SCIENCE CHINA
Information Sciences

• Supplementary File •

A Clustering-based Approach for Mining Dockerfile Evolutionary Trajectories

Yang ZHANG[1,2*], Huaimin WANG[1,2] & Vladimir FILKOV[3]

[1]*Key Laboratory of Parallel and Distributed Computing;*
[2]*College of Computer,National University of Defense Technology, Changsha, 410073, China;*
[3]*University of California, Davis, CA, 95616, USA*

Appendix A Background

(1) Docker[1)] is an OSS project that implements operating system-level virtualization [9], and is built on many technologies from operating systems research: LXC [2] (Linux Containers), virtualization of the OS [3], *etc.* Docker has become the de-facto industry standard, and its usage is spreading rapidly [1]. As of December 2017, the most common Docker registry, *i.e.*, Docker Hub[2)], has hosted close to $1,700,000$ Docker projects. The Docker technology is primarily intended for developers to create and publish *containers* [4]. With containers, applications can share the same operating system and, whenever possible, libraries and binaries [5]. In fact, Docker launches its containers from a *Docker image*, which is a series of data layers on top of a base image [7]. 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 [8]. Since production environment replicas can be easily made in local computers, developers can test their changes in a matter of seconds. Also, changes to the containers can be made rapidly as only sections that need changing are updated.

The content of a 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 [6]. To meet the requirements of project development, the content of a dockerfile may be modified at different stages by project maintainers, which we refer to as the *dockerfile evolution.* During our preliminary data gathering of more than 57,000 projects from Docker Hub, nearly 70% of them had 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 structures. But, for projects with similar requirements and cultures, a similar dockerfile evolutionary trajectory may exist. Investigating dockerfile evolution can help understand the configuration practices of maintainers and benefit the project maintenance in the Docker environment.

(2) dockerfile[3)] is a text document that contains all the commands a user could call on the command line to assemble an image [6]. 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 [14]. Each instruction represents one layer in Docker image. Thus, the scale of a dockerfile can reveal the size and complexity of the corresponding Docker image.

As we mentioned above, the content of dockerfile may be modified at different stages by project maintainers to meet the project goals. *E.g.*, during the previous stage, the owner *inutano* of project *inutano/wpgsa-docker* 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 scale evolution, which depends on the practices of the individual projects. It may vary because different projects have different codes and structures. But projects

* Corresponding author (email: yangzhang15@nudt.edu.cn)
1) https://www.docker.com/
2) https://hub.docker.com/
3) https://docs.docker.com/engine/reference/builder/

with similar goals and cultures may exhibit similar dockerfile scale evolutionary trajectories. This motivated us to conduct an exploratory study of the dockerfile scale evolutionary trajectories to quantify the Docker evolution and shed light on the project maintainers' different learning curves on Docker configuration, which can benefit the project maintenance.

Appendix B Research Data

(1) Projects selection. From the container list in Docker Hub[4], we collected basic information for all the containers' that were there on or before July 2017. Docker Hub is a cloud-based registry service providing tools for developers to link to their source code repositories, to build image artifacts, and to deploy the images. These images are stored and maintained in Docker Hub repositories, of which there are close to 1,700,000 at the time of December 2017. Since Docker's inception in 2013, a large number of GitHub projects have used Docker [1]. 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[5], which allow developers to build their images automatically from GitHub sources [3]. Moreover, the builds data and dockerfile information on Docker Hub are available for download, if the repositories are public and use 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 selected projects, we extracted the dockerfile change information from the GitHub commit logs, including {*repo, changed_date, commit_sha*}. We downloaded the dockerfile content of each change by using the regular URL expression[6]. Then we extracted detailed data of each dockerfile, *e.g.*, base image and image scale (lines of commands without blank lines and comments) by using text parsing. To better fulfill our quantitative study, we filtered our data according to the following set of conditions:

- The project should be created before August 2016, *i.e.*, project age$\geqslant$12 months;
- The project should change dockerfile enough times, *i.e.*, dockerfile versions$\geqslant$10;
- The dockerfile should have complete information, *i.e.*, dockerfile have a base image and its scale>0.

After this filtering, we obtained our final set of 2,840 projects. In total, our dataset contains 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 commands and 16.1 (median: 12) image layers.

(3) Pre-clustering. Before performing our approach, we drew the heatmap of the dockerfile scale evolutionary trajectories of total 2,840 projects, as shown in Figure B1. By manually looking the color change trends, we can see that there are six clusters with significant differences. We marked them by using the dotted lines. So we set the number of clusters, k=6 in our K-means clustering process. After clustering all projects' evolutionary vectors, we marked each project as one of six categories. To better understand in what way dockerfile scale evolves in different clusters, we drew the dockerfile scale variation curves of some examples in each cluster (see Figure B1) and conducted case studies on randomly selected projects. After our manual analysis, we summarized below the specific paradigms of the six clusters.

4) https://store.docker.com
5) https://docs.docker.com/docker-hub/builds/
6) https://raw.githubusercontent.com/[repo]/[commit_sha]/dockerfile

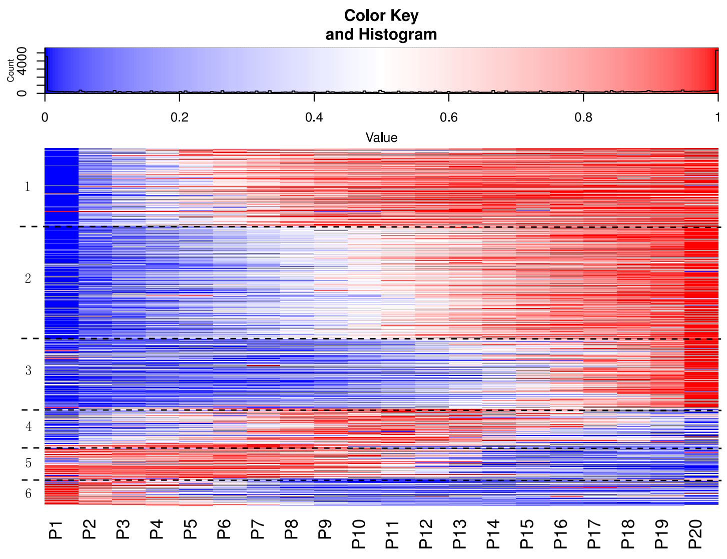

Figure B1 Heatmap of the dockerfile scale evolutionary trajectories.

Appendix C Dockerfile Evolutionary Paradigms

(1) C1: Increasing and holding. 21.8% of projects belong to this cluster. In this cluster, we find that developers added new instructions or new settings to the dockerfile at the early periods. But after reaching to a certain size, they just updated the basic environment variables or just changed the location of instruction. So the dockerfile size maintains stable (see Figure C1, *top*, *left*). For example, in project *inutano/wpgsa-docker*, its initial dockerfile size was 7 lines. Then the manager *inutano* added `USER` instruction and new python scripts so that the dockerfile size reached to 12 lines. After that, the manager just changed the image or container and user name, *e.g.*, "*FROM jupyter/datascience-notebook:4.0*"→"*FROM inutano/research-base:0.1.1*". Then he just updated the plugin *wPGSA*'s version, *e.g.*, "*0.2.0*"→"*0.3.0*".

(2) C2: Constantly growing. 31.6% of projects belong to this cluster. We find that in this cluster, developers kept adding services, support or plugin to the dockerfile which makes dockerfile size continue to increase. Compared with other paradigms, the evolution path of this paradigm is the simplest and the easiest to understand (see Figure C1, *top*, *right*). For example, in project *macintoshplus/php*, the initial size of dockerfile was 22 lines. Then developer *macintoshplus* added new `RUN` instruction, new plugin or new support, *e.g.*, *SQLite* and *Java*, which increases the size of dockerfile to 33 lines.

(3) C3: Holding and increasing. 19.2% of projects belong to this cluster. We find that in this cluster, developers just updated the basic environment variables at the early periods, which makes dockerfile size maintain stable. Then, new instructions, plugins or supports being added, and dockerfile size increased (see Figure C1, *middle*, *left*). For instance, in project *ckeyer/obc-env*, its initial dockerfile size was 6 lines. During the first 5 changes, developer *ckeyer* just added new parameters to `RUN` instruction or updated the file path, *e.g.*, "*cd /github.com...*"→"*cd $GOPATH/src/github.com...*". The dockerfile size still was 6 lines. Then *ckeyer* changed the base image "*FROM ckeyer/obc:base*"→"*FROM alpine:edge*" and added new `RUN` and `ENV` instructions. During the last 8 changes, 15 lines of code have been added to the dockerfile which makes dockerfile size up to 21 lines.

(4) C4: Increasing and decreasing. 10.2% of projects belong to this cluster. We find that in this cluster, developers kept adding new instructions at the early periods, so dockerfile size increases. But after reaching to a big size, maintainers tried to reconstruct the dockerfile by moving useless plugins/services, or moving some settings to additional script files, which makes dockerfile size drop at the later periods (see Figure C1, *middle*, *right*). For example, in project *combro2k/virtualmail*, dockerfile initial size was 24 lines. During the first 101 changes, developer *combro2k* kept adding new instructions, *e.g.*, `RUN` and `ADD` which makes the dockerfile size up to 163 lines. But during the later changes, *combro2k* tried to reconstruct the dockerfile by moving some instructions to other script files, *e.g.*, moving 12 `ENV` settings of software versions to the "*resource/bin/setup.sh*" file. Finally, dockerfile size dropped to 13 lines.

(5) C5: Holding and decreasing. 9.5% of projects belong to this cluster. We find that in this cluster, developers changed very little, dockerfile size maintains stable at the early periods. While at the latter periods, developers tried to change the dockerfile structure by moving some instructions to additional script files or using a more lightweight base image, so dockerfile size dropped (see Figure C1, *bottom*, *left*). For instance, in project *mobingidocker/ubuntu-apache2-ruby*, dockerfile had 22 lines of initial code. Then developer *David Siaw* added new script file, logging, git service and dependencies into the `RUN` instruction. After 6 changes, dockerfile size had 23 lines, only one line was added. At the 7th change, *David Siaw* moved 11 `RUN` instructions to the *provision.sh* file. Finally, dockerfile size dropped to 13 lines.

(6) C6: Gradually reducing. 7.7% of projects belong to this cluster. We find that in this cluster, developers kept removing useless instructions or changed the base image to reduce image layers, which makes dockerfile size continue to decrease. But at the later periods, the decreasing trend began to slow down (see Figure C1, *bottom*, *right*). For example, in project *keymetrics/pm2-docker-alpine*, the initial dockerfile size was 28 lines. Then developer *Unitech* changed the base image, "*FROM alpine:latest*"→"*FROM mhart/alpine-node:4*" and its related instructions, so dockerfile size dropped to 14 lines. After that, *Unitech* removed some instructions and merged similar instructions to reduce image layers and speed up the dockerfile build process. Finally, dockerfile size dropped to 7 lines.

Appendix D Differences Comparison

Those paradigms may indicate different project development stages and goals. To further explore the difference between projects in the six paradigms, we compared the project age and average dockerfile scale.

(1) Project age. We find that on average, the project age of Cluster-1 is 24.7 months (median: 23.0), of Cluster-2 the value is 24.8 (median: 23.0), of Cluster-3 the value is 24.8 (median: 22.0), of Cluster-4 the value is 25.7 (median: 24.0), of Cluster-5 the value is 26.2 (median: 24.0), and of Cluster-6 the value is 24.5 (median: 24.0). As shown in Figure C2, the later three clusters seems to have larger project ages than the previous three clusters. Overall, projects in *Cluster-1*, *Cluster-2*, and *Cluster-3* may be in the early stages of development, so they need to add content to the dockerfile to meet the requirements. As for projects in *Cluster-4* and *Cluster-5*, they may be in the latter stages of development. So in the late periods, dockerfile does not need to be added more new content, even some of its content needs to be removed or adjusted to make the dockerfile architecture more reasonable. It reflects the learning curves of project maintainers. Interestingly, we find it difficult to explain why the projects of *Cluster-6* have been reducing the dockerfile size. One explanation is that maintainers are not very satisfied with their original dockerfile architecture, probably because it contains a lot of unnecessary content, so they are constantly adjusting and modifying it later.

(2) Average dockerfile scale. We find that projects of Cluster-1 have an average of 28.6 lines of dockerfile scale (median: 22.1), of Cluster-2 the value is 28.3 lines (median: 22.8), of Cluster-3 the value is 24.0 lines (median: 19.1), of Cluster-4 the value is 26.3 lines (median: 18.9), of Cluster-5 the value is 21.8 lines (median: 17.0), and of Cluster-6 the value is 19.8

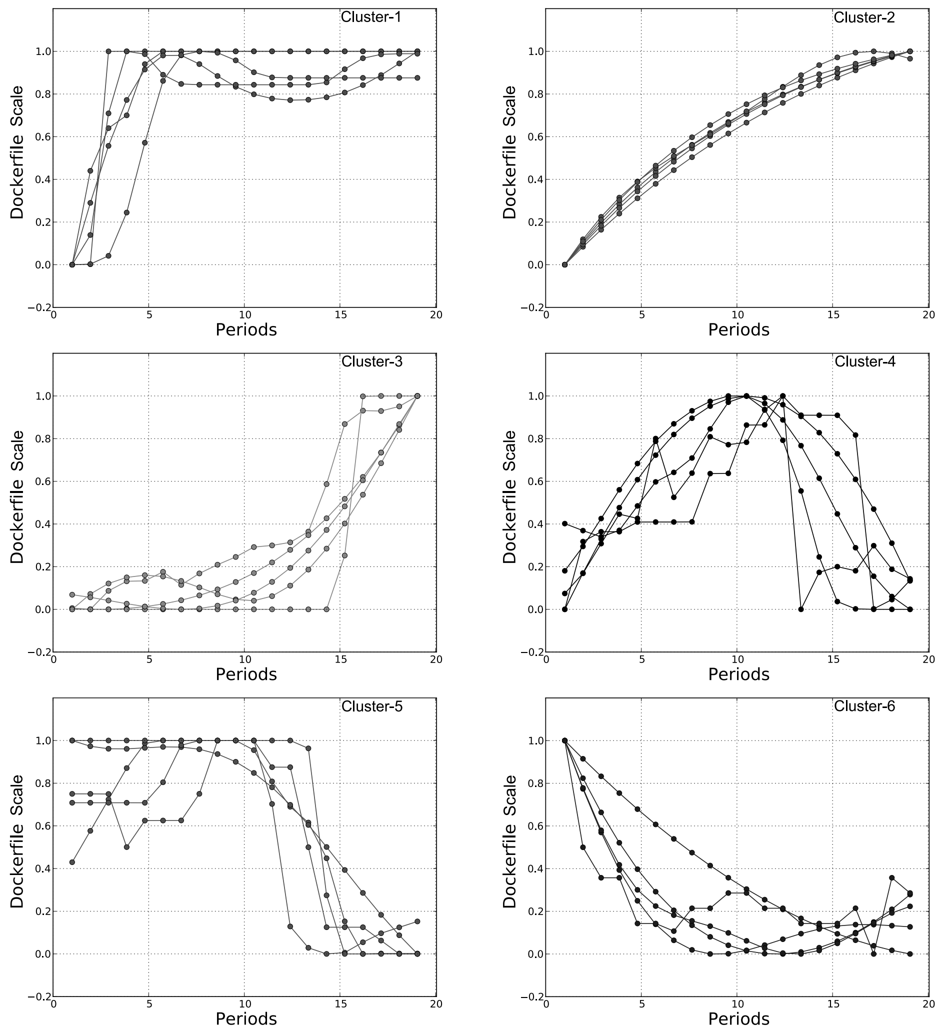

Figure C1 Examples of dockerfile scale (has been normalized) variation curves in different Clusters.

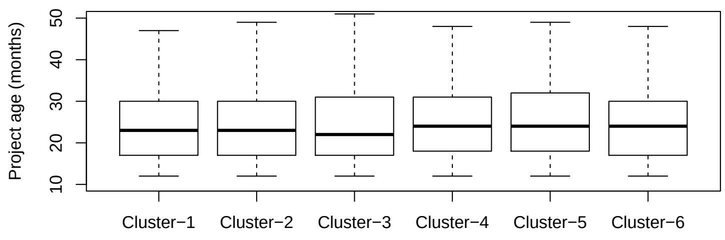

Figure C2 Difference of project ages between different Clusters.

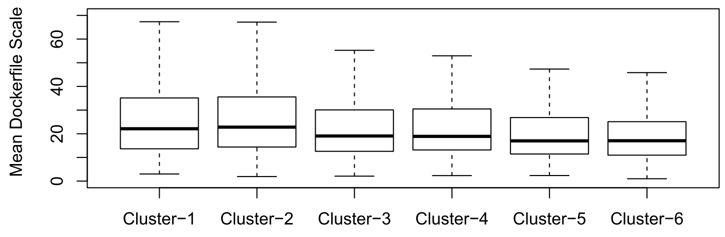

Figure C3 Difference of dockerfile scale between different Clusters.

lines (median: 17.1). As shown in Figure C3, the previous three clusters seems to have larger mean dockerfile scale than the later three clusters, which corresponds to that the later three clusters decrease their dockerfile scale in history. This indicates that different clusters of projects may have different development goals at different stages, *i.e.*, some projects kept adding components, plugins to make the project more powerful, while some other projects tend to make the project more lightweight and easy to maintain by removing unnecessary components or refactoring the entire framework.

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 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.
3 C. Boettiger. An introduction to docker for reproducible research. ACM SIGOPS Operating Systems Review, 2015, 49(1): pp. 71-79.
4 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.
5 D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 2014, 1(3): pp. 81-84.
6 D. Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014, 2014(239): p. 2.
7 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.
8 A. Mouat. Using Docker: Developing and Deploying Software with Containers. O'Reilly Media, Inc., 2015.
9 C. Anderson. Docker [software engineering]. IEEE Software, 2015, 32(3): pp. 102-c3.
10 N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. The American Statistician, 1992, 46(3): pp. 175-185.
11 G. C. Cawley and N. L. Talbot. Fast exact leave-one-out cross-validation of sparse least-squares support vector machines. Neural networks, 2004, 17(10): pp. 1467-1475.
12 M. J. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. Advances in optimization and numerical analysis. Springer, 1994, pp. 51-67.
13 J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm, Journal of the Royal Statistical Society. Series C (Applied Statistics), 1979, 28(1): pp. 100-108.
14 D. Jaramillo, D. V. Nguyen, and R. Smart. Leveraging microservices architecture by using docker technology. In: Proceedings of SoutheastCon. IEEE, 2016. pp. 1-5.