2018 42nd IEEE International Conference on Computer Software & Applications

An Insight into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency

Yang Zhang, Gang Yin, Tao Wang, Yue Yu, Huaimin Wang
Key Laboratory of Parallel and Distributed Computing, College of Computer,
National University of Defense Technology, Changsha, 410073, China
{yangzhang15, yingang, taowang2005, yuyue, hmwang}@nudt.edu.cn

***Abstract*—Containerization is a software development approach aimed at packaging an application together with all its dependencies and execution environment in a light-weight, self-contained unit, of which Docker has become the de-facto industry standard. By defining the specific Docker image architecture and building orders, dockerfile plays an important role in the Docker-based containerization process. Understanding the evolution of dockerfile and which dockerfile architecture attributes enhance dockerfile quality and reduce image build latency can benefit the efficient processing of containerization. In this paper, we perform an empirical study on a large dataset of 2,840 projects to shed light on the impact of dockerfile evolutionary trajectories on quality and latency in the Docker-based containerization. Based on the six categories of dockerfile evolutionary trajectories we discovered, we build two regression models to explore the impact of dockerfile evolutionary trajectories and specific architecture attributes on dockerfile quality and image build latency, which derives a number of suggestions for practitioners.**

***Index Terms*—Docker, dockerfile, evolutionary trajectory, containerization, project maintenance**

I. Introduction

Containerization is the virtualization technology that enables packaging an application together with all its dependencies and execution environment in a light-weight, self-contained unit. In particular, Docker[1] containers have become the de-facto industry standard, and their usage is spreading rapidly [1]. Docker *images* of the packaged application can be specified declaratively, versioned together with the rest of the infrastructure code, built automatically, and published to some cloud-based registry. The most common such registry is Docker Hub[2], which hosts close to 1,700,000 Docker image repositories at the time of December 2017.

The content of the Docker image is defined by declarations in the *dockerfile* which specifies the Docker commands and the order of their execution for creating the desired images [2]. Since defining the specific image architecture and building orders, dockerfile plays an important role in the Docker-based containerization process. During our preliminary statistics, nearly 70% of projects in our original data set has changed their dockerfile at least once. Over 5,000 projects changed their dockerfile more than 10 times. Among individual projects, the evolution of dockerfile may vary, because different projects have different codes and goals. But, for projects with similar requirements and structure, a similar dockerfile evolutionary trajectory may exist, which has been validated in our prior study [3]. Understanding which categories of dockerfile evolutionary trajectories and which specific architecture attributes, *e.g.*, number of layers, layer size, *etc.*, enhance dockerfile quality and reduce image build latency can bring benefits to the efficient processing of containerization.

In this paper, we seek to aid in finding dockerfile best practices, by collating lessons learned from different dockerfile configuration practices in open source software (OSS) projects and summarizing their important messages. To this end, we focus on the OSS projects on GitHub, the largest public code repository host. Fortunately, many of the Docker Hub images are linked to a GitHub repository, which enables mining and study. Starting from these two public sources of data, GitHub and Docker Hub, we explore how people write their dockerfile and what the differential benefits among specific dockerfile architectures are, through our quantitative study. We collected data from 2,840 projects, involving Docker Hub builds and GitHub Git logs about dockerfile. Based on the six categories of dockerfile evolutionary trajectories we found in our prior study (see §II-B), we separately regressed dockerfile quality and image build latency over variables of interest and various controls, investigating how the dockerfile evolutionary categories and specific dockerfile architecture attributes affect the Docker-based containerization outcomes.

We mainly examine the following two research questions:
RQ1: *Do dockerfile evolutionary trajectories and architecture factors affect the dockerfile quality?* (see §IV-A)
RQ2: *Do dockerfile evolutionary trajectories and architecture factors affect the build latency of docker image?* (see §IV-B)

The highlights of our findings are:

- Different categories of dockerfile evolutionary trajectories have different effects on the containerization outcomes. In general, *C-1* (dockerfile scale is increasing and holding) has less dockerfile quality issues and shorter build latency.
- More diverse instructions, fewer image layers, fewer descriptions, and using an official image would help reduce dockerfile quality issues.
- To shorten image build latency, developers should cut down both the number of image layers and each layer's size. They should diversify instructions and control their operations locally without accessing external resources.

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

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00026

In the following, we introduce related concepts and our prior study results in §II, followed by our research methods (§III) and results (§IV). We discuss the significance of our contributions in §V and offer some concluding remarks in §VI.

II. Preliminaries

A. Docker and Dockerfile

Docker [4], as an OSS project that implements operating system-level virtualization, is built on many technologies from OS research, *e.g.*, LXC [5] (Linux Containers) and virtualization of the OS [6]. The technology is mainly used by developers to create and publish the *containers* [7]. With containers, applications can share the same OS and, whenever possible, libraries and binaries [8]. Docker launches its containers from the *Docker image*, which is a series of data layers on top of a base image [9]. When developers make changes to a container, instead of directly writing the changes to the image of the container, Docker adds an additional layer containing the changes to the image [10]. Since its inception in 2013, a large number of GitHub projects have used Docker [1]. These facts prompted us to conduct our study on the Docker-based projects on GitHub.

In Docker, dockerfile[3] is a text document that contains all the commands a user could call on the command line to assemble an image [2]. Users can build an automated build that executes several command-line instructions in succession by using `docker build`. Docker has provided multiple types of instructions in the dockerfile, involving `FROM`, `MAINTAINER`, `RUN`, `COPY`, `ADD`, `ENV`, *etc.* Specifically, The `FROM` instruction specifies the *Base image*, which can give a first indication of what it is that the projects use Docker for [1]. `MAINTAINER` instruction provides the name and email of an active maintainer. `ENV` instruction sets the environment variables. `COPY` instruction places files into the container. `ADD` instruction places files and unpacks the archive (*e.g.*, zip) in the container. `RUN` instruction executes any possible shell commands in a new layer on top of the current image and commits the results. Docker runs instructions in a dockerfile in order and treats lines that begin with "#" as a comment. A dockerfile must start with a `FROM` instruction. Other parts are then added on top of the base one [11]. It shows that each instruction represents one layer in Docker image. Figure 1 shows an example of dockerfile. It has 13 instructions, *i.e.*, 13 image layers, and 6 comments. Thus, we find that the size (scale) of a dockerfile (lines of commands) can reveal the size and complexity of the corresponding Docker image.

B. Dockerfile evolutionary trajectories

To meet the requirements of project development, *e.g.*, efficient containerization outcomes, the content of dockerfile may be modified at different stages by project maintainers, which we called *dockerfile evolution*. E.g., during the earlier stages, the owner *inutano* of project *inutano/wpgsa-docker*

[3]https://docs.docker.com/engine/reference/builder/

```
FROM example/rails

MAINTAINER example(example@gmail.com)

# Add here your preinstall lib(e.g. imagemagick, mysql lib, ssh config)
## Install imagemagick
RUN apt-get update
RUN apt-get -qq -y install libmagickwand-dev imagemagick

## Install for mysql gem
RUN apt-get install -qq -y mysql-server mysql-client libmysqlclient-dev

## Install for Webshots
RUN apt-get install libssl0.9.8 -y
RUN apt-get install ttf-unfonts-core -y

#(required) Install Rails App
ADD Gemfile /app/Gemfile
ADD Gemfile.lock /app/Gemfile.lock
RUN bundle install --without development test
ADD . /app

# Overwrite unicorn
ADD config/unicorn.rb /app/config/unicorn.rb
```

Fig. 1. An example of dockerfile.

added `USER` instruction and new python scripts to the initial dockerfile. But during the later stage, he just updated the plugin wPGSA's version, *e.g.*, "0.2.0"→"0.3.0". These changes can be intuitively reflected in the dockerfile evolution, which depends on the practices of the individual projects. It may vary because different projects have different codes and structures. But projects with similar goals and cultures may exhibit similar dockerfile evolutionary trajectories. To validate it, our prior study [3] conducted an exploratory study of the dockerfile evolutionary trajectories to quantify the Docker evolution and shed light on the project maintainers' different learning curves on docker configuration. We proposed a clustering-based approach that defined the dockerfile scale as the number of valid command lines of the dockerfile without blank lines and comments. After clustering all the projects' evolutionary vectors, we obtained six categories of dockerfile evolutionary trajectories:

- ***C-1*: *Increasing and holding***. This category comprises 21.8% of projects. In the *C-1*, developers added new instructions or new settings to the dockerfile in project's early periods. However, after reaching to a certain size, they just updated the basic environment variables or changed the location of instructions. Thus, the dockerfile scale stayed stable in project's later periods.
- ***C-2*: *Constantly growing***. This category comprises 31.6% of projects. In the *C-2*, developers kept adding services, support, or plugins to the dockerfile, causing its scale to continually increase. Compared with other categories, the evolution path of this category is easiest to understand.
- ***C-3*: *Holding and increasing***. This category comprises 19.2% of projects. In the *C-3*, developers just updated the basic environment variables in project's early periods, keeping the dockerfile scale stable for a while. Then, new instructions, plugins, or support were added, and the dockerfile scale increased.
- ***C-4*: *Increasing and decreasing***. This category comprises 10.2% of projects. In the *C-4*, developers kept adding new instructions in project's early periods, increasing the dockerfile scale. However, after reaching a large size, maintainers tried to reconstruct the dockerfile by

removing useless plugins/services or moving settings to additional script files, reducing the dockerfile scale in project's later periods.

- ***C-5*: *Holding and decreasing***. This category comprises 9.5% of projects. In the *C-5*, developers change little in project's early periods and the dockerfile scale is stable. In project's later periods, developers tried to change the dockerfile structure by moving some instructions to additional script files or using a more light-weight base image, so the dockerfile scale dropped.
- ***C-6*: *Gradually reducing***. This category comprises 7.7% of projects. In the *C-6*, developers kept removing useless instructions or changed the base image to reduce image layers, continually decreasing the dockerfile scale. However, the decrease slows in project's later periods.

III. Methodology

A. Data Set

1) Projects selection: To select our research projects, we extracted the project basic information from Docker Hub. Docker Hub provides good GitHub integration and developers can easily combine their GitHub and Docker Hub repositories in their workflow. It also provides some featured tools, *e.g.*, automated builds[4], which allow developers to build their images automatically from GitHub sources [6]. From the container list in Docker Store[5], we collected basic information for all the containers' that were there on or before July 2017. The builds data and dockerfile information on Docker Hub are available for collection, if the repositories are using the auto-builds tool. We identified projects that use the auto-builds tool by checking for the presence of the string "`is_automated`" through the Docker Hub API, *i.e.*, true means the project has auto-builds. After removing the projects that forked from other GitHub project or hosted on Bitbucket, we collected the basic information of 47,149 projects, including their names, creation times, and linked GitHub repository addresses.

2) Data collection and filtering: Based on the information of selected projects, our data collection involved mining two types of sources: (1) Docker Hub: Docker Hub builds, using Docker Hub API; and (2) GitHub: commits and Git logs, using GitHub V3 API. For each build, we collected its basic information, including {*repo, creation_time, finish_time, status*} (status>0 means it's a successful build without errors). We extracted the dockerfile change information from the GitHub commit logs, including {*repo, changed_date, commit_sha*}. Based on the regular URL expression[6], we downloaded the dockerfile content of each change. By using text parsing, we can extract detailed information of each dockerfile, *e.g.*, its base image and image size (lines of commands without blank lines and comments).

To better fulfill our quantitative study, we filtered our data according to the following constraints: (1) projects should be created before August 2016, *i.e.*, project age≥12 months; (2) projects should have enough successful builds, *i.e.*, successful builds≥10; (3) projects should change dockerfile enough times, *i.e.*, dockerfile versions≥10; and (4) dockerfiles should have complete information, *i.e.*, dockerfile must have a base image (start with a `FROM` instruction) and its size>0.

TABLE I
AGGREGATE STATISTICS OF THE 2,840 PROJECTS.

Statistic	Mean	St. Dev.	Min	Median	Max
Project age (month)	25.0	10.6	12	23	101
#Total builds	141.5	222.4	10	49	1,004
#Successful builds	120.9	200.3	10	38	1,000
#Dockefile versions	27.1	47.8	10	18	1,824
Avg. dockerfile size	26.1	22.1	2	20.5	400.4
Avg. dockerfile layers	14.2	9.0	2	11.8	89.7

3) Basic descriptive statistics: After the filtering, we obtained our final set of 2,840 projects. In total, our dataset contains 401,924 builds (85.5% of them are successful builds) and 76,925 versions of dockerfile. On average, each project has changed dockerfile 27.1 times (median: 18) and each version of dockerfile has 32.7 (median: 22) lines of instructions and 16.1 (median: 12) image layers. Table I presents aggregate descriptive statistics over the 2,840 projects.

B. Regression analysis

To explore the relationship between dockerfile architecture and containerization outcomes, we built two mixed-effects linear regression models, *Quality issues model* and *Build latency model*, (using function `lmer` and `lmer.test` in R) with the same random-effect term for the *base image*. As described in §II-A, we expected that the base image has an important effect on the building process of docker image, especially, the build latency. We captured the base image information by extracting its name in the specification, *i.e.*, a tuple of the form `namespace/name(:version)`. This way, we can capture base image variability in the response. All other variables were modeled as fixed effects. We divided the builds data of each project into different time-windows (stages) based on the dockerfile changes. The start and end time of each stage are the times when two adjacent dockerfile versions are generated. Our dependent variables are observed during those stages. The variables are as follows:

- ***avgQualityIssues***: the average number of quality issues of the dockerfile per stage, computed by the dockerfile Linter tool[7]. Dockerfile Linter is a popular tool based on the contributions by the GitHub community. This linter tool parses a given dockerfile and checks adherence to a set of rules representing the best practices on top of the resulting AST.
- ***avgBuildLatency***: the average time latency of successful builds per stage, as a proxy for build speed. Build latency is the time duration from build start to end, in minutes.

Our independent variables come from two confound areas: global level and local level.

[4]https://docs.docker.com/docker-hub/builds/
[5]https://store.docker.com
[6]https://raw.githubusercontent.com/[repo]/[commit_sha]/Dockerfile
[7]http://hadolint.lukasmartinelli.ch/

- ***totalBuilds***: total number of builds in the project's history, as a proxy for project size/activity.
- ***category***: different categories of the dockerfile evolutionary trajectory, as described in §II-B. The baseline is "*C-2*", because its evolutionary trajectory (constantly growing) is the most common one in all categories, and it is the simplest compared to other categories.
- ***timeFlag***: label of the time window, in months, computed since the earliest image build.
- ***officialImage***: Binary, True if the project use an official base image, *e.g.*, "Ubuntu".
- ***nLayers***: number of image layers of the dockerfile, as a proxy for dockerfile complexity.
- ***layerSize***: average number of lines of each layer, as a proxy for image layer complexity.
- ***insDiverstiy***: instruction entropy of the dockerfile, computed by the Shannon entropy [12], as a diversity of the instructions.
- ***additionalScript***: Binary, True if the project use additional script to run their instruction.
- ***getURLResource***: Binary, True if the project get the libraries/packages from outside URL source.
- ***nComments***: number of comments in the dockerfile.
- ***useLabel***: Binary, True if the project use the `Label` instruction to define environment variables.

In our models, where necessary we log-transformed dependent variables to stabilize their variance and reduce heteroscedasticity [13]. We removed the top 2% of the data to control outliers and improve model robustness. The variance inflation factors, which measure multicollinearity of the set of predictors in all our models, were safe, below 3. For each model variable, we report its coefficients, standard error, significance level, and sum of squares (obtained from ANOVA analyses). We consider coefficients important if they were statistically significant ($p<0.05$). Model fit was evaluated using a marginal (R^2_m) and a conditional (R^2_c) coefficient of determination for generalized mixed-effects models [14], [15]. R^2_m describes the proportion of variance explained by the fixed effects alone, and R^2_c describes the proportion of variance explained by the fixed and random effects together.

IV. Study Results

A. RQ1: Influence on dockerfile quality.

We built the *Quality issues model* to investigate the influence of dockerfile architecture on dockerfile quality. Table II shows the regression model result. The fraction of total deviance explained by the fixed-effects part of the model is R^2_m=0.12. The variability explained by the random effect is R^2_c=0.47.

As expected, the number of image layers (***nLayers***) has a significant positive effect on quality issues (25.3% of the variance explained), holding other variables constant. It indicates that *more image layers may bring more quality issues*. Interestingly, layer size (***layerSize***) has a significant negative effect (9.6% of the variance explained), meaning *quality issues may decrease if image layer size increases*, holding other variables constant. One explanation is that a large layer size may mean that instructions have complete and detailed parameters, so there are few quality issues, to be explored further in the future. The instruction diversity (***insDiversity***) has a strong, negative effect on quality issues (25.7% of the variance explained), holding other variables constant. Thus, *more diverse instructions tend to have less quality issues*.

TABLE II
QUALITY ISSUES MODEL. THE RESPONSE IS *log(avgQualityIssues)*. R^2_m=0.12, R^2_c=0.47.

	Coeffs (Error)	Sum Sq.
(Intercept)	-0.1079 (0.0325)***	
log(totalBuilds)	0.0034 (0.0049)	0.32
log(timeFlag)	-0.0290 (0.0047)***	24.55***
log(nLayers)	0.1854 (0.0057)***	715.36***
log(layerSize)	-0.1083 (0.0054)***	270.86***
insDiversity	-0.1662 (0.0050)***	725.13***
officialImage TRUE	-0.2241 (0.0150)***	149.98***
additionalScript TRUE	-0.1200 (0.0291)***	11.35***
getURLResource TRUE	0.1322 (0.0106)***	103.22***
useLabel TRUE	0.3806 (0.0114)***	741.61***
log(nComments+0.5)	0.0386 (0.0053)***	35.03***
category = C-1	-0.0745 (0.0124)***	49.15***
category = C-3	-0.0438 (0.0132)***	
category = C-4	0.0444 (0.0170)**	
category = C-5	-0.0059 (0.0184)	
category = C-6	-0.0798 (0.0195)***	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

As for the specific architecture factors of dockerfile, using an official base image (***officialImage***) has a significant negative effect on quality issues (5.3% of the variance explained), holding other variables constant. *The official images may bring fewer quality issues*, because it's obvious that official images provide more reliable and stable materials than personal images. Using additional script (***additionalScript***) has a small, significant negative effect on quality issues (0.4% of the variance explained), holding other variables constant. The additional scripts can reduce the burden on the original dockerfile, so *additional scripts may help reduce quality issues*. Getting external URL resource (***getURLResource***) has a significant positive effect on quality issues (3.7% of the variance explained), holding other variables constant. It may be explained that compared to using only local resources, additional instructions are required to extract and use the external URL resources, which indicates that *external URL resources may bring more quality issues*.

Defining environment labels (***useLabel***) has a significant positive effect on quality issues (26.2% of the variance explained), holding other variables constant. Although labels can give developers more detailed supplementary information, *e.g.*, the maintainer name and email address, the format requirement of a label is also prone to error, *e.g.*, the name of the maintainer is set but the address of the maintainer is missing. To some extent, *the use of labels tends to increase the risk of quality issues arising*. Number of comments (***nComments***) has a significant positive effect on quality issues (1.2% of the variance explained), holding other variables constant. It indicates that *quality issues may occur in the places where there are many comments*, suggesting that developers should

pay more attention to the dockerfile instructions with many comments, because those instructions may be complex and prone to quality issues.

For the different categories of dockerfile evolutionary trajectory (***category***), we find that there are some differences between different categories (1.7% of the variance explained). Compared to the baseline *C-2*, projects in *C-1*, *C-3*, and *C-6* have fewer quality issues, holding other variables constant. It indicates that *different dockerfile evolutionary trajectories tend to have different effects on dockerfile quality issues*. If we consider the coefficients, holding and increasing (*C-1*), or gradually reducing the dockerfile scale (*C-6*) seems to help more to reduce quality issues. We notice that projects in *C-4* tend to have more quality issues, we think it is probably because of too much pre-accumulation quality issues in the earlier periods, then the projects choose to reduce the dockerfile scale in their later periods.

Summary: The specific dockerfile architecture attributes, *i.e.*, number of image layers, size of each layer, diversity of instructions, *etc.*, have different and significant effects on the dockerfile quality. Compared to other categories except for the category of increasing and decreasing (*C-4*), the category of constantly growing (*C-2*) is associated with more quality issues. So we answer our RQ1:

Different dockerfile architecture attributes and evolutionary trajectories have different effects on the dockerfile quality.

B. RQ2: Influence on image build latency.

Next, we use the *Build latency model* to investigate the effect of dockerfile architecture factors on image build latency. Table III shows the result of regression model. The fraction of total deviance explained by the fixed-effects part of the model is R_m^2=0.17. A considerable amount of variability explained by the random effect (R_c^2=0.54). This is consistent with our description in §III-B, of the strong effect of the base image latency on the total build latency.

As expected, dockerfile layers (***nLayers***) has a significant positive effect on build latency (35.8% of the variance explained), holding other variables constant. It shows that *more image layers may bring longer build latency*. Image layer size (***layerSize***) also has a significant positive effect on build latency (31.7% of the variance explained), holding other variables constant. It indicates that *larger image layer size may cause longer build latency*, suggesting that developers should also pay attention to each image layer's size in addition to reducing the number of image layers. The instruction diversity (***insDiversity***) has a significant negative effect on build latency (13.0% of the variance explained), holding other variables constant. This means that *more diverse instructions tend to bring shorter build latency*. Because Docker provides a set of instructions with a variety of functions, developers can build a powerful containerization process with good outcomes, if they can reasonably combine those instructions.

For the specific architecture of dockerfile, using an official base image (***officialImage***) has a small, significant positive effect on build latency (0.4% of the variance explained), holding other variables constant. In indicates that *the official images may bring longer build latency*. One explanation is that official images have more burden content than personal images, which would cause longer build latency. Using additional script (***additionalScript***) has a significant positive effect on build latency (0.9% of the variance explained), holding other variables constant. It shows that *additional script would bring longer build latency*. Getting external URL resource (***getURLResource***) has a significant positive effect on build latency (0.5% of the variance explained), holding other variables constant. Compared to using only local resources, getting external URL resources would cause longer processing time as well as build latency, which indicates that *external URL resources may bring longer build latency*. Interestingly, defining environment labels (***useLabel***) has no significant effect on build latency, holding other variables constant. And the number of comments (***nComments***) only has a very small, positive effect on build latency (less than 0.1% of the variance explained), holding other variables constant. So we find that *labels and comments have almost no effect on build latency*.

For the different categories of dockerfile evolutionary trajectory (***category***), we find that there also exist some differences between different categories (1.7% of the variance explained). Compared to the baseline *C-2*, projects in *C-1* have shorter build latency, holding other variables constant. But projects in *C-4*, *C-5*, and *C-6* tend to have longer build latency, which may explain why these projects reduced their dockerfile scale during their dockerfile evolution, especially during their later periods of development.

Summary: Image build latency is affected by the specific dockerfile architecture attributes, especially, number of image layers, size of each layer, and the diversity of instructions. Compared to the category of constantly growing (*C-2*), the category of increasing and holding (*C-1*) is associated with shorter build latency, while other categories except for the category of holding and increasing (*C-3*) are related to longer build latency. So we answer our RQ2:

TABLE III
BUILD LATENCY MODEL. THE RESPONSE IS *log(avgBuildLatency)*. R_m^2=0.17, R_c^2=0.54.

	Coeffs (Error)	Sum Sq.
(Intercept)	-0.0195 (0.0325)	
log(totalBuilds)	0.1446 (0.0047)***	571.34***
log(timeFlag)	0.1144 (0.0045)***	384.08***
log(nLayers)	0.3239 (0.0054)***	2,155.54***
log(layerSize)	0.2857 (0.0050)***	1,909.54***
insDiversity	-0.1727 (0.0048)***	782.07***
officialImage TRUE	0.0859 (0.0144)***	21.19***
additionalScript TRUE	0.2645 (0.0271)***	56.77***
getURLResource TRUE	0.0737 (0.0101)***	31.91***
useLabel TRUE	0.0072 (0.0109)	0.26
log(nComments+0.5)	0.0146 (0.0050)**	5.02**
category = C-1	-0.0418 (0.0117)***	104.96***
category = C-3	-0.0101 (0.0125)	
category = C-4	0.0789 (0.0159)***	
category = C-5	0.1217 (0.0174)***	
category = C-6	0.1457 (0.0186)***	

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Different dockerfile architecture attributes and evolutionary trajectories have different effects on image build latency.

V. Discussion

A. Practice Implications

We find that the number of image layers, each layer's size, and other architecture attributes have different effects on the dockerfile quality (see §IV-A). Thus, a natural recommendation is to reduce the dockerfile image layers. In addition to that, developers should adjust their dockerfile architecture to make instructions more diverse and remove additional descriptions, *i.e.*, labels and comments. Using official image rather than personal images and using additional scripts would also help reduce quality issues in the dockerfile. But getting external URL resources would bring more quality issues, which should be concerned by developers.

We also find that other architecture attributes in addition to the image layers, layer size, and diversity of instructions, have different effects on the image build latency (see §IV-B). To reduce the build latency, we suggest that developers should reduce their image layers as well as each layer's size, and diversify the instructions. Also, developers should design their operations locally as much as possible without external resources or scripts.

Finally, our study shows that compared to the category of constantly growing (*C-2*), other categories have different effects on containerization outcomes (see §IV-A and §IV-B). In general, the category of increasing and holding (*C-1*) has relatively good influence with fewer dockerfile quality issues and shorter image build latency. An efficient, more targeted category of dockerfile evolution could be more beneficial for the containerization, but it is unclear how this can be achieved and this could be a direction for further study.

B. Threats to Validity

In this work, the six categories of dockerfile evolutionary trajectories are based on our prior study results. However, such summarizing of evolutionary categories may be incomplete. In the future, we will discuss more categories that may exist. In the regression modeling process, we controlled for the dockerfile instruction complexity with the number of layers, each layer's size, and instruction diversity, and we set the base image as a random effect. However, different dockerfile instructions could have different commands inside, which may cause some bias, although in our manual examination we did not find evidence for it.

We only considered Docker repositories that are on GitHub. Thus, our findings cannot be assured to generalize to projects hosted on other services, *e.g.*, Bitbucket and GitLab, although there is no inherent reason why they would be biased. Moreover, we only analyzed open source software. dockerfile evolution and its influence might be different in closed source environments. Finally, we conducted our study on the Docker-based containerization process. We cannot assume that our findings generalize to other containerization processes that are not using Docker.

VI. Conclusion and Future work

We conducted an empirical study to investigate the impact of dockerfile evolutionary trajectories on the dockerfile quality and image build latency in the Docker-based containerization. Based on the six categories of dockerfile evolutionary trajectories that teased out in our prior study, we regressed the dockerfile quality and image build latency against an extensive set of variables, fitted to the processed data, and two well-fitting models were obtained. Our findings indicate that more diverse instructions with fewer image layers and layer size can significantly help reduce dockerfile quality issues and shorten image build latency. We also distill some other practical advice for developers. In future work, we will continue this study by conducting a deeper analysis of the specific instruction evolution of dockerfile.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 61502512 and 61432020). We also thank Prof. Vladimir Filkov for his guidance in the prior study of dockerfile evolutionary trajectories.

References

[1] J. Cito, G. Schermann, J. E. Wittern, et al. An empirical analysis of the docker container ecosystem on github. In: Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, 2017. pp. 323-333.

[2] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014, 2014(239): p. 2.

[3] Z. Yang, W. Huaimin, and F. Vladimir. A clustering-based approach for mining dockerfile evolutionary trajectories. SCIENCE CHINA Information Sciences. https://doi.org/10.1007/s11432-017-9415-3.

[4] C. Anderson. Docker [software engineering]. IEEE Software, 2015, 32(3): pp. 102-c3.

[5] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In: Proceedings of the International Conference on Cloud Engineering. IEEE, 2014. pp. 610-614.

[6] C. Boettiger. An introduction to docker for reproducible research. ACM SIGOPS Operating Systems Review, 2015, 49(1): pp. 71-79.

[7] A. Manu, J. K. Patel, S. Akhtar, et al. Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. In: Proceedings of the International Conference on Circuit, Power and Computing Technologies. IEEE, 2016. pp. 1-14.

[8] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 2014, 1(3): pp. 81-84.

[9] W. Felter, A. Ferreira, R. Rajamony, et al. An updated performance comparison of virtual machines and linux containers. In: Proceedings of the International Symposium On Performance Analysis of Systems and Software. IEEE, 2015. pp. 171-172.

[10] A. Mouat. Using Docker: Developing and Deploying Software with Containers. O'Reilly Media, Inc., 2015.

[11] D. Jaramillo, D. V. Nguyen, and R. Smart. Leveraging microservices architecture by using docker technology. In: Proceedings of SoutheastCon. IEEE, 2016. pp. 1-5.

[12] Lin J. Divergence measures based on the Shannon entropy. IEEE Transactions on Information theory, 1991, 37(1): pp. 145-51.

[13] J. Cohen, P. Cohen, S. G. West, et al. Aiken. Applied multiple regression/correlation analysis for the behavioral sciences. Routledge, 2013.

[14] S. Nakagawa and H. Schielzeth. A general and simple method for obtaining r2 from generalized linear mixed-effects models. Methods in Ecology and Evolution, 2013, 4(2): pp. 133-142.

[15] P. C. Johnson. Extension of nakagawa & schielzeth's r2glmm to random slopes models. Methods in Ecology and Evolution, 2014, 5(9): pp. 944-946.