image size and improve build time and reduce space requirements as a consequence.

Base Image Recommendation. Following the insight from our previous section, we envision a recommender system that analyzes an existing Dockerfile and produces transformations such that the same application can be run with a different base image to reduce the overall size and preferably also build time. Even small reductions in size can have tremendous effects when applied at scale. The size of an image can have an effect for continuous integration (CI) when an application consists of multiple microservices [7] that are built with Docker. In the same vein, when these services are deployed within an orchestration system (e.g., Kubernetes), the size of a container makes quite a difference for schedulers.

Instruction Abstraction. Our analysis shows the most commonly used instruction to be the generic RUN (apart from the mandatory FROM). When breaking down the RUN statement, we saw that around 45% are used to define dependencies. Given these results, we argue that there might be a benefit by introducing a new abstraction for defining dependencies in a more explicit manner. This would also help with transitioning to different base OS images, when the implementation underneath the dependency abstraction layer can take care of, for instance, which package manager should be used.

V. QUALITY OF DOCKERFILES AND STANDARD COMPLIANCE

A. Most Violated Rules

Similar to various programming languages or other IaC languages (e.g., Puppet, Chef), a set of best practices evolved [8] for describing images in Docker's declarative language. Best practices are not only helpful to avoid common mistakes (e.g., using ADD instead of COPY), but also foster the traceability of images (e.g., the maintainer's contact information). In order to identify how Docker repositories on GitHub comply to those best practices we relied on a Dockerfile linter [5], which is based on the aforementioned best practices and contributions by the GitHub community. This linter parses a given Dockerfile and checks adherence to a set of rules representing the best practices on top of the resulting AST. We executed the linter for every Dockerfile in our data set and added the reported rule violations to our database.

TABLE III
12 MOST VIOLATED RULES ACROSS ALL REPOSITORIES, TOP 100 AND TOP 1000 REPOSITORIES BASED ON THEIR STAR RATING, AND A RANDOM SAMPLE OF 560 BUILT DOCKERFILES.

	others	**apt-get upgrade** (DL3005)	**pip version pinning** (DL3013)	**maintainer email** (DL3012)	**image :latest** (DL3007)	**use WORKDIR** (DL3003)	**double quote** (SC2086)	**img. version pinning** (DL3006)	**delete apt-get lists** (DL3009)	**avoid add. packages** (DL3015)	**copy instead of add** (DL3020)	**apt-get vers. pinning** (DL3008)	**maintainer missing** (DL4000)
All (n=33670)	8.8%	1.3%	2.0%	2.2%	3.4%	4.2%	4.3%	9.7%	10.0%	11.9%	12.8%	13.5%	15.9%
Top-100	9.5%	0.6%	4.8%	1.3%	3.2%	7.1%	4.8%	4.3%	11.1%	12.2%	5.0%	14.5%	21.7%
Top-1000	7.9%	0.9%	2.2%	1.6%	2.9%	5.5%	5.3%	7.7%	9.1%	11.1%	12.2%	13.8%	19.8%
Build Failure (n=203)	10.8%	0.4%	2.3%	1.7%	3.1%	4.7%	3.5%	14.9%	7.6%	10.8%	17.5%	11.4%	11.4%
Build Success (n=357)	6.6%	0.7%	1.9%	1.7%	4.1%	3.7%	6.4%	7.7%	10.0%	13.1%	13.6%	14.8%	15.6%

In addition to checking rule violations across the entire population, we wanted to identify how many Dockerfiles build successfully. Since building all Dockerfiles is not feasible, we selected a random sample of 560 repositories (confidence level 95%, confidence interval 4). To avoid transient effects (e.g., connection speed, network time out) we repeated "docker build" three times for each Dockerfile and collected the final build outcome (i.e., success or failure) and the build time in seconds. To avoid caching effects due to Docker's layering approach, we removed all images and containers after every single build. For six repositories we obtained both successful and failed builds, and thus conducted another build of the respective Dockerfile. For all six repositories this resulted in either three successful or three failed builds and a single outlier (likely due to transient effects). We replaced the outlier by the additional run to ensure a stable set of failed and successful builds.

Table III shows the 12 most violated rules reported by the linter. The severity of the rule violations vary, hence missing contact information of the maintainer is not as crucial as rules regarding version pinning, as the usage of incompatible versions might lead to failed builds. After providing a short overview about the quantity of rule violations we focus on specific rules and the differences we identified among the considered segments of the population. Detailed information and examples for each rule not covered in detail (e.g., DL3003) can be found in the linter's GitHub repository [5].

Overview. Across the entire population, Dockerfiles violate 3.1 rules on average, while Dockerfiles of the Top-100 most prominent projects violate 3.2 rules and Dockerfiles of the Top-1000 projects 3.5 on average. Focusing on the Top-100 and their Dockerfiles, 19 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 16% of the repositories.

Version Pinning. The linter checks for four types of version pinning: 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`). In both cases, the absence of a concrete version can lead to the usage of a version which is not compatible with the other components of the container, thus to failed builds, or failures hitting surface only at container execution. Therefore, best practices suggest to specify concrete versions. 9.7% of all Dockerfiles across the entire population violate the rule of image version pinning. Interestingly, when looking at the Top-100 repositories, just 4.3% of the considered Dockerfiles violated this rule. Therefore, more popular repositories might be more aware of bad practices associated with build failures. However, we also identified that in case of the installation of specific packages (i.e., pip and apt-get version pinning), the most popular repositories perform worse than the entire population, even though on a smaller scale. We assume that this is related to more sophisticated Dockerfiles, hence more dependencies and therefore more `RUN` instructions to violate those rules. This assumption is supported by our finding that the Top-100 projects have on average more `RUN` instructions. That the absence of concrete versions could be problematic for the build success is supported by the finding that 14.9% of our repositories with failed builds violate the rule of image version pinning, compared to only 7.7% of repositories with successful builds. Violations because of the ":latest" tag for the base image are uncommon, thus developers seem to be widely aware of this problem.

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. It allows, for example, downloading resources from a URL and automatically unpacks compressed local files (e.g., tar, zip, etc). This additional "magic" is considered dangerous and can lead to accidental failures as long as developers are not aware of it, e.g., when he/she wants to add a zipped folder to the image and `ADD` automatically unpacks it. Across all Dockerfiles, still, 12.8% violated the practice to prefer `COPY`. Again, this is not the case for the Top-100 repositories with only 5%. While the sample of successful builds (13.6%) is in the range of the entire population, 17.5% of the repositories with failed builds violated this rule, making it to the top violated rule of failed builds.

Missing Maintainer Information. Regarding maintainer

information (DL4000) we detected diverging results across the different segments of the population. 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. We assume 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.

B. Build Analysis

We were able to successfully build 357 images from the 540 Dockerfiles in our sample. Compared to results reported for building Ruby (72.7%) and Java-based (82.4%) projects on GitHub using TravisCI [9] the success ratio of Docker builds (66%) is lower. Table IV provides descriptive statistics for the measured build time and Figure 7 shows the distribution of the build time using a density plot.

TABLE IV
BUILD TIME STATISTICS OF FAILED AND SUCCESSFUL DOCKERFILE BUILDS IN SECONDS.

	Failed	Successful
%	0.34	0.66
mean	90.5	145.9
median	24.7	76.7
min	0.0	0.7
max	1539.0	2742.0
sd	197.4	264.0

The average standard deviation between the three build runs is 9.1 seconds, 13.9 seconds for failed builds and 6.3 for successful builds. Due to the different nature of the various projects, the overall standard deviation of build duration is higher (see Table IV). Based on our sample, it can take up to 1539 seconds and on average 90.5 seconds until a build fails. In comparison, for Java projects built on TravisCI, the average duration (i.e., build duration excluding latencies such as VM scheduling) of failed builds is 9.7 seconds. We used the TravisTorrent [10] data set to retrieve this value. This difference regarding build duration means that while Java developers have almost immediate feedback about a build's outcome, it takes substantially longer for Docker builds. Such long build durations are especially problematic if the step to build the image is part of a continuous deployment or delivery pipeline [11]. The idea of fast, frequent releases [12] with immediate customer feedback is impeded by long lasting build processes. Docker's strategy of splitting images into multiple layers addresses this problem and speeds up the process by only downloading those resources (e.g., packages) which are not yet available locally. However, when utilizing CI services such as TravisCI or CircleCI, it is not guaranteed that a new build happens on the same instance (e.g., virtual machine) as the previous build. Therefore, for each build, all resources and thus layers have to be downloaded again, slowing down the overall build process. An implication for service providers such as TravisCI would be to provide means to cache those layers. Only recently with version 1.13, Docker released a feature (i.e., "`--cache-from`") allowing CI service providers to address this problem and thus confirms our finding of this critical practical issue. However, at the time of the submission of this paper, there was not yet any practical implementation of it.

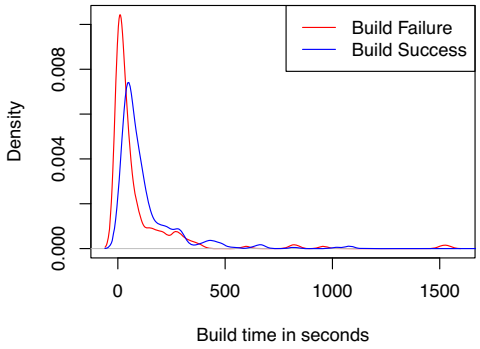

Fig. 7. Build time density

C. Multi-Process Management

Traditionally, a Docker container runs 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). 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. However, there are cases where it is practical to run multiple processes inside a container (e.g., the actual application and a `ssh` server for being able to connect to the running container). Since doing this manually is known to be error-prone, there are third party tools addressing this issue (e.g., *supervisord*, *runit*, *monit*).

We investigated how prevalent the problem of multi-process containers is, and what tools developers prefer. To that end, we analyzed the parameters supplied to `ENTRYPOINT` and `CMD` instructions. We identified that *supervisord* is by far the most common tool for multi-process management. 1357 repositories called *supervisord* in the `CMD` instruction, and 152 in the `ENTRYPOINT` instruction. *s6* (i.e., *s6-svscan*) is on the second place and used by 29 repositories in the `CMD` instruction, and 2 in the `ENTRYPOINT` instruction. Other tools including *monit*, *runit*, *system.d*, and *upstart* are used by 10 and less repositories.

In total, 1581 projects (i.e., 4% of all Docker projects) make use of such tools. We identified no usage within the Top-100 and 52 projects within the Top-1000. This is less than we expected due to the prominent coverage of these approaches in online resources (e.g., developer blogs). As we only analyzed the execution of multi-process tools in `ENTRYPOINT` and `CMD` instructions our results might be lower than the actual population. Dockerfiles allow the execution of shell scripts in both the `ENTRYPOINT` and `CMD` instructions, thus the calls to such tools can be "outsourced" to separate scripts. To check

our results, we analyzed whether respective `RUN` instructions install these tools using common package managers such as *apt-get*, *yum*, or *dpkg*. However, our results for *supervisord* were on the same level as before, but we counted a higher installation rate of *system.d* with 184, *runit* with 105, and *monit* with 25 repositories. Those additional installations might be either called in those separate shell scripts, or never used. However, compared to the entire population of projects, those tools our only used by a minority, and thus also confirmed our initial results on `CMD` and `ENTRYPOINT` instructions.

D. Takeaways

Based on our analysis regarding the compliance to standards and best practices, and our build time analysis we derived takeaway messages concerning *quality check integration*, *build acceleration*, and *multi-process support*.

Quality Check Integration. During our analyses we identified that the Docker linter provides valuable feedback concerning the quality of Dockerfiles. We assume that especially the rules tackling version pinning help developers to prevent build failures. Dockerfiles that do build successfully right now might not do so in future when dependencies break. Specifying concrete versions for both the base image and all other dependencies is the way to mitigate these problems. Therefore, our implication for the Docker tool chain is that it would be beneficial to directly include such checks into the "Docker build" process. Developers should receive at least warnings when they try to build an image from a Dockerfile which violates rules that might influence the build result (e.g., version pinning, `COPY` vs. `ADD`). This would foster awareness and directly pinpoint developers to problematic instructions.

Build Acceleration. We have seen that, for example in comparison to failed Java builds, failed Docker builds take fundamentally longer. The larger the chosen base image and the more dependencies are required, the longer it takes to download all required resources. CI service providers still need to find solutions to cache images (i.e., their layers), and speed up these time consuming downloads [13]. With its current version (1.13) Docker has provided a first step tackling this issue. Besides technical solutions, service providers might recommend developers to reconsider their overall build and deployment process and to rely on already built images hosted on platforms such as Docker Hub. However, this would require strictly separating the process of building the image from the CI part of the application (e.g., running tests) and container deployment.

Multi-Process Support. Given our finding that tools such as *supervisord* are only used by a minority of the Docker repositories (4%), we do not see any indication to consider multi-process support as first-class citizen for Docker. Further analysis is required to determine why those 4% have to rely on multi-process tools and why the common design practice of isolating single functionality is not sufficient for them.

VI. Maintenance and Evolution

We now turn towards analyzing the maintenance and evolution of Dockerfiles. We consider revisions, where every revision is formed by a commit that added, removed, or modified at least one line in a Dockerfile. For every Dockerfile, we use revisions per year as a metric to define how often it is updated. We count the original commit as a revision, so every Dockerfile has at least one revision.

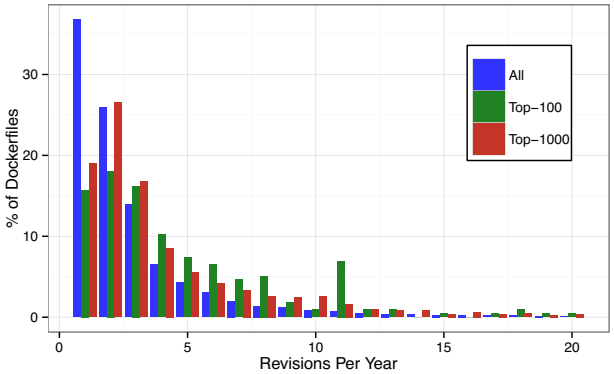

Fig. 8. Distribution of yearly revisions

A. Rate of Change

Figure 8 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 3.11. However, this mean is biased, as the distribution appears to closely follow a power law, with 62.27% of Dockerfiles being revised 0 or 1 times per year after the initial commit. Interestingly, Dockerfiles belonging to the Top-100 and Top-1000 projects are in fact updated substantially more often, with an arithmetic mean of 5.81 and 4.70 revisions per year respectively. Taken together with the results discussed in Section IV, we can conclude that more popular projects are not only larger, but also maintained more actively. For the Top-100 and Top-100 samples, no power law can be observed. Instead, in both cases, the most common number of revisions per year is 2. Further, the slope of the distribution is flatter than predicted by a power law, i.e., more Dockerfiles are revised more frequently. Finally, for the Top-100 projects, we observe that a surprisingly large percentage of Dockerfiles is updated 12 times per year. We assume this spike to be a statistical anomaly that is due to the relatively low sample size in the Top-100 projects category (216 distinct Dockerfiles).

1) Magnitude of Change: Another relevant dimension to this question is how large, and of what kind, revisions to Dockerfiles are when they happen. For this analysis, we filter out all initial commits, so that we only consider revisions that actually modify an already existing Dockerfile. Our data set shows that such revisions are typically small, with on average only 3.98 lines of code in Dockerfiles changed. 80.39% of all revisions consist of 5 lines of Docker code changed or less. Figure 9 depicts the number of total, added, removed, or

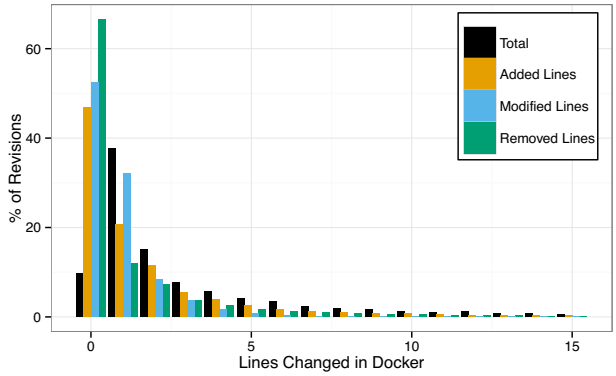

Fig. 9. Distribution of revision sizes measured in lines changed

modified lines of Docker code per revision. For this analysis, we have not observed a significant difference for the Top-100 and Top-1000 projects as compared to the entire data set. Hence, we have chosen to only plot the entire data set in Figure 9. Most changes to Dockerfiles consist of instruction additions (47.44%), followed by the removal (33.17%) and modification (19.40%) of instructions.

B. Nature of Change

Table V drills deeper into the nature of changes to Dockerfiles. This table lists all currently supported instruction types as discussed in Section II, and specifies how many changes refer to an instruction of the respective type. We omitted results for `LABEL`, `USER`, `VOLUME`, and `ARG` because their changes in all categories constituted for less than 0.1%. 53% of all changes relate to `RUN` instructions. Given that `RUN` instructions only make up 40% of all instructions (cf. Table I), this type of instruction appears to be changed more frequently than others. Drilling down into the types of changed `RUN` instructions (e.g., dependency management, file system) reveals a similar distribution as for Dockerfiles in general (cf. Table II). Interestingly, `RUN` instructions are often added or removed (59% and 61% respectively), but much more rarely modified (29%). Conversely, `FROM` instructions are frequently modified (29%), but almost never added or removed. This can be explained by the special nature of this instruction – it is used to specify a base image and is required for a Dockerfile that can be executed. Consequently, Dockerfiles typically contain exactly one `FROM` instruction. An update to this instruction may indicate a switch to a different base image, or a version change. Table V also 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 (26% for the Top-100 versus 29% in the entire population), which may be explained by projects with a large user base being more conscious of significant changes such as updating the base image.

C. Takeaways

Given our results regarding maintenance and evolution, we can reinforce the implications from the previous two sections.

Dependency Evolution. Dependencies are over-proportionally represented in the change behavior of Dockerfiles. We see a high rate of change regarding dependencies as an opportunity to echo our findings from before. First, introducing an explicit abstraction (i.e., instruction) to define dependencies can support the specific scenarios of change. Second, pointing developers to issues concerning dependencies (e.g., version pinning) early in the process may avoid breaking the build and reduce the need for change.

Structured Diffs. Most changes manifest as unstructured (or semi-structured at best) text. When developers observe and deal with change in Dockerfiles, they often deal with it on a line-by-line basis through a built-in history analysis tool such as `diff`. Developers could benefit from tooling that allow for structural differencing between two consecutive versions of infrastructure. Similar approaches have already been established for program source code [14], [15].

VII. Threats to Validity

We now discuss the main threats to the validity of our study.

Construct Validity. An essential threat to construct validity is that the relational model we designed 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. Our tooling is based on the assumption that Dockerfiles follow the standard naming convention `Dockerfile` without supplied file type. We have mitigated this threat 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, we assume 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. We 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. To extend the generalizability of our study we considered the entire population of Docker repositories on GitHub. 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. We only considered Docker repositories on GitHub. Consequently, it is not assured that our outcomes generalize to projects hosted on other services, such as Bitbucket. Moreover, we solely analyzed

TABLE V
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

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. Our analyses are only based on Docker and do not include other container technologies such as Linux containers (LXC). Finally, our results regarding the build quality and duration are limited and not entirely generalizable as we conducted our analysis only on a randomly selected subset (n=560) of the population.

VIII. RELATED WORK

With the increase in relevance of modern Web engineering concepts, such as continuous deployment [16] and cloud computing [17], empirical research on how to quickly build and deploy code on virtualized infrastructure has received more attention. Seminal early work on build systems has been conducted by McIntosh et al., who studied both, Java-based build systems [18] and building software more generally [19]. In cloud-based systems, Infrastructure-as-Code (IaC) has been identified as the foundation that enables elasticity and large-scale deployments [20].

So far, there is little research on how to develop and maintain IaC code, even though infrastructure code has been shown to often contain "code smells" [21] or to otherwise behave unexpectedly. For instance, following Hummer et al. [2], about a third of popular recipes for the often-used IaC language Chef are not idempotent, as required by Chef. Jiang and Adams have analyzed the co-evolution of IaC code with production and test code, and compare them with build files [3]. They conclude that IaC code is tightly coupled with test code, and is adapted about as often.

Existing empirical studies related specifically to Docker typically focus on performance aspects, often comparing container performance and overhead with traditional virtualization techniques [22], [23], [24]. However, despite this shortage of empirical work, the importance of Docker for academia and industry is rarely doubted in literature. For instance, Boettiger and Cito et al. have concurrently proposed that containerization technology may be an important game changer in making systems and software engineering research more reproducible [4], [25]. In industry, Docker is increasingly being used to build next generation Platform-as-a-Service clouds [26].

Despite this importance, to the best of our knowledge, no existing research has investigated the quality of Docker IaC code, the evolution of Dockerfiles, nor empirically studied the open source ecosystem of Docker. With this paper, we attempt to address this research gap, and provide insights into the practical usage of Docker in the open source community.

IX. CONCLUSION

We conduct the first large-scale empirical study to analyze the ecosystem, quality aspects and evolution behavior of Docker containers on Github. Our study is based on 70197 Dockerfiles from 38079 projects, which is the entire population of non-forked projects as of October 2016. We find that most containers inherit infrastructure from heavy-weight operating systems, most probably out of convenience, which defeats the purpose of container virtualization of reducing its footprint. We also find that 28.6% of quality issues (as indicated by the Docker Linter) arise from missing version pinning. Further, we were not able to successfully build 34% of Dockerfiles from a representative sample of 560 projects. Integrating quality checks, e.g., to issue version pinning warnings, into the container build process could result into more reproducible and stable builds in the future. Finally, we have observed that Dockerfiles are not changed often, with a mean of 3.11 to 5.81 revisions per year. Most of these deal with dependencies, which are currently not explicitly managed and dealt with in Docker. We argue that introducing an abstraction that could deal with the intricacies of different package managers and could improve migration to more light-weight images in the future. This would help Docker live up to its claim as a light-weight alternative to standard virtualization.

ACKNOWLEDGMENT

The research leading to these results has received funding from the Swiss National Science Foundation (SNSF) under project names "Whiteboard" (Project no. 149450), "Minca" (Project no. 165546), and the CHOOSE Forum.

REFERENCES

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

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

[3] 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*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 45–55. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820527

[4] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2723872.2723882

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

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

[7] G. Schermann, J. Cito, and P. Leitner, "All the Services Large and Micro: Revisiting Industrial Practice in Services Computing," in *Proceedings of the 11th International Workshop on Engineering Service Oriented Applications (WESOA'15)*, 2015.

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

[9] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An analysis of travis CI builds with github," *PeerJ PrePrints*, vol. 4, p. e1984, 2016. [Online]. Available: http://dx.doi.org/10.7287/peerj.preprints.1984v1

[10] ——, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[11] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[12] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. Gall, "An empirical study on principles and practices of continuous delivery and deployment," PeerJ Preprints, Tech. Rep., 2016.

[13] G. Schermann, J. Cito, P. Leitner, and H. C. Gall, "Towards quality gates in continuous delivery and deployment," in *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 2016, pp. 1–4.

[14] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, 2007.

[15] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[16] 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*, vol. 123, pp. 263 – 291, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215002812

[17] 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.*, vol. 25, no. 6, pp. 599–616, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.future.2008.12.001

[18] 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*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985813

[19] 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.

[20] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 393–403. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786826

[21] 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*, ser. Middleware '16. New York, NY, USA: ACM, 2016, pp. 1:1–1:13. [Online]. Available: http://doi.acm.org/10.1145/2988336.2988337

[22] R. Morabito, J. Kjllman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, ser. IC2E '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 386–393. [Online]. Available: http://dx.doi.org/10.1109/IC2E.2015.74

[23] 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)*, vol. 00, pp. 171–172, 2015.

[24] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901761

[25] J. Cito, V. Ferme, and H. C. Gall, *Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research*. Cham: Springer International Publishing, 2016, pp. 609–612. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-38791-8_58

[26] 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*, ser. IC2E '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 610–614. [Online]. Available: http://dx.doi.org/10.1109/IC2E.2014.41