Structured information on state and evolution of dockerfiles on github

Schermann, Gerald ; Zumberi, Sali ; Cito, Jürgen

DOI: https://doi.org/10.1145/3196398.3196456

Structured Information on State and Evolution of Dockerfiles on GitHub

Gerald Schermann, Sali Zumberi, Jürgen Cito
Software Evolution and Architecture Lab
University of Zurich
Zurich, Switzerland
{lastname}@ifi.uzh.ch

ABSTRACT

Docker containers are standardized, self-contained units of applications, packaged with their dependencies and execution environment. The environment is defined in a Dockerfile that specifies the steps to reach a certain system state as *infrastructure code*, with the aim of enabling reproducible builds of the container. To lay the groundwork for research on infrastructure code, we collected structured information about the state and the evolution of Dockerfiles on GitHub and release it as a PostgreSQL database archive (over 100,000 unique Dockerfiles in over 15,000 GitHub projects). Our dataset enables answering a multitude of interesting research questions related to different kinds of software evolution behavior in the Docker ecosystem.

KEYWORDS

Docker, GitHub, Containers, Mining Software Repositories

ACM Reference format:
Gerald Schermann, Sali Zumberi, Jürgen Cito. 2018. Structured Information on State and Evolution of Dockerfiles on GitHub. In *Proceedings of MSR '18: 15th International Conference on Mining Software Repositories, Gothenburg, Sweden, May 28–29, 2018 (MSR '18),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Containers are standardized, self-contained units of applications, packaged with their dependencies and execution environment, which can be used for software development and to run the application on any system independent from the underlying operating system or hardware. The contents of a Docker container are defined in a *Dockerfile*, which specifies instructions to arrive at a certain infrastructure state [6], following the notion of Infrastructure-as-Code (IaC) [7]. Software repositories containing Dockerfiles enable the execution of program code in an isolated environment.

Given the fast rise in popularity, both in industry and academia, and its surrounding claim of enabling reproducibility [2], we developed a tool chain that transforms Dockerfiles and their evolution in Git repositories into a relational database model. Only recently, we conducted an initial exploratory study [3] on the Docker ecosystem on GitHub using this dataset. We had a first look on typical base images and programming languages used for Docker, investigated prevalent quality issues (e.g., how many Dockerfiles build successfully), and finally at the evolution of Dockerfiles. Our dataset has the potential to revisit these questions from a more recent perspective (i.e., allow for replication studies) and to dive even deeper, allowing for example, to investigate the co-evolution of Dockerfiles. Additionally, our data has potential to be combined with other software repositories to explore an even broader range of phenomena (e.g., GHTorrent [5], TravisTorrent [1]).

In the following, we present the key characteristics of our dataset, the process of data collection, details on the underlying data model, and finally, we list a few illustrative questions that might be answered when exploring our dataset.

2 DATASET AT A GLANCE

Our dataset comprises the entire population of more than 100,000 unique Dockerfiles originating from about 15,000 GitHub projects (state February 2018), enriched with information from the GitHub API to get additional metadata (e.g., owner type, owner name, used programming languages, project size, number of forks, issues, or the number of watchers). The dataset is available as a PostgreSQL database archive of around 26GB (uncompressed) in our online appendix [9]. The dataset can be easily explored using tools such as *pgadmin*.

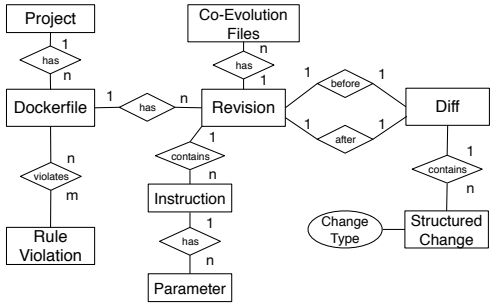


Figure 1: Schematic relational data model

We parse each Dockerfile and all its revisions into a relational data model (see Figure 1 for a schematic view of the underlying data model). A *Project* contains one to many *Dockerfiles*. A *Dockerfile* contains one to many *Instruction* entities (more than *5,900,000 instructions* over all Dockerfiles), each of them having one to many *Parameter* entities (≥ *11,000,000 parameters*). Each *Dockerfile* entity stores one to many *Revision* entities (≥ *350,000 revisions*), which reflect every commit on this Dockerfile. 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 (more than *2,500,000 changes*).

MSR '18, May 28–29, 2018, Gothenburg, Sweden

We categorized different types of differences (*Change Type*): ADD, MODIFY, DELETE, with subcategories for each instruction, which enables more fine-grained evolution analysis. Moreover, we collect information on co-evolution ($\geq$*100,000,000 co-evolution files*), i.e., capturing files (e.g., source code) that changed along with a Dockerfile or within a range of commits before or afterwards. Finally, we gather data on the adherence to best practices by reporting the results of a Dockerfile linter [8] (*rule violation* entities).

3 DATA COLLECTION METHOD

In the following, we provide insights on the process (see Figure 2 for an illustration) we established to collect our data.

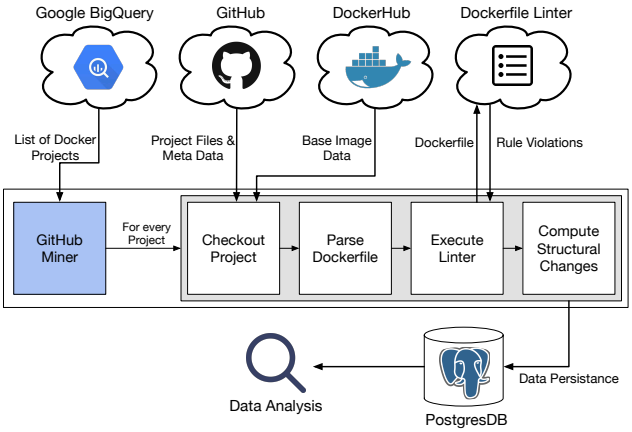


Figure 2: Method overview

- We started by retrieving a list of repositories that contain Dockerfiles from the public GitHub archive on Google's BigQuery[1] in January 2018.
- We removed repositories that were forks from other repositories to avoid biasing analysis of the dataset with duplicate entries and to keep the dataset compact. This would have been particularly problematic, as especially large, popular projects such as *Kubernetes* or *nginx* are forked frequently.
- The observation period for revisions and changes that we mined to analyze evolution behavior was from the first Dockerfile commit that appeared in the respective GitHub repository until January 2018.
- Our tool **DFA** (Docker File Analyzer) is a Java project responsible for checking out a specified GitHub project, parsing and storing all identified Dockerfiles in a relational database (Postgres), collecting information on the project's co-evolution, and computing structural changes with distinct change types between all Dockerfile revisions in a repository. The tool's sources and further details on how to run it are available in our online appendix [9].
- In addition to the data gathered from the Git repository, we also collected meta data from GitHub (e.g., number of stargazers) and DockerHub (e.g., official image available).
- We ran the linter [8] on the most recent version of a Dockerfile and store the resulting rule violations in our dataset.

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

Challenges and Limitations. The tooling for the data collection was built on the assumption that Docker projects on GitHub follow the standard naming convention for Dockerfiles (i.e., *Dockerfile* without attached file type). Moreover, retrieving the entire Docker ecosystem on GitHub is challenging as Docker has gained massive popularity. Therefore, the data collection process lasts multiple days even on multi-core server infrastructure and a thread-based execution model. As we mined the entire ecosystem we did not exclude "toy projects", i.e., projects that were created to play with and test the functionality of Docker. These projects might influence potential analysis conducted with our dataset. Finally, we only considered Docker repositories hosted on GitHub. As a consequence, findings based on the sole analyses of our dataset might not generalize to Docker projects hosted on other services such as Bitbucket or GitLab.

4 DATA MODEL

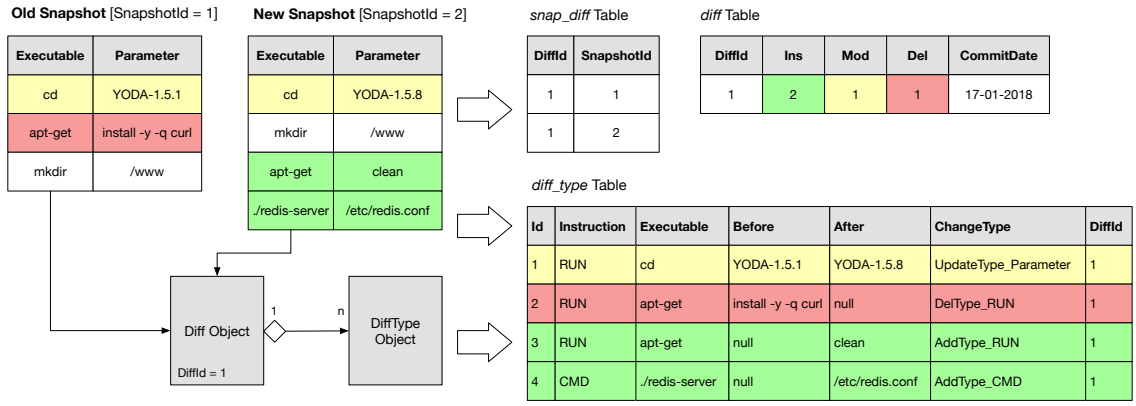
Figure 1 presents a simplified, schematic view of the underlying data model. A comprehensive entity relationship diagram of the database hosting the dataset is provided in our online appendix [9]. In Table 1 we present a more detailed description on selected tables (i.e., *Project*, *Dockerfile*, *Snapshot*, *Changed_Files*, and *From*) including type information and featuring an example data point.

In our dataset, we distinguish between Docker instructions used once within a Dockerfile (i.e., single instructions in Table 1 such as *From*) and instructions used multiple times (e.g., *Add*, *Expose*). Further, we distinguish between evolution data (i.e., subsequent revisions of Dockerfiles) and co-evolution data (i.e., which kind of changes happen to other files in the proximity to changes on Dockerfiles). While evolution within Dockerfiles is covered by the tables *Snapshot*, *Snap_Diff*, *Diff*, *Diff_Type*, co-evolution is mainly covered by *Changed_Files*.

Structurally Persisting Revisions. For each revision (i.e., a snapshot in our data model) of a Dockerfile, we compute the structural changes with distinct change types. Figure 3 showcases how we identify and structurally persist revisions for example instructions *RUN* and *CMD*. When comparing snapshots 1 and 2 in Figure 3 we identify a subtle change on the *change directory (cd)* command executed within a *RUN* instruction, i.e., the parameter "YODA-1.5.1" is modified to "YODA-1.5.8". In our table *Diff_Type* this is reflected by an *UpdateType_Parameter* change type entry (id 1). Before and after columns represent the change in detail. In addition, in snapshot 2, a *RUN* instruction is added executing *apt-get* with parameter "clean". In our dataset, this is reflected by an *AddType_RUN* change type entry (id 3). A *CMD* instruction is added executing *redis-server* with an attached configuration file (i.e., *AddType_CMD* change type entry with id 4). Finally, since the execution of *apt-get* is removed in snapshot 2, a *DelType_RUN* change type is added to the *Diff_Type* table (id 2).

Table *Diff* summarizes the number of changes (split into number of inserts, modifications, and deletes) occurring for each revision. For this concrete example, two inserts, one modification, and one deletion.

Quality Issues and Adherence to Best Practices. Best practices and coding guidelines not only exist for programming languages but also evolved for Infrastructure-as-Code (IaC) languages such as Puppet, Chef, or in our case Docker [4]. For this reason,



Executable	Parameter
cd	YODA-1.5.1
apt-get	install -y -q curl
mkdir	/www

Executable	Parameter
cd	YODA-1.5.8
mkdir	/www
apt-get	clean
./redis-server	/etc/redis.conf

DiffId	SnapshotId
1	1
1	2

DiffId	Ins	Mod	Del	CommitDate
1	2	1	1	17-01-2018

Id	Instruction	Executable	Before	After	ChangeType	DiffId
1	RUN	cd	YODA-1.5.1	YODA-1.5.8	UpdateType_Parameter	1
2	RUN	apt-get	install -y -q curl	null	DelType_RUN	1
3	RUN	apt-get	null	clean	AddType_RUN	1
4	CMD	./redis-server	null	/etc/redis.conf	AddType_CMD	1

Figure 3: Illustration of how two snapshots (revisions) are structurally persisted

during the data collection process, we executed an open source linter [8] for each Dockerfile to analyze the adherence to best practices and to reveal potential quality issues. The reported violations of best practices (i.e., ids of violated rules) were added to the *Violated_Rules* table in our dataset. Detailed descriptions and illustrating examples for rule violations can be found in the linter's documentation on GitHub.

5 FUTURE RESEARCH QUESTIONS

In the following, we list a few illustrative questions that can be answered with our dataset, potentially in combination with data from other software repositories. Meta data such as commit hashes allow researchers to fetch additional information for example from the GHTorrent [5] or TravisTorrent [1] datasets to explore an even broader range of phenomena.

- Characterizing Co-Evolution: What types of files change frequently together with Dockerfiles?
- What is the influence of failed CI tests of Dockerfile builds (e.g., what kind of changes happen after failed builds)?
- Do Dockerfiles change in similar rates to regular source code?
- Is there a connection between Dockerfile quality (e.g., adherence to best practices, build quality) and the quality of program source code?
- How is documentation (i.e., comments) in Dockerfiles used compared to regular source code?
- What is the relationship between Dockerfile documentation and quality?
- What kind of configuration files (e.g., shell scripts, ini files) have evolved into parts of Dockerfiles and what kind of configuration files are kept separated?
- What is the proportion of infrastructure code that is still executed in an external shell script as opposed to in Dockerfiles?
- Can we relate quality issues in Dockerfiles to questions asked on StackOverflow?
- Does Dockerfile quality correlate with build success/failure?
- Are projects that have updated Dockerfiles more often adopted by the community (i.e., comparing Dockerhub and GitHub stats with Dockerfile evolution)?

6 EXAMPLE QUERIES

In our online appendix [9], we provide examples as to how the dataset can be effectively queried. In the following we list three concrete examples. The first query returns the top used base images and their frequencies in descending order. The second query selects the most frequent used *RUN* instructions and parameters. Finally, the third query returns the average number of files that change (i.e., co-evolution) when a Dockerfile gets updated.

```
SELECT imagename, count(imagename) FROM df_from
  WHERE current = true
  GROUP BY imagename
  ORDER BY count DESC
```

```
SELECT executable, count(executable), run_params
  FROM df_run df NATURAL JOIN run_params rp
  WHERE df.current
  GROUP BY executable, run_params
  ORDER BY count(executable) DESC
```

```
SELECT avg(snap_id)
  FROM (
    SELECT snap_id, count(snap_id)
    FROM changed_files
    WHERE range_index = 0
    GROUP BY snap_id
    ORDER BY count(snap_id) DESC ) s
```

7 CONCLUSIONS

With our dataset we provide researchers the possibility to explore the Docker container ecosystem on GitHub in a fast and simple manner. Structured change types move the investigation of the evolution of Dockerfiles into a new light as we parse Dockerfiles specified in a declarative language and enrich them with change information. In addition, we foster analyses on co-evolution as we capture information of files that are changed in the proximity of a Dockerfile change. Finally, our dataset is enriched with information about the adherence to Dockerfile best practices. In our online appendix, we provide additional information to our dataset, including a link to download the database dump, how to build and execute our toolchain to start the data collection process, and some example queries demonstrating how to query the dataset.

Table 1: Description of selected tables and data fields (excerpt)

Column Name	Description	Type	Example
Table *Project*			
project_id	Auto generated id (primary key)	Integer	1
git_url	Project URL	String	https://github.com/zazujs/zazu
repo_id	GitHub repository ID	Integer	40297144
repo_path	Repository name	String	zazujs/zazu
created_at	Project creation date	Integer	1438846911
i_forks	Number of forking projects	Integer	1
i_owner_type	Owner type (User or Organization)	String	Organization
i_size	Project size in KB	Integer	511
i_stargazers	Number of stargazers on GitHub	Integer	3
Table *Dockerfile*			
dock_id	Auto generated id (primary key)	Integer	3
docker_path	Path to the Dockerfile	String	Dockerfile
first_docker_commit	Date when Dockerfile was added	Integer	1438846911
commits	Number of commits on the Dockerfile	Integer	2
project_id	Foreign key to Project table	Integer	1
Table *Snapshot*			
dock_id	Foreign key to Dockerfile table	Integer	3
snap_id	Auto generated id (primary key)	Integer	3
instructions	Number of instructions in this snapshot	Integer	10
from_date	Date of commit of this revision	Integer	1463433484
to_date	Date of commit of subsequent revision	Integer	1485480002
image_is_automated	Automated GitHub Build	Boolean	false
image_is_official	Official DockerHub image?	Boolean	false
current	Current snapshot?	Boolean	true
Table *Changed_Files*			
snap_id	Foreign Key to Snapshot table	Integer	3
changedfile_id	Auto generated id (primary key)	Integer	43
changetype	ADD / DELETE / MODIFY	String	MODIFY
commit_sha	Commit SHA	String	f87d9dad99b779936bf32e38662be4631840c675
insertions	Insertions made to the file	Integer	1
deletions	Deletions made in the file	Integer	1
range_index	0: Revision occurred when Dockerfile was changed; -1: Revision occurred one commit before Dockerfile; 1: Revision occurred one commit after Dockerfile;	Integer	-2
file_name	Name of changed file	String	instances
file_path	Path of changed file	String	server/init/apps/
file_type	Type of changed file	String	js
Table *From*			
snap_id	Foreign Key to Snapshot table	Integer	15
current	Current FROM instruction?	Boolean	true
full_name	Full image name	String	ubuntu:14.04
image_version	Image version	String	14.04

ACKNOWLEDGMENTS

The research leading to these results has received funding from the Swiss National Science Foundation (SNSF) under project name "Whiteboard" (SNSF Project no. 149450). We further like to thank the Swiss Group for Software Engineering (CHOOSE) for providing financial support to attend the conference.

REFERENCES

[1] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.

[2] Jürgen Cito, Vincenzo Ferme, and Harald C. Gall. 2016. *Using Docker Containers to Improve Reproducibility in Software and Web Engineering Research*. Springer International Publishing, Cham, 609–612. https://doi.org/10.1007/978-3-319-38791-8_58

[3] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the Docker container ecosystem on GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 323–333.

[4] Docker. 2018. Best practices for writing Dockerfiles. (2018). https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/, accessed 2018-02-01.

[5] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[6] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. *Testing Idempotence for Infrastructure as Code*. Springer Berlin Heidelberg, Berlin, Heidelberg, 368–388.

[7] Yujuan Jiang and Bram Adams. 2015. Co-evolution of Infrastructure and Source Code: An Empirical Study. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 45–55.

[8] Lukas Martinelli. 2018. Haskell Dockerfile Linter. (2018). https://github.com/lukasmartinelli/hadolint, accessed 2018-02-01.

[9] Gerald Schermann, Sali Zumberi, and Jürgen Cito. 2018. Paper – Online Appendix. (2018). https://github.com/sealuzh/msr18-docker-dataset, accessed 2018-03-12.