Noname manuscript No.
(will be inserted by the editor)

A Study of How Docker Compose is Used to Compose Multi-component Systems

Md Hasan Ibrahim · Mohammed Sayagh · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Many modern software applications are composed of several components (e.g., a web application is composed of a web server component and a database component). Each of these components can be instantiated as a container from a Docker image. Each Docker image corresponds to a software package (e.g, Apache or MySQL) along with various configuration details. Such containerization simplifies, speeds up, and enables the systematic deployment and maintenance of components at scale. As a natural progression of Docker, applications are now using "Docker Compose" to compose multi-component (aka. multi-container) applications by specifying the various components and their relations – in turn simplifying the deployment and maintenance of complex multi-component applications. This paper reports on a study of 4,103 open-source Github projects that use Docker Compose. Our primary goal is to better understand how it is used in the wild. We observe that over a quarter (26.8%) of the studied projects use Docker Compose for single-component applications. The Docker Compose file for an application is infrequently updated with 30% of such files never changed. We also observe that most of the composed applications leverage basic Docker Compose options instead of using advanced options (e.g., just 4.3% of the multi-component applications use a security related option). While Docker Compose has evolved over the years (it is currently at version 3), applications rarely adopt the new versions and 2.4% of the studied projects downgraded to an earlier version due to platform and option compatibility issues. Our study highlights that while applications

Md Hasan Ibrahim · Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University
Kingston, ON, Canada
E-mail: {ibrahim.mdhasan, ahmed}@cs.queensu.ca

Mohammed Sayagh
ÉTS - Québec Unviersity
Montréal, Canada
E-mail: {mohammed.sayagh}@etsmtl.ca

are using Docker Compose, they appear to be content with its basic options and earlier versions in many instances. Future studies are needed to better understand how to improve the uptake of the more advanced aspects of Docker Compose, if they are needed at all.

Keywords Docker · Docker Compose · Multi-component applications · Containerization

1 Introduction

Docker [8] enables the containerization of a software package along with associated configuration and setup details. Such containers can be easily and rapidly deployed while avoiding compatibility issues. In fact, a recent study reports that Docker can speed up the deployment of software components by 10-15 folds [23].

Docker is one of the most popular containerization technologies nowadays. Docker has captured 83% of the containerization market [24] – a market with an estimated revenue of $2.7 billion [1] by 2020. Datadog [5] (an emerging cloud service monitoring startup) reported in 2018 that around a quarter of their customers have already adopted Docker [6].

Much of today's applications are multi-component (i.e., multi-container) applications. For instance, a simple web application would require a web server and a database component. Docker Compose, a natural progression of Docker, enables practitioners to compose such complex applications [10]. Applications transcribe such compositions in a Docker Compose file, where components are specified by describing their Docker image and associated configuration as well as the relations between components. For example, one can specify a database component that uses a MySQL Docker image and stores data in a given directory. Furthermore, one can specify various actions to follow when a component fails.

Prior studies of Docker mostly focused on Docker images without considering the use of such images in multi-component applications. For example, Tak et al. [26], Shu et al. [25], and Zerouali et al. [27] studied the security of Docker images, while Cito et al. [4], and Zhang et al. [29] studied the evolution of Dockerfiles (i.e., the specification files for Docker images).

This paper reports on a study of 4,103 Github open-source projects that use Docker Compose. Our goal is to gain a solid empirical understanding of how applications use Docker Compose. We structure our study along with the following research questions:

RQ1. How do applications leverage Docker Compose?
Over a quarter (26.8%) of the studied projects use Docker Compose but just for single-component applications. Among the multi-component applications, 23% of them compose only local images, 40.2% of them compose both local and registry-hosted images, and the remaining 36.8% compose only registry-hosted images.

RQ2. What are the most used Docker Compose options?
22.6% of the available Docker Compose options are not used in any studied application. Advanced options like the ones for security and logging are rarely used. The security and logging options are used by 4.3% and 1.7% of the studied projects respectively.

RQ3. How do Docker Compose files evolve?
Docker Compose files change infrequently. Changes primarily occur to image related options (33.9% of changes), i.e., how to build an image, and data management options (24.9% of changes), i.e., how to store the data. Applications opt to pin the version of their composed images after facing compatibility issues between images and their applications (e.g., due to library version updates).

RQ4. How do applications use different versions of Docker Compose?
21.5% of the studied applications are still using version 1 of Docker Compose which will become deprecated in the very near future. A small number (8.5%) of the projects upgraded their Docker Compose version. 2.44% of the projects downgraded their Docker Compose version – many of the downgrades are due to applications wanting to use options that are no longer available in the new version of Docker Compose. An upgrade/downgrade requires changing a median of 10 lines of Docker Compose file code, which is double the typical number of changed lines in a regular change.

Our study highlights that while applications are using Docker Compose, they appear to be content with its basic options and earlier versions of Docker Compose in many instances. Future studies are needed to better understand how to improve the uptake of the more advanced aspects of Docker Compose, if they are needed at all. Our replication package is available online [1].

The rest of this paper is organized as follows: Section 2 provides background and related work about our study. Section 3 presents our data collection approach. Section 4 presents the observations of our study. Section 6 discusses the threats to validity of our observations. Finally, Section 7 concludes the paper.

2 Background and Related Work

One can compose a multi-component application from a set of components, each of which is defined as a Docker image and a set of options that configure the component's behaviour. In this section, we first discuss how to create an image and how to instantiate it, then how to connect components to compose a multi-component application.

[1] https://github.com/SAILResearch/replication-21-ibrahim-dockercompose

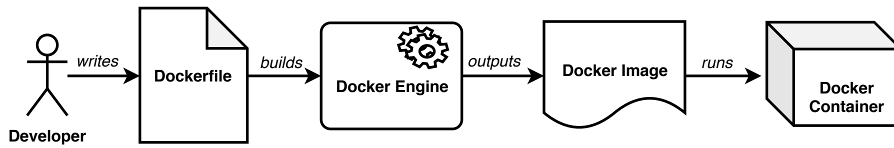


Fig. 1: Building a Docker Image and instantiating it.

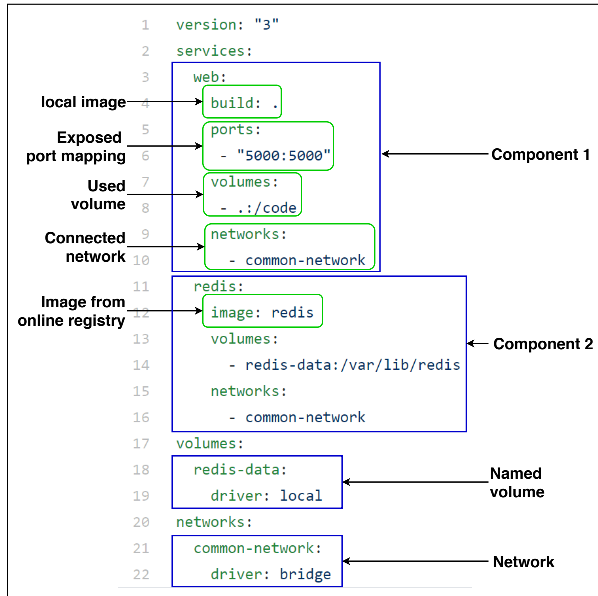


Fig. 2: Components and options in a Docker Compose file.

2.1 Docker Images and Containers

Figure 1 gives a high level overview of how Docker containers are created. A Docker container is an instance of a Docker image, which is created by the Docker Engine using a Dockerfile as input. A Dockerfile contains information about the software package, along with the required configurations for the package to function correctly. Such configurations include operating system versions, and other required software packages. One can instantiate local images or online registry images. Online registry images are ones that have been published on an online registry such as DockerHub [16], which allows other developers to reuse published images.

2.2 Docker Compose

One can compose a multi-component application using Docker Compose which composes a set of **components, each of which is an image and a set**

of options that specify how the component should behave. One can reuse the same image for different components; the reused images will result in different containers once instantiated.

Such composition of components is specified using a configuration file such as `docker-compose.yml`. Figure 2 shows an example of a Docker Compose file, which composes a multi-component application from the `web` and `redis` components. The `web` component is represented by a local image. The `redis` component is created from the "redis" image, which is hosted on online registry (e.g., DockerHub [16]). Additionally, both components (in Figure 2) have additional options that configure their environment. The web component exposes the port 5000, stores its data in an external volume (stored in the "/code" path in the host machine), and uses the `common-network` specification to access other components of the same multi-component application.

2.3 Related Work on Docker

Prior studies focus on the quality and the evolution of Docker images. They examine such images independent of their global context and do not consider how such images are composed together to create more complex applications. For example, Tak et al. [26] reported that 92% of DockerHub images contain security vulnerabilities and compliance issues, Shu et al. [25] developed a vulnerability analysis tool named DIVA (Docker Image Vulnerability Analysis) which can discover, download, and analyze images from DockerHub and identify security vulnerabilities, Zerouali et al. [27] analyzed the outdated and most updated DockerHub images and reported that even the most up-to-date images can contain severe security vulnerabilities, Cito et al. [4] reported that Dockerfiles (i.e., the specification files for Docker images) change a median of 3.11 times per year, and Zhang et al. [28] identified that Dockerfiles follow six different evolutionary patterns. Henkel et al. [20] leverage existing Dockerfiles to mine rules, which are used to better parse Dockerfiles and provide better semantic support for Dockerfile developers. Horton et al. [21] proposed an approach that builds Docker specifications for gist scripts, as many of these scripts do not get executed due to missing dependencies.

3 Data Collection

We wish to study a large number of non-trivial applications which are composed using Docker Compose. Hence, we follow the following steps:

- We first queried the Github database in Google Big Query [18] to retrieve projects that contain at least one `docker-compose.yml` file. We obtained an initial list of 21,269 projects.
- We then removed deleted projects (since they are no longer available) and forked projects (to avoid any bias due to the duplicated maintenance activities of such projects). We obtained the list

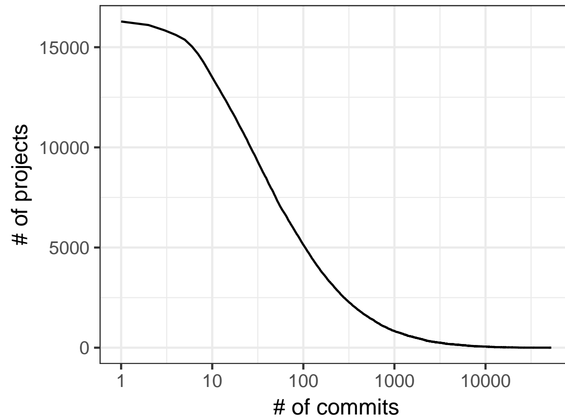


Fig. 3: Number of commits in the collected projects.

of deleted and forked project from the GHTorrent database [19] (updated until April, 2019). We ended up with a list of 16,283 projects.

- To avoid studying trivial (e.g., toy or personal) projects, we filtered out projects with less than 100 commits, so we ended up with 5,139 projects as shown in Figure 3.
- We then cloned these projects from Github. However, we could only obtain 4,917 projects since the remaining projects are no longer available on Github (e.g., made private or deleted after April 2019).
- We limited our dataset to projects that contain a single `docker-compose.yml` file since it is not feasible to know which Docker Compose file is used. We ended up with 4,327 projects.
- We also removed any projects with an empty Docker Compose file, which resulted in a dataset with 4,136 projects
- While parsing the Docker Compose files (using our own parser available in our replication package) we could not parse 33 files due to the incorrect syntax of these files. Hence, we removed these projects which resulted in a **final dataset of 4,103 projects**.

4 Results

The goal of this paper is to better understand how developers compose their applications using Docker images. To achieve this goal, we address the following four research questions:

RQ1. How do applications leverage Docker Compose?
RQ2. What are the most used Docker Compose options?

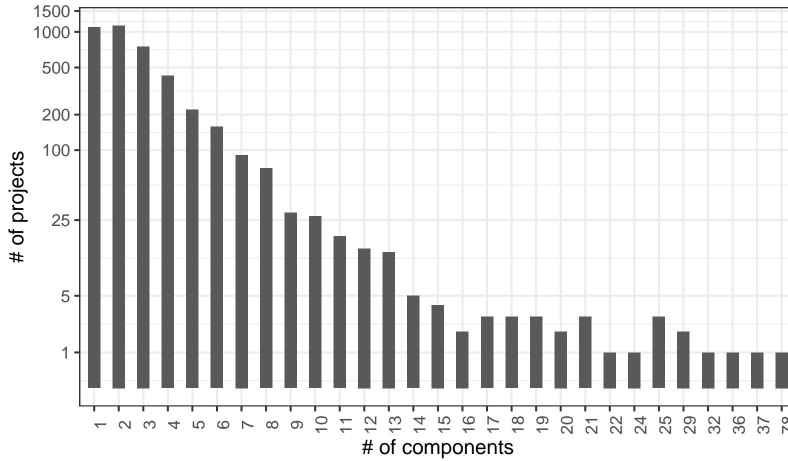


Fig. 4: The number of projects for a given number of components.

RQ3. How do Docker Compose files evolve?
RQ4. How do applications use different versions of Docker Compose?

RQ1. How do applications leverage Docker Compose?

Motivation: The goal of this research question is to understand how applications are composed using Docker Compose. Such an empirical understanding of how Docker Compose, a relatively new approach, is used for automating the composition of multi-component applications would help researchers and practitioners understand common practices and help identify open research and practical challenges.

Approach: To understand how applications are composed using Docker Compose, we parse the Docker Compose file of each of the selected projects to find out the images that they use. In this regard, we examine the `build` and `image` options for each defined component in the parsed Docker Compose files (as shown in Figure 2). For instance, we identify local images and online registry images that are hosted on online registries such as DockerHub.

We also investigate the commonly used online registries. To identify images that are hosted on online registries, we identified images that are specified using the following four patterns [3]:

- index.docker.io/{repository}/{image_name}
- docker.io/{repository}/{image_name}
- {repository}/{image_name}
- {image_name}

Furthermore, we also identify official DockerHub-hosted images by searching for images that are specified using the following patterns [3]:

- library/{image_name}
- {image_name}

Results: The studied applications are composed from a median of two components and as much as 78 components (as shown in Figure 4). Based on our manual analysis of all the 110 projects that have at least 10 components, we observe that just 51 projects are multiple component software systems. 14 among these 51 projects claim to implement a micro-service architecture. We observe 5 repositories that are toolkits for a certain field. For example, the repository "dceoy/docker-bio" represents a set of tools for the biometric field. The "instedd/nuntium" repository provides different components to send messages, each of these components is dedicated to one type of message such as SMS, emails and twitter. 4 repositories provide development environments for other projects. For example, the "Codewars/codewars-runner-cli" project provides a set of components, each of which is "to execute small sets of code within various languages, using various testing frameworks". Among the components of our 110 manually studied projects with more than 10 components, we observe that 13 repositories have components that provide environments for testing. For example, the "dustinleblanc/veccs" repository has components for the needed infrastructure software such php and nginx, and components to test these infrastructures such as testphp and testnginx. That repository also uses the component "wernight/phantomjs", which is a testing framework.

15.6% of the studied multi-component applications specify components which reuse the same Docker image. We observe that some images are reused by as little as one component and as much as 51 components in the same multi-component application. With many applications having many components that reuse the same Docker image; we observe that definitions of such components have a large amount of duplication (as the definitions of these components are often quite similar modulo some minor differences).

29 out of the 110 projects use the same Docker image for different components. While these projects use the same images for different components, each component has a different configuration such as storing a dataset in different data volumes, or having different values for environment variables. We also observe cases where the unique difference between the components is the version of the used image. For example, the "rstiller/inspector-metrics" project has 15 components that are identical for the compilation and tests, the unique difference is the version of the NodeJS that is used by each component. The last example is the "vlam321/Inf191BloomFilter" project, which implements the sharding pattern (i.e., having different database environments, each of which stores a subset of the data). The project has the same component repeated, where each component corresponds to a shard.

Over a quarter (26.8%) of the studied applications use Docker Compose for single-component applications, even though the primary motivation behind Docker Compose is to compose multi-component applications [10]. (65%) of these single component applications specify the

mapping of the virtual storage in a component to an actual physical location on the host machine. 35% of the single-component applications do not specify any virtual storage. Based on a manual analysis of a representative random sample (with a 95% confidence level and 5% confidence interval) of 284 single component projects, we identified three reasons for using docker-compose with just one component. (1) A project can provide docker-compose for other contributors as a means to set up an environment and quickly execute a project, rather than using docker-compose to connect different components. (2) One can use docker-compose for quickly setting up a testing environment. (3) We also observe that some projects use docker-compose to codify all the command line parameters that one would have to consider when building an image using just a Dockerfile. For example, one has to execute a simple command like "docker-compose up -d ." instead of "docker run -d -p 10086:10086 -v /var/run/-docker.sock:/var/run/docker.sock tobegit3hub/seagull". All those "docker run" parameters are codified within the docker-compose file. Note that we were able to identify the reasons for using docker-compose with one component for 58 projects, by leveraging these projects' readme file and the commit messages that changed the docker-compose file. For instance, commit messages are often not explicit about the reason behind using docker-compose. For example, the "smartystreets/smartystreets-php-sdk" project mentions as a commit message "Added docker-compose.yml" when they started using docker-compose.

Multi-component applications leverage components that are built from local Docker images as well as registry-hosted (mostly DockerHub) images. Among the multi-component applications, 23% of them compose only local images, 40.2% of them compose both local and registry-hosted images, and the remaining 36.8% compose only registry-hosted images. Finally, 1.7% (134) of the multi-component applications are built dynamically. Such applications receive their image name as a command line parameter.

DockerHub is the most used online registry for remote images. 95.2% of the identified registry-hosted images are hosted on DockerHub, while the remaining 4.8% of the registry-hosted images are hosted on registries such as Quay (124 images) or Google Container Registry (21 images). 53.5% of the identified DockerHub images are official images, while the remaining (46.5%) images are either community or private images on DockerHub. We had expected more prominent use of official images. Future research is needed to investigate the rationale for applications not using official images as prominently in an effort to improve such official images.

The most popular DockerHub images are related to infrastructure images. Most of the popular images are related to infrastructure components such as databases (e.g., Postgres, Redis, Mongo, MySQL) and web servers (e.g., Nginx), as shown in Figure 5. Several popular combinations of images exist that are co-used in a considerable number of applications. For example, the most popular combination of images is Postgres and Redis, which accounts for 5.9% (99 applications) of the multi-component applications that use more than one registry-hosted image (1,686 applications) as shown in Figure 6.

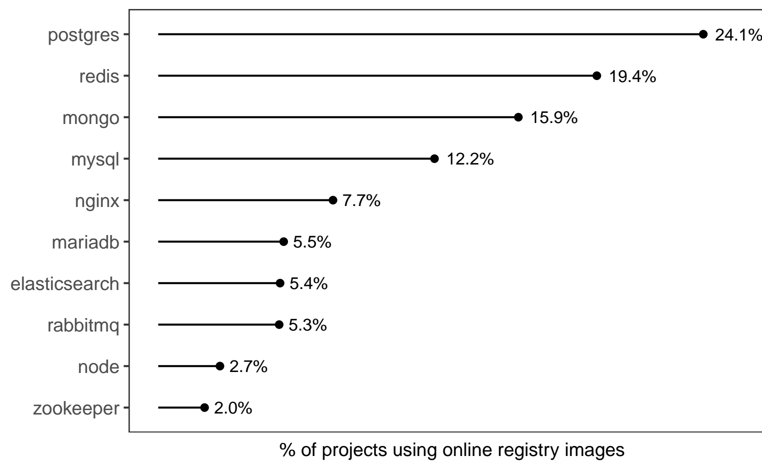


Fig. 5: The top 10 most used images by the percentage of projects.

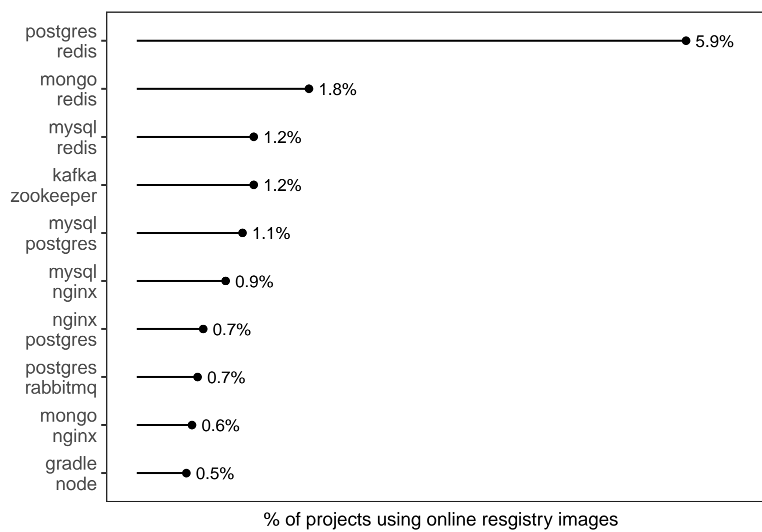


Fig. 6: Top 10 most popular image combinations used by the percentage of the multi-component applications that use more than one image from an online registry.

Summary of RQ1

26.8% of the studied applications use Docker Compose to create single-component applications. Multi-component applications are being composed from local and registry hosted components with most of the registry hosted components from DockerHub.

Table 1: Categories of options in Docker Compose across all versions.

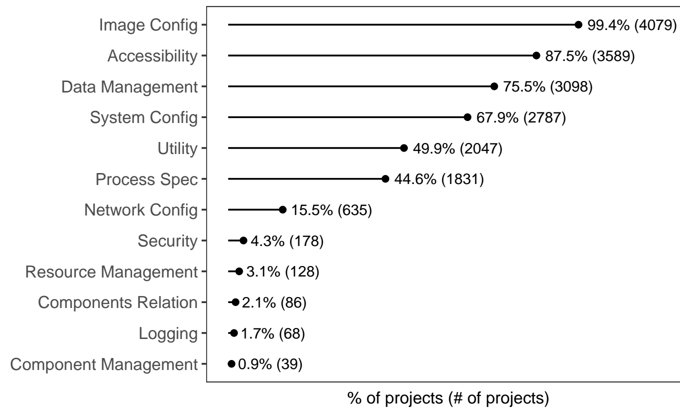
Category	**# of Options**	**Description**	**Example**
Accessibility	12	These options define different accessibility methods for components and devices.	external_links, ports
Component Management	2	These options manage components by distributing and replicating them.	placement, replicas
Components Relation	1	This option helps applications reuse configurations across components.	extends
Data Management	9	These options manage the volumes for components.	volume_driver, volume_from
Image Config	7	These options define how a component will be built.	build, image
Logging	3	These options help applications log the activities of components.	log_driver, log_opt
Network Config	17	These options are related to configuring the network for components.	network_mode, dns
Process Spec	14	These options specify how different tasks will be carried out.	restart_policy, priority
Resource Management	25	These options are related to managing computing resources such as CPU and RAM.	cpu_quota, mem_limit
Security	11	These options are related to the security policy of a component.	isolation, cap_drop
System Config	5	These options set the environment of a component.	environment, platform
Utility	9	These options provide different utility functions for running components.	healthcheck, tty

RQ2. What are the most used Docker Compose options?

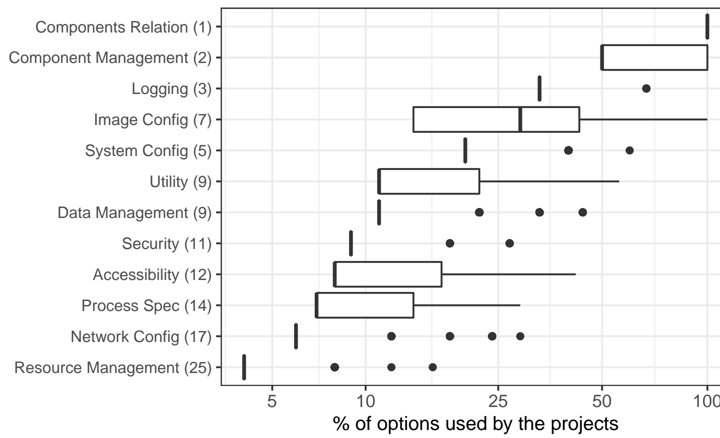
Motivation: The goal of this research question is to identify the commonly used Docker Compose options and the ones that are not used. Follow up research is needed to better understand the rationale for such usage patterns and the impact of such patterns on the evolution of Docker Compose itself.

Approach: To identify the commonly used Docker Compose options, we first collect the keywords for the existing Docker Compose options that are available across the three major versions of Docker Compose [12, 14, 15]. Then, we parse our dataset of Docker Compose files to determine the usage frequency for these options. To better understand how these options are used, we also use association rule mining to identify the co-occurrence of options [2]. Finally, we classified Docker Compose options based on their goals to identify the common reasons for using different options.

Results: Applications mostly use basic Docker Compose options, while advanced options like the ones for security and logging are rarely used. We categorized the 115 existing options into 12 categories as shown in Table 1. The most used Docker Compose options are related to building a component (`Image Config` options), accessing it (`Accessibil-`



(a) Percentage of projects using different categories of Docker Compose options.



(b) Distribution of proportion of used options across each option category. The number inside parentheses in the y-axis indicates the number of options in that category.

Fig. 7: Usage of different option categories in Docker Compose files.

`ity` options), and managing its data (`Data Management` options). On the other hand, sensitive options, such as security or logging options are rarely used. The security and logging options are used by only 4.3% and 1.7% of the studied projects respectively, as shown in Figure 7a. Even when applications use a category of options, they use a small portion of its options as shown in Figure 7b. For example, applications use a median of 9% of the `Security` options and 11% of the `Data Management` options.

22.6% of the total options that were introduced in major versions of the Docker Compose were never used in our studied projects. 22.4%, 24%, and 9% of the options of the first, second, and third docker-

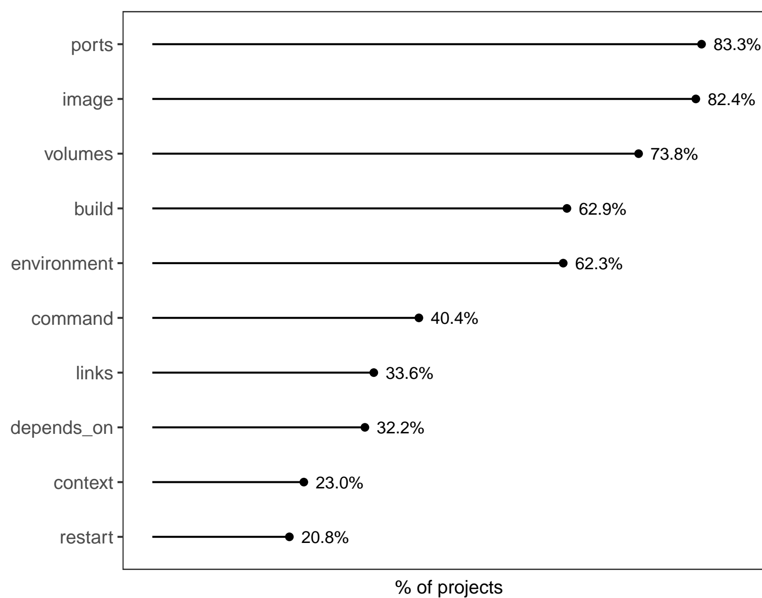


Fig. 8: The top 10 most used options across the studied projects.

compose version are not used, respectively. The unused options are listed in Table 2: 12 options are for resource management, 5 options are security related, 4 options are for network configuration, and 3 options are for process specification. For example, no studied project uses the `rollback_config` option, although it is important to specify the rollback action to be taken when a component fails.

`ports` is the most used option (used by 83.3% of the studied projects). `ports` exposes an internal port of an image so other images and external resources can access it. The options `image` and `build` are within the top 4 most used options (as shown in Figure 8) since these options are the core Docker Compose options – they are used to create and compose images.

73.8% of the applications manage their resources outside their components, as shown in Figure 8. The `volumes` option helps export data outside a component. Since components are stateless and the data is destroyed once a component is destroyed, one must use the `volumes` option to make the application data accessible on the host machine, and to enable the sharing of the data with other components as well.

Table 2: Options in Docker Compose that are never used. Note that the classification of these options is based on our own analysis.

Category	**Options**	**Supported versions**	**Description**
Data Management	storage_opt	2	Sets the option for using a storage driver in a component.
	volume_driver	1, 2	Specifies the default volume driver for all used volumes in a component.
Network Config	dns_opt	2	Lists the custom DNS options for a component.
	link_local_ips	2	Lists the link-local IPs for a network used by a component.
	mac_address	2	Sets the mac address that will be used by a component.
Process Spec	oom_kill_disable	2	Boolean value to enable or disable OOM (Out Of Memory) killer for a component.
	pids_limit	2	Sets the PID limits for a component.
	priority	2	Specifies the order in which components will be connected to the networks in case of multiple network connections.
	rollback_config	3	Specifies the procedure to rollback in case of an update failure.
Resource Management	blkio_config	2	Sets the limits for block IO in a component.
	cpu_count	2	Sets the allocated number of CPUs for a component to run (option is only available for Windows systems).
	cpu_period	2	Sets the period for the CPU CFS (Completely Fair Scheduler) used by a component.
	cpu_rt_period	2	Sets the CPU real-time period for a component.
	cpu_rt_runtime	2	Sets the real-time runtime for a component.
	device_read_bps	2	Sets the limit in bytes per second for read operations on a given device.
	device_read_iops	2	Sets the limit in operations per second for read operations on a given device.
	device_write_bps	2	Sets the limit in bytes per second for write operations on a given device.
	device_write_iops	2	Sets a limit in operations per second for write operations on a given device.
	mem_swapiness	2	Specifies the total memory limit including the swap memory of a component.
	weight	2	Sets the proportion of allocated bandwidth of a component with respect to other components.
	weight_device	2	Sets the relative bandwidth allocation of a device by a component.
Security	isolation	2, 3	Specifies the isolation technology of a component.
	userns_mode	2, 3	Used to disable user namespace in a component.
	device_cgroup_rules	2	Used to add rules in the devices that allow cgroup.
	group_add	2	Adds groups of which the component user should be a member.
	credential_spec	3	Sets the credentials for the managed components in a Windows environment.

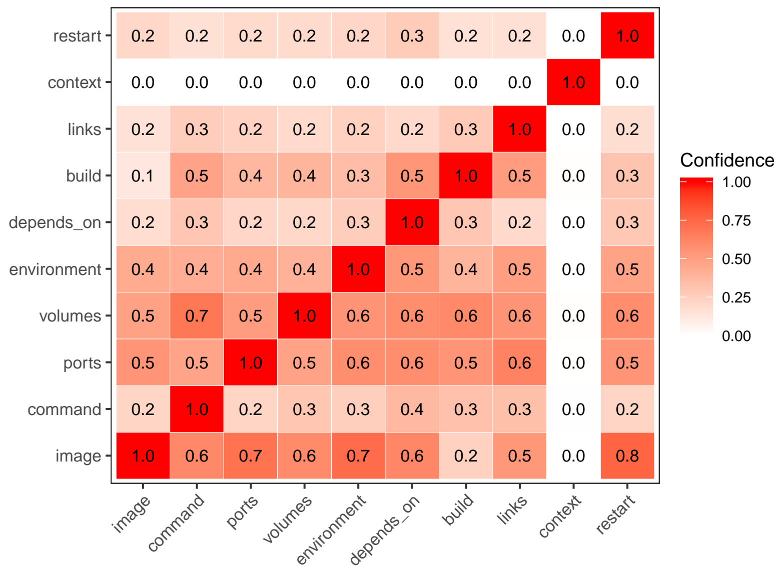


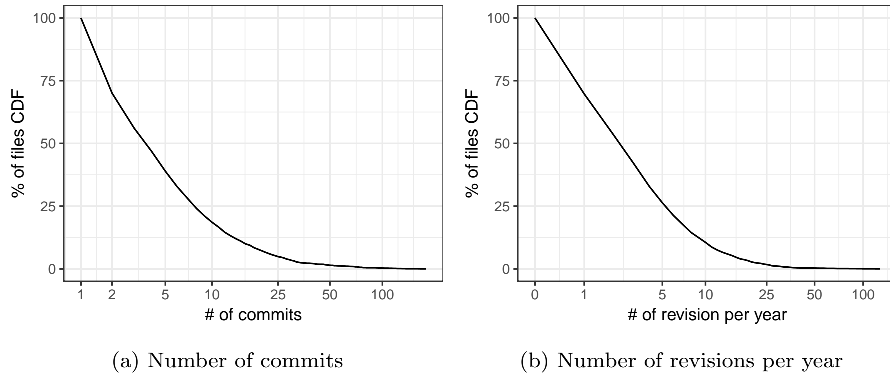
Fig. 9: Confidence metric values of co-occurrence of the top 10 most used options appearing in the same component.

We do not observe a clear pattern of how options are used as few options co-occur together. Although some options are used to configure other options, we observe that such options do not co-occur (as shown in Figure 9). For example, in just 20% `image` options configure their `restart` option – even though one should ideally configure how to restart a component in case of a failure. While 50% of the components that use the `image` option use the `ports` option. That said either components are not exposing their ports or their Docker images have already exposed a port through their Dockerfile – highlighting that a few options are configurable at the Dockerfile level or the Docker Compose level leading to inconsistencies and making it impossible to simply examine the Docker Compose file to gain a complete view of how a multi-component application is connected.

Summary of RQ2

22.6% of the available Docker Compose options are not used in any studied application. Advanced options like the ones for security and logging are rarely used.

RQ3. How do Docker Compose files evolve?



(a) Number of commits

(b) Number of revisions per year

Fig. 10: The frequency of change for Docker Compose files. The y-axes represents the cumulative percentage of Docker Compose files. For example, in Figure 10a, 18.4% of the Docker Compose files have at least 10 commits.

Motivation: The goal of this research question is to examine the type of changes that occur on Docker Compose files in order to better understand the stability of such files and the complexity of their evolution.

Approach: To understand the evolution of Docker Compose files, we study the change history of each of the studied Docker Compose files. We measure the amount of changes that these files exhibited, as well as changes to their usage of the different Docker Compose options. These options are obtained using an approach that is similar to the one that was used in the previous research question, but on each revision of the studied Docker Compose files. Note that we do not consider merged commits to avoid double counting.

We extended our investigation of the reasons behind such changes by manually investigating the commit messages and the changes for the top two most frequently changed options. In this regard, we randomly selected a statistical representative weighted sample of 370 commits that either added, removed, or modified those two options (out of 10,143 commits with a 95% confidence level and a 5% confidence interval). Our random sample consists of 197 changes to the `volume` feature and 173 changes to the `image` feature.

Results: Docker Compose files are revised infrequently. The median number of commits in a Docker Compose file is just three commits including the initial commit. As shown in Figure 10a, around 30% of the projects never changed their Docker Compose files after the initial commit. Moreover, Docker Compose files were updated just once during the whole life-cycle of 1% of the projects. Furthermore, the median number of revisions of a Docker Compose file is only two revisions per year as shown in Figure 10b, which is similar to the finding of Cito et al. [4] on the evolution of Dockerfiles. The number of revisions of a Docker Compose file is moderately positively correlated with the number of lines of code in that Docker Compose file (Spearman's rank

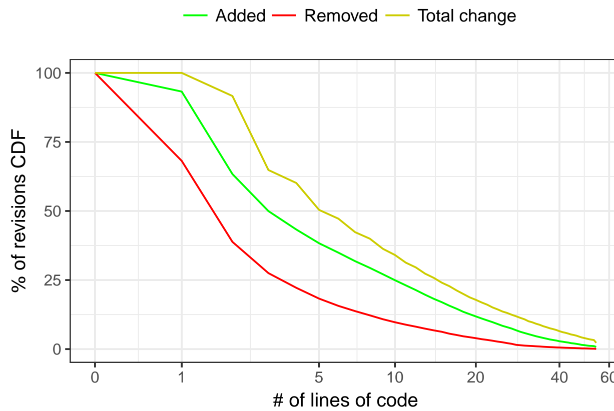


Fig. 11: Number of changed lines across the proportion of revisions. The y-axis represents the cumulative proportion of revisions, while the x-axis represents the number of affected lines of code. For example, 50% of the revisions have changed at least five lines of code.

correlation coefficients $r_s = 0.49$). Note that our finding is similar to prior studies on the frequency of changing other builds artifacts [17].

Docker Compose files exhibit small changes. The median number of added lines in a Docker Compose file per revision is just two, while the median number of removed lines is just one per revision. Overall, the median number of changed lines in a revision of a Docker Compose file is five as shown in Figure 11. We do not observe any correlation between the size (number of lines) of Docker Compose file and its number of changed lines on each commit (Spearman's rank correlation coefficients (r_s) = -0.01) as shown in Figure 12.

10.2% of the projects moved from a single-component to multi-component applications, 5.1% of the projects moved from a multi-component to single-component application. Our initial investigations on 10 Docker Compose files shows that applications removed components to cleanup their Docker Compose files, particularly, we observe five cases where applications removed images that were used for development purposes and which are no longer needed.

Most of the studied changes are related to image and data management options. 33.9% and 24.9% of the studied changes are related to the basic configuration (`Image Config` and `Data Management`) as shown in Figure 13a. Indeed, 27.6% and 24% of the changes modify the `image` and `volumes` options respectively as shown in Figure 14a.

On the other hand, a median of 47.7% (considering all used options) of the projects changed their used options. For example, 50.8% of the projects changed their used `networks` option as shown in Figure 14b. Furthermore, 61.6% of the 86 projects that used `Components Relation` category options,

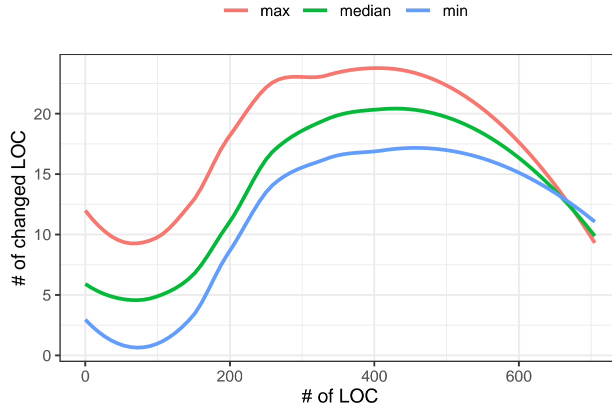


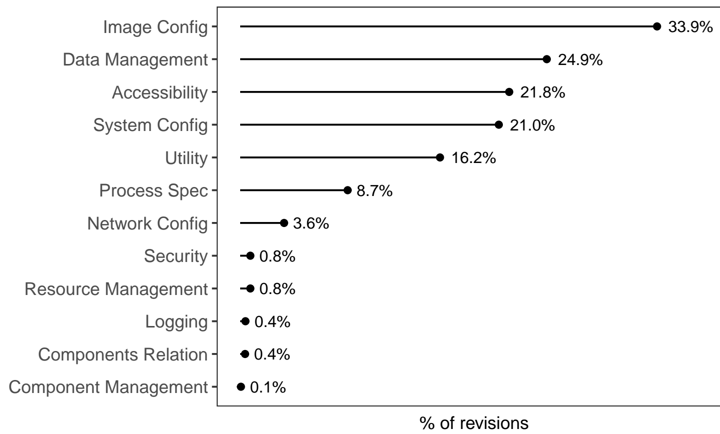
Fig. 12: Correlation between the number of lines of code and the number changed lines of code.

Table 3: Options that never changed in any revision of the studied Docker Compose files.

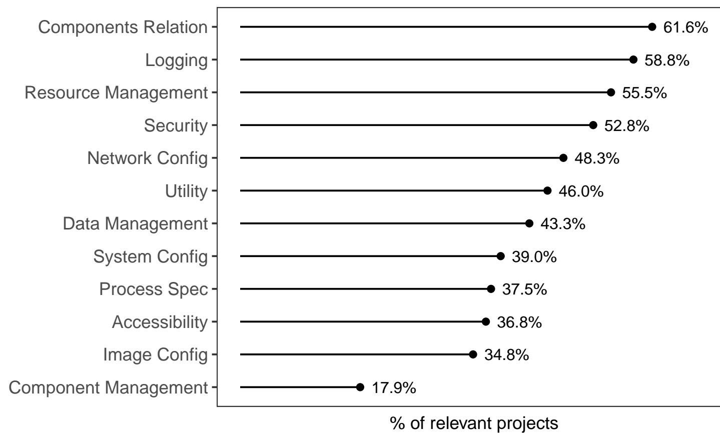
Category	Option	Supported versions	Description
Network Config	ipv6_address	2	Sets the IPV6 address of a network used by a component.
	enable_ipv6	2	Boolean value to enable or disable IPV6 addresses.
Resource Management	cpu_quota	1, 2	Sets the number of allowed CPU CFS (Completely Fair Scheduler).
Process Spec	endpoint_mode	3	Sets the method for component discovery.
	order	3	Sets the order of operations during an update or rollback.

changed those options. Similarly, 58.8% of the 68 projects that used `logging` category options, changed those options as shown in Figure 13b.

Applications are made more stable by pinning the versions of the used images. Most of the image related changes (85.7%) modify the version of the used image, while just 14.3% of those changes switch from one image to another one, as shown in Figure 15. 3.7% of these modifications remove the image's version, which is not a good practice [7]. In fact, version pinning is a better practice which is recommended by Docker for specifying an image in a Dockerfile [7], which is also applicable for writing Docker Compose files since applications also need to specify an image to build a component. Version pinning avoids any unintentional version upgrades of an image which may break the application. For instance, we observe 11 cases in our manual



(a) Percentage of revisions that change the different categories of Docker Compose options.

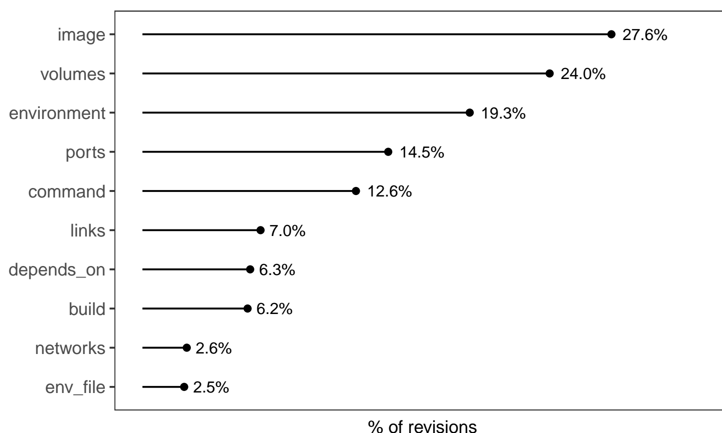


(b) Percentage of projects that use a category of options and revise it.

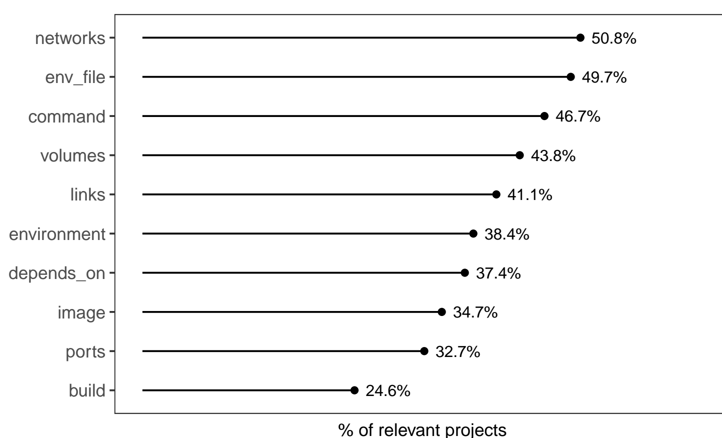
Fig. 13: The frequency of the changes of different option categories based on the analysis of 2,316 projects that revised at least one Docker Compose option. Note that in Figure 13b, the relevant projects are the projects that use a specific option category.

analysis where applications pined the exact version of their used images to fix issues that were introduced due to an update of the image.

Interestingly, we observe 10 cases where applications tag a version of their image similarly to tagging source code releases, which is a fundamental continuous integration principle where stable and reproducible infrastructure images are stored similarly to the source code [22]. For example, when the `'meenakommo64/squid'` project releases a new version such as `'3.3.8-23'`, it also releases a similar version for the infrastructure image such as



(a) Most frequently revised options based on the percentage of revisions.



(b) Most frequently revised options based on the percentage of relevant projects.

Fig. 14: Top 10 most frequently revised options in Docker Compose files based on the analysis of 2,316 projects that revised at least one Docker Compose option. Note that in Figure 14b, the relevant projects are the projects that use a specific option.

`'3.3.8-23'`. The goal of this tagging is to associate an infrastructure version to the release version so that applications can be easily re-built from a previous release using the appropriate infrastructure and environment.

The `volumes` option is the second most frequently changed option since it is changed in 60.6% of the studied revisions and by 24% of the studied projects. Applications commonly modify the way that their data is managed by modifying the `volumes` option. We observe three typical changes related to the `volumes` option: (1) 51.9% of the changes modify the

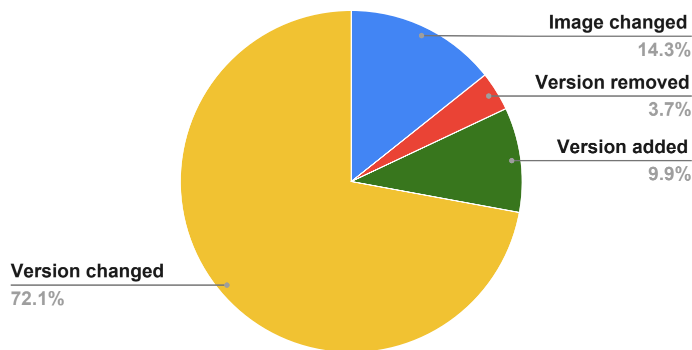


Fig. 15: Types of modification done on images.

volumes configuration, (2) 32.3% of the changes add a volumes to their components, and (3) 15.8% of the changes remove the volumes option. We discuss below each of these types of changes.

Our manual study of the 197 changes to the volumes option shows different reasons for which applications modify that option. We observe that applications modify the synchronization method of the mounted volumes, for example, whether the change in the component data will immediately be synchronized with the physical host or whether the data will be synchronized at the time of destroying the component as shown in Figure 16a. Applications also modify the values of volumes option to fix incorrect paths. Other modifications are related to the refactoring of the data and the path (11 cases) such as restructuring the files or variable substitution [9].

We also observe through our manual analysis three reasons for adding a volume: (1) persisting data in the host machine, (2) sharing data between components, and (3) improving the performance of an application. (1) Through our manual analysis, we observe that applications (22 out of the 58 volume addition cases that we were able to classify) add one or more entries to the volumes option to keep data persistent between the host and the component, as shown in Figure 16b. (2) They (8 out of the 58 volume addition cases) add the same named volume to multiple components to share the data among these components. (3) Finally, instead of uploading data from the host to the component, applications (35 out of 58 volume addition cases) add a volume that is directly accessible from components. That avoids uploading data into the component at each start-up or restart of the component, which can be time consuming for large datasets. For example, instead of cloning a project at each start-up or restart of a component, developers stored the cloned project in a volume that is shared with the container, as shown in Figure 16c. However, we observe cases that just intend to share certain configuration files and security files (e.g., Nginx SSL) with Docker containers when leveraging the volumes feature. Such a practice might be risky when these shared files do not have

Use delegated volume mount to increase performance on macOS.

```
Newer versions of Docker for Mac support caching options for macOS
volume mounts. Using the `delegated` mount option allows the container
and host to temporarily be out of sync, making the container's view
authoritative, which can improve performance because writes within the
container do not need to propagate to the host before they return.

Empirically, this slightly speed up the Webpack build from ~40s to ~35s
on my MacBook Pro.

https://docs.docker.com/compose/compose-file/#caching-options-for-volume-mou
```

```
2 docker-compose.yml
@@ -10,7 +10,7 @@ services:
10 10        ports:
11 11          - '8080:8080'
12 12        volumes:
13    -        - .:/usr/src/app
   13 +        - .:/usr/src/app:delegated
14 14        working_dir: /usr/src/app
```

(a) `volumes` modified

ability to load db snapshot into development

master

```
3 docker-compose.yml
...  ...  @@ -1,5 +1,8 @@
1  1      db:
2  2        image: postgres
   3  +     volumes:
   4  +       - .:/myapp
   5  +
3  6      web:
4  7        build: .
5  8        command: bash docker/start_web.sh
```

(b) `volumes` added

Mount open-streets project directly

master v2.3 … V1.1

lukasmartinelli committed on Nov 1, 2015 1 parent

Showing 2 changed files with 2 additions and 1 deletion.

```
1 docker-compose.yml
@@ -28,6 +28,7 @@ export:
28 28      build: ./src/export
29 29      volumes:
30 30       - ./export:/data/export
   31 +     - ./open-streets.tm2source:/data/tm2source
31 32      links:
32 33       - postgis:db
33 34      environment:
```

```
2 src/export/Dockerfile
@@ -12,7 +12,7 @@ RUN npm install -g tl \
12 12            tilelive-mapnik
13 13
14 14     # custom tm2source project
15    -  RUN git clone https://github.com/geometalab/open-streets.tm2source /data/tm2source
   15 +  VOLUME /data/tm2source
16 16     ENV SOURCE_PROJECT_DIR=/data/tm2source
17 17
18 18     # destination for exported vector tiles
```

(c) `volumes` added to share data from the host to the container

Remove unnecessary linkage to d1lod package

```
The worker doesn't need to load d1lod
```

master

```
1 docker-compose.yml
@@ -4,7 +4,6 @@ worker:
4  4          - graphdb
5  5          - redis
6  6        volumes:
7     -      - ~/src/d1lod/d1lod:/usr/local/d1lod
8  7        - ~/src/d1lod:/dump
```

(d) `volumes` removed

Fig. 16: Examples of different types of changes to the `volumes` option.

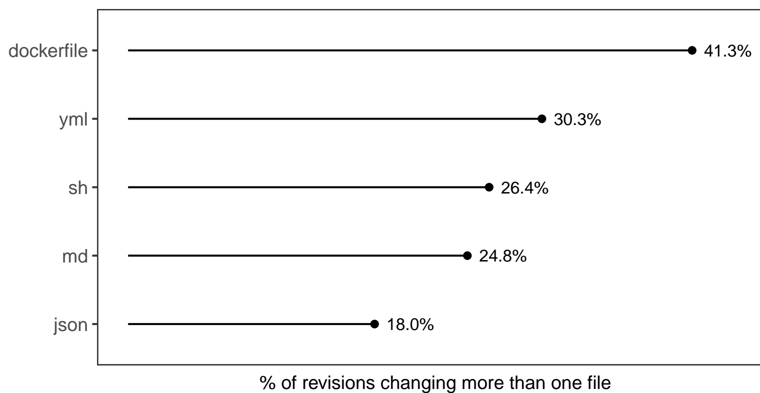


Fig. 17: Top five types of files that co-evolve the most with the Docker Compose file.

the appropriate permissions as they can be accidentally modified or removed from a running Docker component. We observe an interesting commit that replaced a path in the `volumes` with several other paths since there is *"no way to exclude a directory, the only way to make this work is to share multiple, smaller directories"*[2].

Finally, applications remove the `volumes` options in 15.8% of the changes related to that option in order to remove unnecessary resources from their components so that the components become more lightweight as shown in Figure 16d. In addition, applications replace `volumes` option with `tmpfs` option to use the data temporarily without writing it in the file-system.

Docker Compose files are coupled with other files that are often used to configure the infrastructure of an application as well. In 70.4% of the revisions to the Docker Compose file, the file is changed with a median of two other files. While Dockerfile is the most co-evolving (41.3% of the 70.4% of the changes) file with Docker Compose as shown in Figure 17. We also observe that `.yml` files co-evolve in 30.3% of the revisions suggesting that applications also revise additional Docker Compose files other than their primary Docker Compose files. Note that we just studied the `docker-compose.yml` file which is the default file that is considered by Docker Compose when composing an application.

Based on a manual analysis of a representative random sample (out of 7,592 commits with a 95% confidence level and a 5% confidence interval) of 366 co-evolution of a Dockerfile and a Docker-compose file, we observe different patterns that are straightforward or project specific, **making it hard to predict when a Dockerfile would co-change with a Docker-compose file and vice-versa**. Among the straightforward co-changes is the addition or removal of a component with its Dockerfile, changing the path location of

[2] https://github.com/startnayit/warehouse/commit/116962abc6437c35056f3fb4c9c0a66fca6c2790

a Dockerfile and updating the Docker-compose with the new path, co-changes where practitioners upgrade all of their dependencies including the dependencies defined in the Dockerfile as well as the Docker-compose, and opening a port in the Dockerfile and using it in the Docker-compose. We also observe project specific cases such as moving the declaration of certain resources from the Dockerfile to the Docker-compose or vise-versa and overriding in the Docker-compose an environment variable that is defined in the Dockerfile. In fact, deciding which environment variable to override depends on the context of the project.

Summary of RQ3

Docker Compose files are quite stable with very few lines of code changed when such files change. In addition, applications are made more stable by pinning the used versions of the images that are used by their components. Finally, applications adjust their `volumes` in order to synchronize the data outside the components.

RQ4. How do applications use different versions of Docker Compose?

Motivation: The goal of this research question is to identify how applications upgrade/downgrade the version of their Docker Compose, and the challenges that they face as they perform such activities, so the developers of Docker Compose would have a better understanding of how applications are adopting their versions.

Approach: There are currently three main versions of Docker Compose. Version 1 was released on November, 2015, while version 2 was released on April, 2016 and version 3 was released on January, 2017. Docker Compose team has announced that Version 1 will become deprecated in a future release. Each version adds support for new options while removing rarely used options.

To better understand the versions of Docker Compose that applications use and how often they upgrade/downgrade across these versions, we collected the used Docker Compose version for each studied project by parsing the value of the `version` option. For the first version of Docker Compose, no `version` option existed hence files with no `version` option are considered as using version 1.

We also studied the upgrade/downgrade of Docker Compose versions by analyzing all code changes that modified the `version` option. We also manually investigated all of the changes that downgraded the Docker Compose Version (107 commits) to better understand the rationale behind such downgrades.

Results: Most projects ($\sim$ 90%) stick to their initial Docker Compose version. 9.8% of the studied projects changed their used Docker Compose

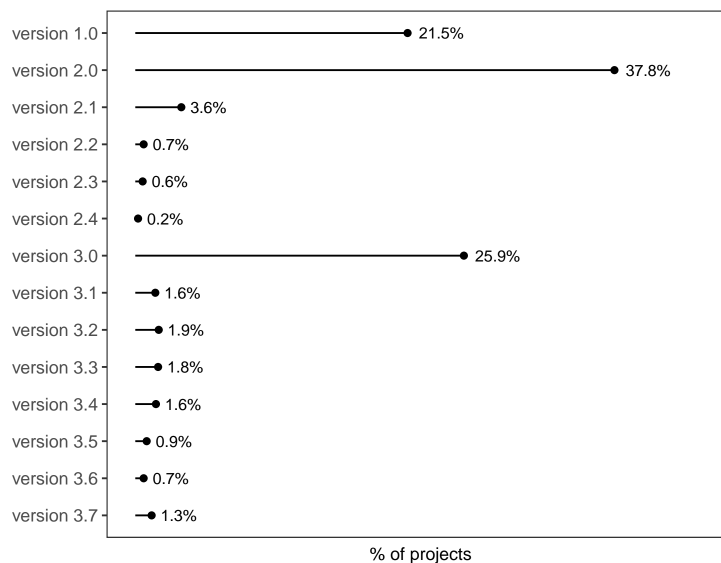


Fig. 18: Exact versions of Docker Compose used by the percentage of projects.

version. Such a change to the version line is often associated with additional changes. In particular, a median of 10 lines of code are changed when a Docker Compose version is changed – double the number of changed lines of code in a regular revision (as discussed in RQ2). Such additional changes might be due to the major syntactical differences between the different versions, the configuration of new options that are offered by the new version, or to deal with the removal of options in the new version.

8.5% of the projects upgraded their Docker Compose version while 2.4% of the projects downgraded their Docker Compose version. Several (59.9%) of the upgrades/downgrades are between major versions, although some upgrades/downgrade occur between sub-versions. Almost all (99%) changes between major versions are between versions 2 and 3. We only find three cases where the version change (upgrade/downgrade) was between versions 1 and 2. Since the structure of Docker Compose has gone through major changes between these two versions, moving between these two versions might take a considerable amount of effort.

21.5% of the studied projects use Docker Compose version 1 which will be deprecated soon according to the documentation of Docker Compose [13] (as of August, 2019). Docker Compose should consider developing a migration support plan for these projects given the large differences between version 1 and the versions 2 and 3. 35.6% of the projects used the latest versions of Docker Compose file (version 3 and its sub-versions), while 42.9% of the projects used version 2 of Docker Compose and its subversions as shown in Figure 18. Although Docker Compose documentation

```
7 docker-compose.yml
@@ -1,4 +1,4 @@
1     - version: '3'
   1  + version: '2'
2  2
3  3    services:
4  4      app:
@@ -21,8 +21,13 @@ services:
21 21        image: mongo:3.6
22 22        ports:
23 23          - 27017:27017
   24 +      volumes:
   25 +        - ./data/mongo:/data/db
24 26
25 27      elasticsearch:
26 28        image: elasticsearch:5.6
27 29        ports:
28 30          - 9200:9200
   31 +      volumes:
   32 +        - ./data/elasticsearch:/usr/share/elasticsearch/data
   33 +      mem_limit: 512m
```

Downgraded version

Not supported in version 3

Fig. 19: An example of a Docker Compose version downgrade. This change downgraded the Docker Compose version from version 3 to version 2.

recommends applications to specify the used minor version of Docker Compose [11], 79.2% of the studied projects that use version 2 or 3 of Docker Compose, do not specify the minor Docker Compose version.

Many unused options have been removed in the latest versions of Docker Compose. 21 out of the 26 options that were never used in our studied projects (as per RQ1) are no longer supported by the latest Docker Compose version (i.e., version 3), 19 of these options were introduced in version 2 and are no longer supported in version 3 of Docker Compose. On the other hand, 15 other currently-in-use options in version 2 have been removed from version 3. These 15 options are used by a median of two projects (minimum of one and a maximum of 206 projects per option).

Platform and option compatibility are the main drivers for downgrading the Docker Compose version. We manually studied all of the 107 commits that downgraded the Docker Compose version. We read through each commit message, examined the changed code and studied the official documentation of Docker Compose versions to identify the rationale for such downgrades. We identified two major reasons for downgrading the version: (1) platform (e.g., AWS and Travis CI) compatibility, and (2) option compatibility.

Cloud platforms, like AWS, on which applications are deployed might not support the latest version of Docker Compose, forcing a version downgrade (e.g., from Docker-Compose 3.6 to 2.4 for AWS incompatibility). Also, applications downgraded their Docker Compose version to be compatible with Travis CI (e.g., from version 3 to 2). Finally, applications downgrade to an older version (version 2) to use options that were removed in a later version (version 3) of Docker Compose as shown in Figure 19.

Summary of RQ4

Many applications are using the oldest version of Docker Compose (even though such version will be deprecated in a near future release). Migrating from one version to another is not straightforward due to compatibility issues with the host (e.g., AWS) or due to the removal of options in new releases of Docker Compose.

5 Implications

We observe through our study that applications use Docker Compose in much more basic ways than envisioned. In fact, applications use Docker Compose without composing multiple images (RQ1), and they use (RQ2) and change (RQ3) just the basic options of Docker Compose, and they hardly upgrade their versions (RQ4). Even when they do so, applications face compatibility issues. Therefore, to improve Docker Compose and help applications fully benefit from its options, we recommend the following suggestions:

We recommend Docker to consider how developers use Docker-compose to better design new versions of Docker-compose. While the primary goal of Docker-compose is to compose multi-component applications, 26.8% of the applications that use it are just single-component applications. Among these single-component applications, Docker-compose is just to simplify the Docker command line, which suggests that Docker should investigate how to simplify the usage of Docker. We also observe that multi-component applications use images from DockerHub, while the DockerHub images are different from each other even for the same software system according to [23], which suggests that Docker-compose should recommend appropriate images when composing a multi-component application.

We also observe that there is a need for better mechanisms to manage volumes. For instance, there is a need for a mechanism to exclude paths from the volumes that are shared between a host and components. Additionally, we observe that a volumes are not used solely to synchronize data between a component and its host (e.g., storing a database in a host outside a component so data is kept after restarting or removing a component), but also to share data in one-way fashion from the host to the component. Such practice might be risky as a component might accidentally modify or even remove data from

the host; so Docker should consider a clear separation between the two main usages of the volumes feature. While Dockerfile has the `COPY` feature, no similar feature exists for Docker-compose.

We recommend Docker to consider how projects use Docker-compose, so Docker can add needed options. Developers commonly have many components that use the same Docker image; leading to many definitions of components with a large amount of duplication (as these components are often quite similar modulo some minor differences). Instead Docker Compose should consider adding the option to create component templates to reduce duplication. Surprisingly, versions 1 and 2 of Docker Compose had an `extends` option which helped with the reuse of the configuration of components (similar to inheritance in object oriented programming). However, that option is no longer supported in version 3 of Docker Compose.

We recommend Docker to investigate unused options. 22.6% of the studied options were never used in our studied applications. We observe that developers focus primarily on the basic options of Docker Compose, while other advanced options are not used much. These options might be important since they can have an impact on the security and the quality of an application. Therefore, we suggest Docker-compose to investigate the need for these options and whether they should be removed.

We recommend Docker to consider how its options are used by practitioners before removing them. We observe that a typical upgrade/-downgrade of Docker Compose versions takes twice the effort of a regular revision since Docker Compose removes a large number of options from one version to another. For example, Docker Compose removed 36 options in the version 3. However, 15 currently-in-use options, by a median of 2 and up to 206 projects, are removed from version 3.

We recommend Docker to propose options to use in earlier stages to prevent errors. We observe that developers change `volumes` to improve the performance of their applications, while such performance degradation could be prevented earlier. Similarly, we observe 11 cases in which developers pin an image version after an incompatibility issue.

We recommend Docker to consider incompatibility issues when releasing a new version of Docker-compose. We observe cases where developers downgraded their Docker-compose version due to incompatibility issues with a Cloud platform or a CI pipeline, such as AWS and Travis CI.

6 Threats to Validity

6.1 External Validity

Our external threat to validity concerns the generalization of our results. We cannot generalize our results to multi-component applications that configure the Docker Compose using other files than `docker-compose.yml` and other

projects. However, our analysis considers a large number (4,103) of open source Github projects.

6.2 Internal Validity

Our first internal threat to validity concerns the maturity of the projects that we studied. In fact, some projects might be just personal projects (e.g., in which one learns how to use Docker Compose). To mitigate this risk, we selected projects that have at least 100 commits.

A second internal threat to validity concerns the analysis of the right Docker Compose file, which is used by a studied project. A project might define multiple Docker Compose files and studying the not used file might lead to incorrect conclusions. To mitigate this risk, we focus on projects that use just one `docker-compose.yml` file.

A third internal threat to validity concerns the threshold of 10 components that we consider for large multi-component systems, which we qualitatively studied. While our threshold leads to a large number of 110 manually studied large multi-component systems, we encourage future studies to investigate a different threshold.

7 Conclusion

In this paper, we conduct an empirical study to better understand the use of Docker Compose. We observe that applications do not fully benefit from the available options nor versions of Docker Compose. Indeed, 26.8% of the projects build their applications from a single image, while the primary goal of Docker Compose is to compose a multi-component application. In addition, we observe that applications use and maintain just the basic options of Docker Compose, while more advanced options are almost ignored by applications. Finally, we observe that few projects upgrade their version of Docker Compose. Future studies are needed to better understand how to improve the uptake of the more advanced aspects and versions of Docker Compose, if they are needed at all.

References

1. 451research. 451 research. https://451research.com, 5 2019. [Online; last accessed: 23 May, 2019].
2. R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Sigmod Record*, pages 207–216, June 1993.
3. Brown. Referencing docker images. https://windsock.io/referencing-docker-images, 4 2015. [Online; last accessed: 26 August, 2019].

4. J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall. An empirical analysis of the docker container ecosystem on github. In *14th International Conference on Mining Software Repositories*, pages 323–333, May 2017.
5. Datadog. Datadog. https://www.datadoghq.com, 5 2019. [Online; last accessed: 23 May, 2019].
6. Datadog. Docker adoption. https://www.datadoghq.com/docker-adoption, 5 2019. [Online; last accessed: 23 May, 2019].
7. Docker. Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices, 2 2017. [Online; last accessed: 22 August, 2019].
8. Docker. Docker. https://www.docker.com, 5 2019. [Online; last accessed: 23 May, 2019].
9. DockerCompose. Environment variables in compose. https://docs.docker.com/compose/environment-variables, 2 2017. [Online; last accessed: 27 August, 2019].
10. DockerCompose. Overview of docker compose. https://docs.docker.com/compose, 2 2017. [Online; last accessed: 23 August, 2019].
11. DockerCompose. Compose file versions and upgrading. https://docs.docker.com/compose/compose-file/compose-versioning, 10 2018. [Online; last accessed: 23 August, 2019].
12. DockerCompose. Docker compose version 1. https://docs.docker.com/v17.09/compose/compose-file/compose-file-v1, 6 2019. [Online; last accessed: 16 July, 2019].
13. DockerCompose. Docker compose version 1. https://docs.docker.com/v17.09/compose/compose-file/compose-versioning/#version-1, 6 2019. [Online; last accessed: 16 July, 2019].
14. DockerCompose. Docker compose version 2. https://docs.docker.com/compose/compose-file/compose-file-v2, 6 2019. [Online; last accessed: 16 July, 2019].
15. DockerCompose. Docker compose version 3. https://docs.docker.com/v17.09/compose/compose-file/compose-versioning, 6 2019. [Online; last accessed: 16 July, 2019].
16. DockerHub. Build and ship any application anywhere. https://hub.docker.com, 5 2019. [Online; last accessed: 23 May, 2019].
17. K. Gallaba and S. McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. In *IEEE Transactions on Software Engineering*, pages 1–1, 2018.
18. Google. Google big query. https://console.cloud.google.com/bigquery?p=bigquery-public-data, 8 2019. [Online; last accessed: 23 August, 2019].
19. G. Gousios. The ghtorrent dataset and tool suite. In *10th Working Conference on Mining Software Repositories*, pages 233–236, 2013.
20. J. Henkel, C. Bird, S. K. Lahiri, and T. Reps. Learning from, understanding, and supporting devops artifacts for docker. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 38–49.

IEEE, 2020.

21. E. Horton and C. Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 328–338, 2019.
22. J. Humble and D. Farley. Continuous delivery: Reliable software releases through build. In *Test, and Deployment Automation*, pages 2013–01, 2010.
23. N. Muhtaroglu, B. Kolcu, and İ. Arı. Testing performance of application containers in the cloud with hpc loads. In *5th International Conference On Parallel, Distributed, Grid And Cloud Computing For Engineering*. Civil-Comp, 2017.
24. Serverwatch. Container revenue growing to 2.7b by 2020. https://www.serverwatch.com/server-news/container-revenue-growing-to-2.7b-by-2020.html, 5 2019. [Online; last accessed: 23 May, 2019].
25. R. Shu, X. Gu, and W. Enck. A study of security vulnerabilities on dockerhub. In *7th ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.
26. B. Tak, H. Kim, S. Suneja, C. Isci, and P. Kudva. Security analysis of container images using cloud analytics framework. In *Web Services*, pages 116–133, 2018.
27. A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *26th International Conference on Software Analysis, Evolution and Reengineering*, pages 491–501, Feb 2019.
28. Y. Zhang, H. Wang, and V. Filkov. A clustering-based approach for mining dockerfile evolutionary trajectories. In *Science China Information Sciences*, volume 62, pages 19101:1–19101:3, 2019.
29. Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In *42nd Annual Computer Software and Applications Conference*, volume 01, pages 138–143, July 2018.