A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub

Changyuan Lin*, Sarah Nadi† and Hamzeh Khazaei‡
*Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada
†Department of Computing Science, University of Alberta, Edmonton, AB, Canada
‡Department of Electrical Engineering and Computer Science, York University, Toronto, ON, Canada
Email:{changyua@ualberta.ca, nadi@ualberta.ca, hkh@yorku.ca}

***Abstract*—Docker is currently one of the most popular containerization solutions. Previous work investigated various characteristics of the Docker ecosystem, but has mainly focused on Dockerfiles from GitHub, limiting the type of questions that can be asked, and did not investigate evolution aspects. In this paper, we create a recent and more comprehensive data set by collecting data from Docker Hub, GitHub, and Bitbucket. Our data set contains information about 3,364,529 Docker images and 378,615 git repositories behind them. Using this data set, we conduct a large-scale empirical study with four research questions where we reproduce previously explored characteristics (e.g., popular languages and base images), investigate new characteristics such as image tagging practices, and study evolution trends. Our results demonstrate the maturity of the Docker ecosystem: we find more reliance on ready-to-use language and application base images as opposed to yet-to-be-configured OS images, a downward trend of Docker image sizes demonstrating the adoption of best practices of keeping images small, and a declining trend in the number of smells in Dockerfiles suggesting a general improvement in quality. On the downside, we find an upward trend in using obsolete OS base images, posing security risks, and find problematic usages of the `latest` tag, including version lagging. Overall, our results bring good news such as more developers following best practices, but they also indicate the need to build tools and infrastructure embracing new trends and addressing potential issues.**

I. Introduction

Containers provide an isolated run-time environment for application execution; they encapsulate the application together with its dependencies, including source code, files, and environment variables. Containers only isolate system processes and resources but share the operating system kernel, making them lightweight, easy to deploy, scale, and migrate. Containerization is widely used in cloud computing [1]–[3] and in applications with a micro-service architecture [4].

Docker [5] is currently one of the most popular containerization solutions [6]. An application encapsulated by Docker is distributed in the form of a *Docker image*, an executable package that includes everything needed to run the application [7]. *Dockerfiles* are descriptive files of Docker images that specify all the needed information about the environment and program execution. Dockerfiles therefore play a key role in creating the software run-time environment, and their quality may affect the reproducibility and quality of the resulting image.

Previous work by Cito et al. [8] studied Dockerfiles hosted on GitHub on October 2016, exploring various characteristics such as smells in Dockerfiles and popular base images. While that study was the first of its kind on Dockerfiles and shed light on various characteristics of the Docker ecosystem, it, along with other later similar studies [9]–[12] focused on mining information solely from code repositories, namely GitHub. Xu and Marinov argue that image repositories contain valuable operations data as well as usage information about containers and that such repositories should be mined for related studies, instead of focusing only on code repositories [13]. We use this argument as inspiration to collect additional information about Docker images, which allows us to re-investigate some of the previously studied phenomenon on a larger and more comprehensive data set, as well as to study how the Docker ecosystem has evolved over these past years.

Specifically, we use Docker Hub [14], the de facto Docker image repository, as our primary source of data. Images hosted on Docker Hub can be directly used in any Dockerfile and are thus more likely to contain images that get used in practice by developers worldwide. We then link this data to the image's source repository hosted on GitHub or Bitbucket. We create a data set containing information about 3,364,529 Docker images, covering 98.38% of images hosted on Docker Hub as of May 3, 2020. From these images, we successfully pinpoint 378,615 git repositories behind them. To the best of our knowledge, this is the largest and most recent available data set of Docker images and corresponding source repositories. More importantly, this is the first data set of Docker images that is based on Docker Hub, GitHub, and Bitbucket, rather than solely on GitHub. Docker Hub provides additional insightful data, such as the image tags (similar to release names or tags), image sizes, image builds, and the number of times an image has been pulled from the registry.

We leverage this new, more comprehensive data set to reproduce previously investigated characteristics, investigate new phenomenon, and for the first time study evolution trends in the Docker ecosystem over the last 5 years. Figure 1 provides a summary of our data set and the empirical study we conduct, which involves the following research questions:

RQ1: What is the current state of Docker image development and how did it evolve over time? To understand the current state of Docker and how it evolved, we start with a basic characterization of Docker images in terms of programming languages, base images, OS versions, image architectures, and image sizes. The first 2 characteristics were

previously studied [8], but the last 3 are new characteristics we can study thanks to the data we collect from Docker Hub. Our results show changes in programming languages, e.g., CSS no longer in top languages while TypeScript is gaining usage. We also find the rise of Alpine as an OS base image, increasing popularity of language run-time and application base images, use of obsolete OS base images, growth of non-AMD64 images, and a declining trend of image sizes.

RQ2: What are the current Docker image tagging practices? Docker images are tagged with a tag name to allow versioned development. Using data from Docker Hub, we, for the first time, investigate how image tag names are used and which tagging approaches are followed. We find that some 50.48% of the studied images rely only on the `latest` tag, which is error-prone and often points to the incorrect version [15]–[18]. We also find that semantic release tagging is more commonly used, compared to SHA pinning, which is criticized for being unreadable [17], [19].

RQ3: How do Dockerfiles co-evolve with other source code? Previous work studied how non-source-code files, e.g., build files [20], [21], co-evolve with source code. Similarly, we investigate, for the first time, the co-evolution of Dockerfiles and other source code. We find that Dockerfiles evolve at a slower rate and a smaller scale than other source code.

RQ4: How prevalent are code smells in Dockerfiles? Code smells are often used as a proxy for code quality. In contrast to results from 2016 [8], we find that some smells such as those causing larger image sizes are no longer as prevalent. Instead, not catching pipe error and incorrectly using the `cd` command have become more prevalent. We additionally study the evolution of these smells and find a general declining trend in the number of smells in Dockerfiles over time.

Our work provides a characterization of the current state of the Docker ecosystem and how it has evolved over time. Some of our findings, such as which base images, tagging practices, and smells are common allow image developers and tool builders to make informed decisions about how to develop their containers and build container-based services while avoiding problems. We also bring some good news; for example, best practices, such as using smaller and tagged base images, and keeping images small are being adopted more.

Our results can also benefit cloud service providers (e.g., AWS ECS & Lambda), who can provide support and tooling for fast-growing languages and cache layers of popular base images to accelerate the online image build. On the other hand, our findings show that there are still some issues with the Docker ecosystem that need to be addressed, such as the use of outdated OS base images, and improper use of the `latest` tag. Such issues can be addressed with tools that can be integrated in Docker Hub's continuous integration service.

In Summary, this paper makes the following contributions: (1) We create, to the best of our knowledge, the largest available Docker data set collected from both real-world images hosted on Docker Hub, and their source repositories from both GitHub and Bitbucket. It contains data about 3,364,529 Docker images and 378,615 corresponding git repositories. (2) Using this new data set, we reproduce phenomenon studied by previous work [8] and compare our results. (3) We investigate new evolution trends to understand the changes in the Docker ecosystem as well as how Dockerfiles co-evolve with other source code. (4) Given the new data from Docker Hub, we study Docker image tagging practices as well as the evolution of image sizes and architectures. Our data set and all analysis scripts and results are available on our artifact page [22].

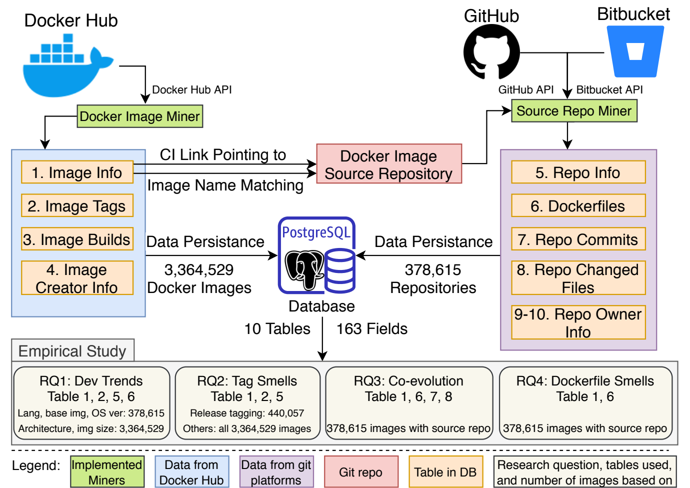

Fig. 1: Overview of the methods for data collection and overview of our empirical study. We indicate the data used for each RQ.

II. BACKGROUND AND TERMINOLOGY

a) Docker: Docker is currently one of the most popular containerization solutions. An application encapsulated by Docker is distributed in the form of a *Docker image*. The Docker image is defined through a *Dockerfile*, which contains all commands and environment variables required to support the application's execution. When the target application encapsulated in the image is deployed, an instance of the image is created [14] and is referred to as the *Docker container*. Hence, a Docker image is like a snapshot of the target application and allows creating containers in a reproducible way.

One of the advantages of the Docker ecosystem is that it is not necessary to write an entirely new Dockerfile to develop a new Docker image. Similar to inheritance in object-oriented programming, a Docker image can inherit image definitions from another *base image* by using the `FROM` command. In this case, all properties and files encapsulated in the base image are inherited by the new image. Any Docker image can be extended and used as a base image. Listing 1 shows a Dockerfile defining a Docker image for executing a Python script. In this example, the image developer only needs to specify the base image for the target Python run-time environment by using the `FROM python:3.7.3-stretch` statement. By doing so, all properties and files encapsulated in the official Python 3.7.3 language run-time image are inherited. The remainder of the Dockerfile contains the necessary instructions required to run the application, which will run in Python 3.7.3.

While building an image, Docker executes statements from the Dockerfile and generates a layer for each instruction.

```
FROM python:3.7.3-stretch
COPY something.py /
CMD ["python3","-u","something.py"]
```

Listing 1: Example of a Dockerfile

Similar to git commits, each layer contains only a collection of differences from the previous layer. The Docker image is built from a massive pileup of layers. Each Docker image has a unique SHA-256 code as its ID. For ease of versioned image development and management, Docker also provides a tagging mechanism where image developers can provide tag names for each version. Hence, in Dockerfiles or in Docker commands, a desired version of image can be referred to using either its SHA256 ID or `image name:tag name`. Listing 1 uses the latter. When no tag name is provided, the default tag name is `latest`, which should point to the latest image version. However, the reliance on `latest` is problematic [15]–[18]; we discuss these problems more in the motivation for RQ2.

b) Docker Hub: A *Docker image registry* is a place to host, index, and manage Docker images. Docker Hub [23], launched in 2014, is the largest, free and public Docker image registry in the world. It is home to more than 3.4 million public Docker images as of May 3, 2020, and the number is still growing. Images published by a few certified companies and organizations are considered as *official* images while images posted by community developers are *non-official* images. Most available Docker images use an official image (e.g., from Ubuntu) as their base image. Docker images also support various hardware architectures, such as ARM and x86. For each image, Docker Hub records the *pull count*, which is the number of times an image was pulled from Docker Hub.

There are mainly three types of official images: OS, language run-time, and application. Docker images that only encapsulate a base operating system are *OS Docker images*; examples include Ubuntu and Debian. *Language run-time images* are Docker images that provide the run-time environment for a given programming language, such as Python and Golang. Docker images that encapsulate a ready-to-use application are *application images*, such as Nginx and PostgreSQL.

Docker Hub provides a continuous integration (CI) service that links the GitHub or Bitbucket source repository containing the Dockerfile and other files of a Docker image [24]. When new code is pushed in the linked source repository, Docker Hub automatically triggers an image build. On Docker Hub, Docker images using such a service have a link to their GitHub/Bitbucket source repository and have additional image information, such as the Dockerfile of the latest successfully built image and build status, on their overview page.

III. Related Work

a) Docker studies: Despite the short history of the Docker ecosystem, there is already some work on Dockerfiles. Cito et al. [8] studied 70,000 Dockerfiles found on GitHub in October 2016. They focused on the Dockerfiles stored in GitHub repositories and did not consider information about the corresponding Docker images, such as their tags or image sizes. They also did not capture evolution trends. Besides, given how Docker is becoming increasingly popular, the dataset and empirical results obtained in their work is currently dated. Schermann et al. [9] developed a toolchain to collect structured information about the state and the evolution of Dockerfiles on GitHub and released it as a public data dump as of January 2018; they did not do a full analysis of this information though. In their work, they concentrated on the Dockerfile collection and obtained about 100,000 Dockerfiles. The dataset they released was from a single source and did not contain the information about Docker images either. In their work, they proposed several future research questions, and our RQ3 is inspired by one of them. In terms of quality issues of Dockerfiles, Tak et al. [10] focused on extracting quality issues related to security caused by compliance violations and outdated dependencies. Similarly, Shu et al. [25] study common security vulnerabilities in Docker images.

Very recently, Henkel et al. [11], [12] used rule mining based on historic changes of Dockerfiles hosted on GitHub to identify 15 rules that can be used to enforce best practices. They then developed a rule enforcement engine, `binnacle`, that can parse Dockerfiles and flag rule violations, and evaluated it on 178,000 Dockerfiles collected from GitHub. Their work focuses more on smells in the bash commands used in Dockerfiles, while the smells we consider in RQ4 stem from both bash commands and official best practices of writing Dockerfiles [26] (e.g., using an untagged version of an image). Since their work was only recently published, future work can involve applying `binnacle` to our much larger data set. That said, we note that two of the smells they identify using rule mining are already included in the smells we study (specifically DL3015 and DL3009).

In summary, the above studies focused on Dockerfiles collected from GitHub. The arguments provided by Xu and Marinov [13] for the importance of looking at container image repositories inspired us to mine data from Docker Hub. Thus, our work is different in that, in addition to studying a larger and more recent data set, we also collect information from Docker Hub, which contains image-specific information, such as the tag names, image sizes, and image builds, which are indispensable for a holistic analysis of the Docker ecosystem. Additionally, none of the above work studied evolution trends to understand how the Docker ecosystem is changing. More specifically, RQ2, RQ3, and parts of RQ1 (image sizes, OS versions, and image architectures) are new. While the first parts of RQ1 and RQ4 reproduce and compare previous results [8], the second parts studying evolution trends are new.

b) IaC studies: Dockerfiles fall under the broad umbrella of infrastructure as code (IaC) [27], so we also look at IaC studies for general insights. Cito et al. [28] studied the development process of cloud applications. They discovered that IaC scripts were generally maintained by dedicated engineers, and there was a tendency to shift from the IaC tools, including Puppet, to containers for configuration management and automation. Jiang et al. [29] found that IaC script and source code evolved at a similar rate, while Sharma et al. [30]

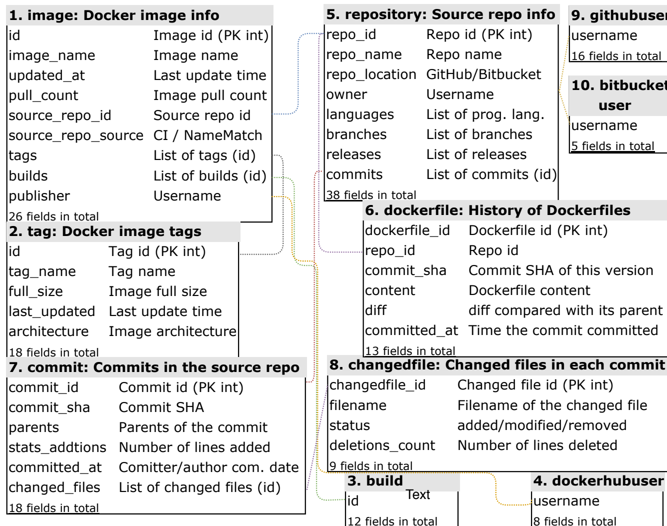

Fig. 2: Data set schema and partial fields description. The full information is available on our artifact page [22]

and Rahman et al. [31] investigated code and security smells in Puppet repositories. Our RQ3 and RQ4 are similar to these goals but focus on Dockerfiles rather IaC scripts.

IV. Data Set Curation

Figure 1 shows an overview of our data collection process, which we now describe in detail.

a) Docker Image Collection: To collect data about real-world Docker images, we focus on public Docker images owned by community developers and open-source organizations. This is because information about a Docker image published by certified companies requires payment. Additionally, the images are not subject to any open-source license and private images are not accessible without credentials.

Using the Docker Hub API [32], we develop `Docker Image Miner` and collect information about 3,364,529 public Docker images hosted on Docker Hub. Compared to the total number of images shown by the Docker Hub search engine [33], which lists all hosted public Docker images, our data covers 98.38% of public images hosted on Docker Hub as of May 3, 2020. We mine relevant information for each image, such as tag name, image size, image pull counts, and the link to its source repository if it is using the CI service described in Section II. We find that 375,518 out of the collected 3,364,529 Docker images are using the aforementioned CI service. Among these Docker images, 357,030 of them link to a GitHub repository, and 18,488 of them link to a Bitbucket repository. We find that Docker images using the CI service typically have a higher median image pull count (98.0 vs 18.0 with the exact distribution on our artifact page [22]).

b) Source Repository Collection: To collect data about source repositories associated with Docker images, we first focus on the 375,518 images hosted on Docker Hub that use the CI service for automatic builds. Through the CI link of each image, we can pinpoint the corresponding GitHub/Bitbucket source repository, where we have a full view of the source code encapsulated in the Docker image, enabling us to answer RQs involving other source code. We develop `Source Repo Miner` that uses the GitHub or Bitbucket APIs [34], [35] to collect data about the source repository, including basic repository information (language, size, releases, etc), Dockerfiles in the repository, commits, and changed files in each commit.

Our goal is to find corresponding source repositories for as many Docker images as possible. Thus, we design an additional name matching method that allows us to detect this relationship, even if the image does not explicitly use the CI service. Docker Hub and GitHub/Bitbucket use the same naming convention for Docker images and GitHub repositories, which is "*username/repository name*". There is a high chance that some developers use the same username and repository name on Docker Hub and GitHub/Bitbucket to manage a Docker image project. Thus, for each image in our data set that does not explicitly use the CI service, we search for a repository on GitHub and Bitbucket that has the same name. This allowed us to obtain data for an additional 64,539 images with GitHub/Bitbucket repositories, resulting in a total of 440,057 images with associated source repositories.

c) Data Collection and Storage: We containerize both `Docker Image Miner` and `Source Repo Miner` using Docker and deploy them on a large Kubernetes cluster with 40 nodes distributed across the country. The miners can continuously obtain new data from Docker Hub and the associated GitHub/Bitbucket repositories, or update existing data. We successfully create a data set containing information about 3,364,529 Docker images hosted on Docker Hub and 440,057 associated source repositories. Note that multiple images can link to the same repository; there are a total of 378,615 *unique* repositories in our data set. These 378,615 repositories have a total of 23,376,349 commits and 124,192,028 changed files in those commits. The repositories contain 434,303 Dockerfiles with their history versions in a subset of 1,920,195 corresponding commits. Overall, we collect a total of 163 fields in 10 tables, and store the data in a PostgreSQL database. Figure 2 provides the data set schema and parts of the field descriptions; the remaining descriptions and the source code of miners are on our artifact page [22].

d) Use of Data in RQs: Figure 1 shows a high-level overview of the stored tables and specifies which tables and data are used in each research question. Out of 10 database tables, we use 6 of them in our empirical study. Answering some questions (image size and architecture analysis in RQ1 and all of RQ2 except release tagging analysis) require only image data from Docker Hub and so we use information about all 3,364,529 Docker images to answer those. Answering questions that require the actual Dockerfiles (base image analysis in RQ1 and Dockerfile smells in RQ4) or repository information (programming language analysis in RQ1) or commit information (RQ3) is based on the collected 378,615 unique associated GitHub/Bitbucket repositories.

V. Empirical Study

We now present the results of our empirical study. For each RQ, we provide a brief motivation behind the question,

describe the methods used to answer it, and then present our findings and their implications.

A. RQ1: Characterization of Docker Image Development & Its Evolution

a) Motivation: Before diving into specifics and potential problems in the Docker ecosystem, we start with a general characterization of the current state and how it evolved. We study the programming languages, base images, OS versions, image architectures, and image sizes of Docker images. Understanding these basic characteristics and their trends can shed light on general community practices and how the ecosystem is evolving. Additionally, evolution trends in terms of used base images, OS versions, and image sizes can identify potential security risks and if official best practices are being followed.

b) Method: For the analysis of programming languages, base images, and OS versions, we use information from the 378,615 unique GitHub/Bitbucket repositories in our data set. Following Cito et al. [8], we determine the programming language of the repository as marked by the git repository hosting platform `languages` field. We also use their same regular expression to extract the base image from each `FROM` statement in each analyzed Dockerfile. For the image size and architecture analysis, we use the `full_size` and `architecture` fields we recorded for all versions (tags) of the 3,364,529 images we obtain from Docker Hub.

To study evolution trends, we need historic data about the state of each characteristic in each year. For characteristics related to actual Dockerfiles, such as the used base images and Ubuntu versions, we can retrieve this historic data from the git history. For other characteristics such as image size or image architecture, Docker Hub does not store an exact evolution history. However, as described in Section II, images on Docker Hub can have versions or tags. Thus, we use these versions/tags to provide us historic information. We consider each version/tag of an image as an image available on the year corresponding to its update date. We intentionally consider each version/tag of an image as a standalone image since an image may have tags with multiple architectures for example. The only characteristic for which we have no historic information is the programming language of each repository. We approximate its evolution by placing a repository in a specific year based on its last commit date.

c) Results: We now discuss the five characteristics.

Programming languages: Figure 3 shows the evolution trend of the popular programming languages over time. We can see that the proportion of Shell projects declines dramatically, from about 48% in 2015 to around 20% in 2020. Nonetheless, it remains the most used language. In contrast, Python, JavaScript, and Go show an upward trend, suggesting that developers are now containerizing applications written in these languages more. The inherent characteristics and popularity of each programming language can interpret the trends we observe. Shell scripts are commonly used for setting environments and running programs so it is natural that they are commonly used in the related source files. Go is a popular solution for distributed systems and parallel computing while JavaScript is widespread in web applications and blockchain tools, which are inherently distributed and benefit from containerization. Based on IEEE Spectrum data [36], Python is already the most popular programming language by far. Given the large differences between Python versions, containerizing the corresponding applications seems like good practice. Interestingly, we can see that TypeScript, which does not appear in the top 15 languages presented by Cito et al. in 2016 [8], has slowly started gaining popularity between 2017 and 2020. Such changes confirm the need to study evolution trends and understand the current state of the Docker ecosystem.

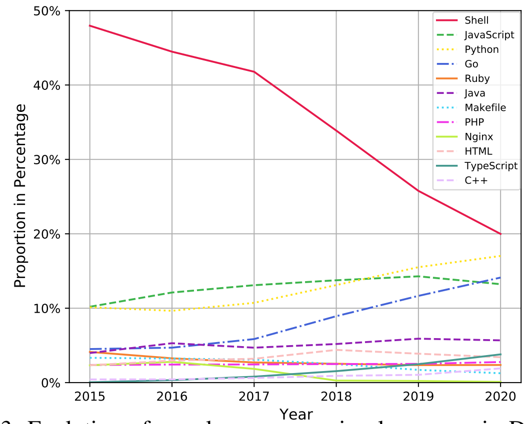

Fig. 3: Evolution of popular programming languages in Docker image source repositories.

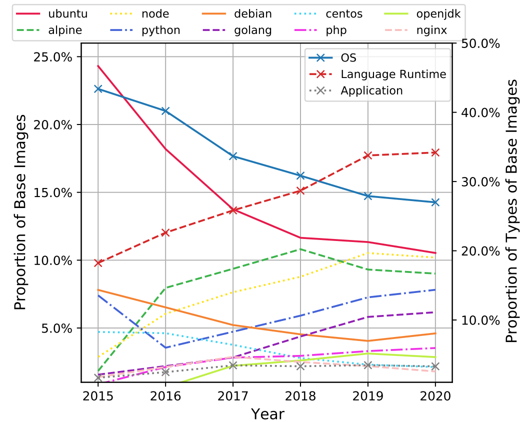

Fig. 4: Evolution of popular base images used by studied images.

Base images: Figure 4 shows the evolution of popular base images over time. The proportion of Docker images using Ubuntu as the base image is declining gradually, and the downward trend also holds for some of the other OS images, including Debian and CentOS. Alpine is the only OS base image with an upward trend in popularity in recent years. Interestingly, one of the observations, and recommendations, by Cito et al. [8] is the need to use smaller base images (particularly Alpine) to align with the goal of containerization in reducing the footprint of virtualization. The official Docker documentation also recommends using Alpine as the base image [26]. It seems that with the maturity of the Docker ecosystem, this recommended practice is now being followed.

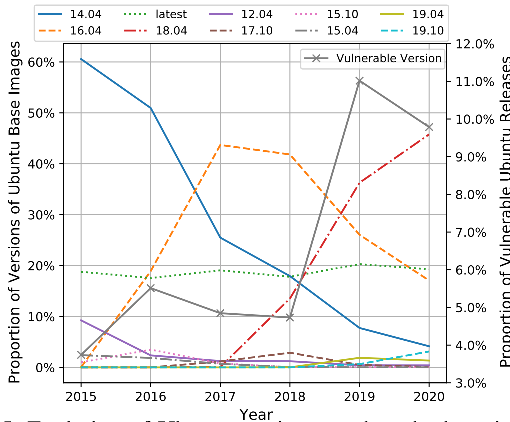

Fig. 5: Evolution of Ubuntu versions used as the base images.

Figure 4 also shows that the proportion of the different types of base images on the right y-axis. We can see that language run-time base images (e.g., Python or Golang) are growing fast. This is consistent with our above findings that Python and Go are among the most popular languages. The results are also consistent with the language trend results where JavaScript, Python, and Golang are gaining popularity over time. Given the various trends shown in the figure, we can conclude that while Ubuntu is still the most used base image, there is a clear upward evolution in language run-time images. We find that OS base images are generally declining in popularity, while language run-time and application base images are gaining popularity, indicating that developers may be becoming more inclined to build and execute programs based on ready-to-use language and application images, rather than configure the run-time environment from scratch based on an OS image.

OS versions: Despite its decrease in popularity over the years, Ubuntu remains the most commonly-used base image, so we further investigate the versions of Ubuntu used by image developers. Figure 5 shows the evolution of used Ubuntu versions. We can see that around 20% of the images each year use the `latest` Ubuntu tag, which points to the latest release of Ubuntu with long-term support (LTS). At the time of writing this paper, `latest` pointed to 20.04. We can see that each Ubuntu LTS version (14.04, 16.04, 18.04, and 20.04) becomes predominant in the following two years after its release. For instance, Ubuntu 16.04, released in April 2016, was the most popular release version used as the base image during 2017 and 2018. Such a two-year period follows the two-year release cycle of Ubuntu LTS versions [37].

Ubuntu releases have five years of life-cycle for LTS versions, and nine months for non-LTS releases in most cases [37]. Ubuntu distributions that reach the end of standard support date tend to have many outdated packages, and do not get any official security and maintenance updates for non-subscribed users. Hence, the security and reliability of containerized applications that use obsolete OS base images could be compromised. Our results on the right y-axis of Figure 5 show that there is a surge in the proportion of vulnerable Ubuntu releases, from 5% to 11%, as Ubuntu 14.04 LTS reached its end of life in early 2019. We anticipate that there will be another surge in 2021 when Ubuntu 16.04 LTS reaches its end of life. Image developers should check the version and life-cycle of OS/Application base images they rely on. Ideally, tools that automatically warn developers about using obsolete Ubuntu versions in their images can also help.

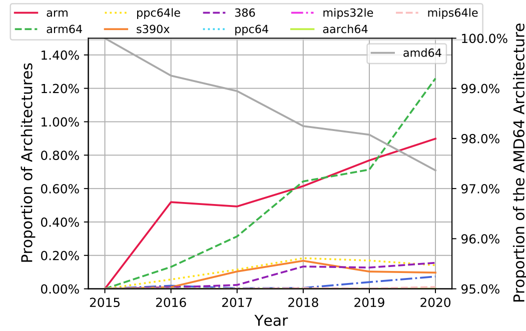

Fig. 6: Architectures used by studied Docker images over time.

Image Architectures: Figure 6 shows the evolution of architectures used by Docker images. The right y-axis shows that while its proportion is decreasing over time, AMD64 (x86-64) is still by far dominant (over 97%) in the Docker ecosystem. The share of ARM is growing fastest among all non-AMD64 architectures, which suggests the need to build tools and infrastructures for ARM-based Docker containers.

Image Sizes: Figure 7 shows the evolution of the Docker image sizes over the past five years. Note that larger images need longer time to deploy and distribute, and may incur security and dependability issues. The figure illustrates a downward trend in the median Docker image size from 2015 to 2020, suggesting that Docker images are becoming smaller and less complex. We also perform linear trend analysis on the size of images with five or more versions (tags) and find that 67.61% (39.63%, p-value $\leq$ 0.05) of images decrease in size, while 31.07% (12.24%, p-value $\leq$ 0.05) of images become larger. Such a trend indicates that developers are following the Docker development best practices [15] to slim down the image to make it lightweight.

Characterization of the Docker ecosystem. Python, Go, and JavaScript, are increasing in popularity as the languages of the containerized applications. While declining, Ubuntu is still the most used base image, but obsolete Ubuntu base images are increasingly being used, which may pose security risks. Overall, there are more images relying on language run-time or application base images, rather than directly on OS ones. ARM is also the fastest growing non-AMD64 image architecture. Image sizes are decreasing over time, suggesting best practices are being followed.

B. *RQ2: What are the current image tagging practices?*

a) Motivation: Docker provides a tagging mechanism for ease of versioned Docker image development and management. *Release tagging* and *SHA hash pinning* are two official tagging/versioning conventions [17], [19]. In release tagging, developers reuse tag names and make them always point

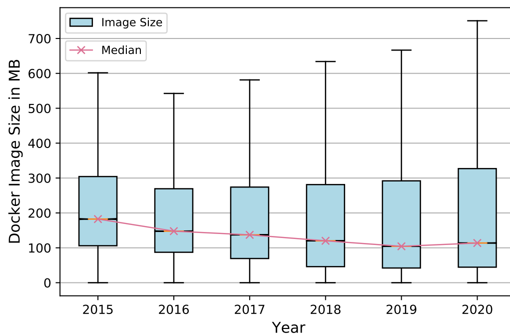

Fig. 7: The distribution of Docker image sizes in MB over time

to stable releases/branches in the source repository. While release tagging can provide semantic versioning of images and is easily integrated with verification processes, it does not provide a roll-back mechanism and incurs uncertainties due to potential updates in the associated branch overwriting the version the tag points to. On the other hand, SHA pinning uses the digest/commit SHA as unique tag names. This guarantees that a tag and the corresponding content will never change, and rolling back to previous versions is possible. However, a SHA hash is not human-readable, and SHA pinning requires more involvement in verification processes since specific SHAs, as opposed to branch names, need to be specified [17]. Since each tagging/naming convention has its pros and cons and the choice of which convention to use may depend on factors such as the DevOps process used by developers, we do not claim that one convention is necessarily better than another. Instead, we simply investigate which convention is used more.

Regardless of the tagging mechanism used, a specific version of a Docker image can be referred to with *image name:tag name*. When the tag name is not specified, the default tag name *latest* is fetched, which most Dockerfile developers often expect to point to the latest version of the desired Docker image. However, the `latest` tag is error-prone and may pose a threat to the quality of containerized applications due to its uncertainty problem [15]–[18]. If a Docker image only has the `latest` tag, then it is true that this `latest` tag always points to the latest version/release of the image (i.e., any updates in the built image will overwrite the latest version). However, if the Docker image uses multiple tags for versioned development, the `latest` tag does not necessarily point to the latest release of an image; the image developer must manually ensure that it does. Thus, if they forget to update the version that `latest` points to, it may actually point to an old version. This is what we refer to as the *version/time lag problem*. Worse, if a tag name is not specified and the latest tag is not explicitly set up, no corresponding image will be found. We evaluate how prevalent/serious these problems are in practice.

b) Method: The `tag` table in our database contains the available tags of 3,364,529 Docker images. We use the recorded `tag_name` field for all available tags, and the `last_updated` field, which stores the update time of the image each tag points to. To identify if SHA pinning is used, we use a regular expression to identify tags named after a SHA hash for all 3,364,529 images in the data set. To check if release tagging is used, we analyze the corresponding repository information for the 440,057 images with associated source repositories. We compare their image tags with the branches/releases/tags in their corresponding repository. For the analysis of `latest`, we first check if an image has a `latest` tag or not. This determines the prevalence of the problem where no image will be fetched in the first place if a tag name is not specified. For images with a `latest` tag, we then check if the tag does point to the latest release or not.

c) Results: We find that 65,509 out of 440,057 images with associated source repositories use release tagging, where their image tags match releases/branches in their source repositories. Among all 3,364,529 images in the data set, only 35,136 use SHA pinning. We also analyze the tagging convention of the top 50 popular images on Docker Hub and find that all of them use semantic tagging and have concise and human-readable names. Our results suggest that release tagging may be favored by more developers, compared to SHA pinning.

With respect to the `latest` tag, we find that 50.48% of the analyzed Docker images *only* have the `latest` tag. These images are not following recommended versioning practices [15]. Similar to pinning versions to ensure backward compatibility in package management tools, such as pip or Aptitude, image developers should always make images versioned, and avoid using only the default latest tag in production. Since, in this case, there is always only one available version of the Docker image, the `latest` tag always points to that latest version without any version tracking. This means that a containerized application based on a previous version of the image may fail due to newly added layers in the new version of the image.

Since images with only one `latest` tag do not follow versioning practices, we now focus only on 1,479,833 images with at least one self-defined tag. We find that 70.49% of these images do not have a *latest* tag. In this case, a valid tag name must be specified when using the image; otherwise, Docker will not find the image and will report an error. To avoid this, image developers should make the *latest* tag available and always pointing to the latest stable image version.

To investigate the version/time lag problem, we focus on 757,349 Docker images that have multiple self-defined tags. Among these, 11.39% have the *latest* tag but the tag does not actually point to the latest version of the image. Worse, 3.94% have a large lag, where there are at least five released versions beyond the version *latest* points to. We also observe a large gap in the update time: 4.41% of images have *latest* pointing to an image that is updated more than three months before the current most recent version of the image. Since many developers rely on the *latest* tag to get the most recent version of the image, the above discrepancies mean that a wrong version of the image will be fetched, since the *latest* tag does not point to the real latest version of the image.

Tagging practices. Semantic release tagging of Docker images is more commonly used over SHA pinning. About half of the Docker images have only one default *latest* tag, while $\sim 11.39\%$ have an outdated *latest* tag.

C. RQ3: How do Dockerfiles co-evolve with source code?

a) Motivation: RQ3 is inspired by one of the future research questions proposed by Schermann et al. [9]. In most cases, adding a new feature to the application would involve modifying source files, but would probably not involve the need to change the run-time environment (i.e., the Dockerfile). This different underlying nature of a Dockerfile and the application built on it may lead to differences in evolution rates and scales. While Cito et al. [8] investigated the evolution of Dockerfiles, they focused on the number of revisions per year and on types of changes solely in Dockerfiles, not in terms of co-evolution with source code. Since our evolution units and scope are different here, we cannot directly compare results.

b) Method: To investigate the co-evolution of Dockerfiles and source code, we use the data of all 378,615 Docker image source repositories. These repositories correspond to 1,920,195 Dockerfiles in the `dockerfile` table, 23,376,349 commits in the `commit` table, and 124,192,028 changed files recorded in the `changedfile` table. To analyze the scale of evolution, we compare the numbers and types of line-level changes in both Dockerfiles and other source files.

To investigate co-evolution, we calculate the ratio between the total number of commits in a repository and the number of commits involving Dockerfiles. For instance, a ratio of 3 means that, on average, the Dockerfile gets a revision every three commits, indicating a slower evolution rate of the Dockerfile. For more meaningful evolution rates, we consider only the source repositories of influential images with image pull count greater than 1,000. We divide commits in the repository history into six groups according to their commit year. We then obtain the distribution of the Dockerfile evolution yearly.

c) Results: Figure 8 shows the distribution of line changes in Dockerfiles and other source files in each commit. The median number of changed lines in Dockerfiles is 6.0 vs. 14.0 for the source code, indicating that Dockerfiles evolve at a smaller scale compared with other source code. The smaller evolution scale also holds for added (4.0 vs 8.0) and deleted (1.0 vs 3.0) lines of Dockerfiles compared to source code.

Figure 9 shows difference in evolution rate between Dockerfiles and source files. We can see that the median ratio increases steadily over time, from 2.5 in 2015 to 4.0 in 2020, which means that not only do Dockerfiles evolve at a slower rate than other source code, but that this evolution rate continues to slow down. We conjecture that this may be due to the increasing maturity of the Docker ecosystem. When Docker was just launched, there may not have been enough dependable Docker images that can be used as base images. Developers had to compose Dockerfiles from scratch to create Docker images suitable for their needs. This likely meant that they needed to make more modifications to their Dockerfiles to reach their desired environment. As the Docker ecosystem becomes more mature, many certified companies and organizations publish their official images on Docker Hub, which usually have high quality and provide many required features. As is evident from RQ1, more and more ready-to-use language run-time images and application images are available and used by more developers. Meanwhile, more and more public Docker images maintained by other developers emerge. Developers can simply choose an appropriate Docker image from Docker Hub to use as the base image for building their application. Hence, less Dockerfile changes may be needed, explaining the decrease in the evolution rate.

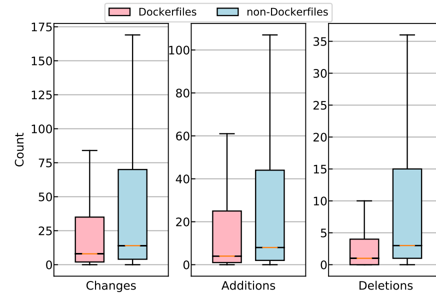

Fig. 8: The distribution of changed line count of Dockerfiles and non-Dockerfiles (other source files).

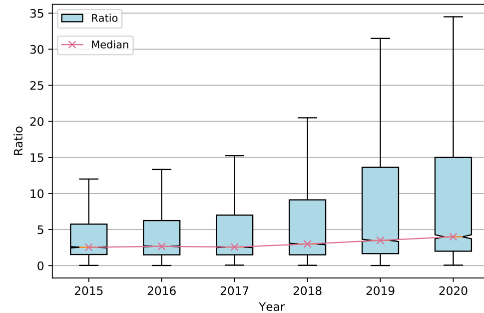

Fig. 9: The distribution of the ratio between the source repository commit count and Dockerfile commit count over time.

Co-evolution of Dockerfiles & Source Code. Dockerfiles evolve at a slower rate and a smaller scale than other source code. The evolution rate of Dockerfiles, when compared to other source code, continues to slow down.

D. RQ4: How prevalent are code smells in Dockerfiles?

a) Motivation: Docker provides a set of best practices for writing Dockerfiles [26], [38]. These practices can help developers avoid common mistakes, write less error-prone Dockerfiles, and build efficient images. Understanding prevalent code smells can help tool builders develop tools to avoid them as well as educate developers to pay attention to them. Previous work by Cito et al. [8] studied such smells and found prevalent violations. We investigate if code smells are still prevalent in the Docker ecosystem, and study their evolution.

b) Method: We use the Haskell Dockerfile Linter [39], a static analysis tool that parses Dockerfiles and checks for 66 types of best-practice violations, such as missing version pinning while installing packages, uncleaned cache, and using deprecated instructions. We show some of these smells in Table I. To identify evolution trends for years 2015 to 2020, we use the last updated version of the Dockerfile in that year (similar to RQ1). For each Dockerfile in each snapshot, we execute the linter and collect the reported violations.

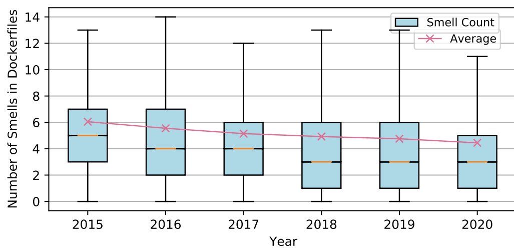

Fig. 10: The distribution of smell count in Dockerfiles over time.

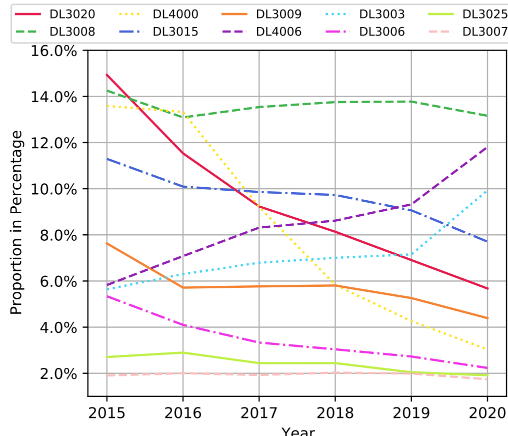

Fig. 11: Evolution of prevalent Dockerfile smells in popular images.

c) Results: We find that the average smell count is 5.13, the median is 4.0, with the 25th quantile at 2.0, and the 75th quantile at 6.0. Only 7.78% of the studied Dockerfiles are code-smell free. Figure 10 shows the evolution of smell count over time, where we observe a downward trend. This suggests that the quality of Dockerfiles is getting better w.r.t number of code smells; it could be because as the community grows, more developers master writing less error-prone Dockerfiles.

We also investigate the evolution of prevalent code smells in Dockerfiles. As smells in Dockerfiles of unpopular images may not necessarily be cause for alarm, we specifically focus on influential images whose image pull count is greater than 1,000. Figure 11 shows the proportion of the most prevalent code smells in these Dockerfiles over time. The definition and overall proportion of these smells are described in Table I. Comparing to Cito et al. [8], we find that DL3008 is still prevalent, while DL3020, DL3009, DL3006 are no longer as prevalent as before. Instead, DL4006 and DL3003 became more prevalent. We now discuss these smells.

Overall, DL3008 is the most frequently violated rule, which is caused by missing version pinning in Aptitude's installation command `apt install`. Installing packages without pinning a specific version can easily lead to compatibility issues, since an unanticipated version of the package may be retrieved based on the package management system's cache.

DL3020 tracks incorrect usage of `ADD` vs. `COPY`, and accounts for 9.04% of all violations. While both commands are similar, the best practice is using `ADD` for extracting a local tar file into a container while using `COPY` to copy other files and directories that are not packed with tar. The good news is that Figure 11 shows that DL3020 is quickly declining

TABLE I: The definition and overall proportion of most frequent smells in Dockerfiles of influential images shown in Figure 11.

Rule	Percentage	Definition
DL3008	13.60%	miss version pinning in apt install
DL3015	9.51%	miss –no-install-recommends
DL3020	9.04%	incorrectly use ADD instead of COPY
DL4006	8.31%	not using -o pipefail before RUN
DL4000	7.65%	deprecated MAINTAINER
DL3003	6.97%	incorrectly use cd instead of WORKDIR
DL3009	5.64%	not cleaning apt cache
DL3006	3.30%	untagged version of the image
DL3025	2.33%	not using JSON notation for CMD
DL3007	1.90%	using the error-prone latest tag

over time. `cd` and `WORKDIR` are also similar commands; `cd` should be used only in subshells, while `WORKDIR` should be used to change the directory because it can automatically create and change the working directory while avoiding any inconsistencies. Even though DL3003 accounts for less smells than DL3020 (6.97%), Figure 11 shows that it grows over time.

DL4006 reports if a Dockerfile does not catch pipe errors and shows a clear upward trend. During the build stage, the Docker engine leverages the `/bin/sh -c` interpreter to execute commands specified by *run*. The `-c` option makes bash determine success by evaluating only the exit code of the last operation in the pipe, meaning that the image build will still succeed even if errors happen in an interim stage. To ensure unexpected errors at any stage cause the image build to fail, developers should pass the `-o pipefail` option to the bash interpreter before executing any *run* commands.

DL3009 relates to not cleaning the cache of package management tools while DL3015 is about installing additional packages that developers do not explicitly need. As we discussed in RQ1, one of the best practices of developing Docker images is to keep Docker images clean and small. These two smells lead to space wastage, and are very likely to cause build failure and expose more attack surface [12]. Developers should always add statements in the Dockerfile to clean the cache and avoid installing unnecessary packages. The downward trend of both DL3009 and DL3015 again demonstrates that the best practice of keeping image small is being adopted more.

DL3006 reports when a tag is not specified when referring to an image and appears in the frequent smells in Table I. As discussed in RQ3, in this case, the default tag name *latest* will be used, which may lead to multiple problems.

Dockerfile Smells. The overall prevalence of code smells causing larger image sizes is declining over time, while not catching the pipe error and incorrectly using the `cd` command became more prevalent. However, overall, the quality of Dockerfiles is improving w.r.t code smells.

VI. Discussion

Studying the Docker ecosystem using a more comprehensive data set that is based on both Docker Hub and GitHub/Bitbucket allowed us to reveal prevalent (and often problematic) practices, as well as their evolution trends. There are several implications from our work for relevant practitioners as well as the Docker engine developers.

a) For Docker image developers & technology providers: 1) Always check the version and life-cycle of your dependencies, and avoid using **obsolete base images** (RQ1). 2) Be careful about the **error-prone latest tag** (RQ2), making sure it points to the latest release version of the image. 3) Continue to follow the practices of **using lightweight base images and keeping image size small** (RQ1). 4) Avoid **prevalent code smells in Dockerfiles**, such as not catching pipe errors or not pinning versions (RQ4). 5) Our results unveil the **increasing choice of using ready-to-use language run-time and application base images** (RQ1), suggesting that organizations who want to help developers adopt their technologies should consider creating ready-to-use base images.

b) For Docker & infrastructure providers: 1) Add support and build tools for **fast-growing ARM-based containers** (RQ1). 2) Provide an automated mechanism to remind image developers to avoid the **error-prone *latest* tag** when pushing new image versions (RQ2). 3) Similar to GitHub's new vulnerability scanners for third-party dependencies [40], Docker Hub could include a base image version checker as part of its CI service, to mitigate **vulnerabilities due to obsolete base images** (RQ1) 4) Container registry and Container-as-a-Service providers can cache layers of **popular base images** to increase the speed of online image builds (RQ1).

VII. Threats to Validity

a) Construct Validity: In RQ2, version/time lags related to the *latest* tag could be due to release practices, such as alpha, beta, or canary releases. For example, a team releases a new image version and monitors how a sub-population of users reacts to it before updating the *latest* tag to that release. As such DevOps processes are unknown to outsiders, we do not try to differentiate these practices but instead report the current state, which may indicate potential problems. In RQ4, smells are not the only/full measure of Dockerfile quality; we use smells as one potential proxy for quality here. When studying trends of programming languages in RQ1, we do not have snapshots of the languages used by repositories at each year, but instead have a single snapshot in May 2020. We approximate evolution by placing a repository in a corresponding year, based on its last commit date. Since a source repository that got its last update in a certain year suggests that the image is active/used in that year, our trends can correctly reflect the programming language evolution we studied. We do not have this problem in the other evolution trends since we have historic data to base evolution on.

b) Internal Validity: There could be duplicate Docker images hosted on Docker Hub, but such proportion is unknown and is not straightforward to find. In RQ2, we check if the release tagging convention is followed for only 440,057 images that have the linked source repository information in the data set; thus, the actual number of images using release tagging could be even larger than reported here. We use GitHub's language detection to analyze the popular programming languages of source files in RQ1, which relies on matching file extensions and may contain inaccuracies. Finally, we use the Haskell Dockerfile Linter to detect smells, and the tool itself might have inaccuracies, such as false positives.

c) External validity: To collect data from source repositories of images hosted on Docker Hub, we relied on the CI link and our name matching heuristic. This limits the generalizability of our study since out of 3,364,529 images from Docker Hub, we could accurately collect the source repository information of only 440,057 images (which is still by far larger than previous studies). Some of our results may therefore not apply to images that do not use the CI service. Besides, we cannot eliminate all toy projects. However, as mentioned in Section IV, images using CI typically have a higher pull count (median 98.0) than images not using CI (median 18.0). Hence, we believe that images using CI are more influential and more likely to be non-toy projects; an empirical analysis based on them can be more reliable.

Our data set covers 98.38% of public images hosted on Docker Hub, as of May 3, 2020. For reasons we could not identify, the Docker Hub API does not return the remaining 1.62%. Given the small proportion, we do not expect any impact on our results. We studied only public images hosted on Docker Hub. To the best of our knowledge, there is no way to get information about private images. While limiting, we do not foresee major differences between other image registries or public/private images that may impact our results.

VIII. Conclusions

Docker is currently the most popular containerization solution. Given this popularity and its resulting impact, understanding evolution trends and quality issues in the Docker ecosystem is important. In this paper, we created a new large-scale and comprehensive data set containing information about 3,364,529 Docker images hosted on Docker Hub and 378,615 source repositories behind them. Based on this data set, we performed a large-scale empirical study related to various aspects of Docker images and their evolution over time.

Our results reveal some problems with the Docker images. Specifically, we find incorrect and problematic usage of the `latest` tag, lack of use of image versioning altogether, and usage of obsolete base OS images. The good news is that many of these problems can be fixed with simple tooling, which can even be integrated into Docker Hub's existing CI service. With the exception of the above problems, our findings generally show a positive and healthy evolution of the Docker ecosystem. For example, we find a downward evolution trend in Docker image sizes over the past few years. Our results also show that (the recommended base OS image by Docker) Alpine is the only OS base image with a significant upward trend in popularity in recent years. We also find that developers are increasingly favoring lightweight and ready-to-use language run-time and application base images, suggesting the maturity of the ecosystem and the adoption of more best practices. Finally, we find that the number of smells found in Dockerfiles is decreasing over time, suggesting the overall improvement of quality of these images. We provide implications of all our results for various types of practitioners.

REFERENCES

[1] N. Kratzke, “A brief history of cloud application architectures,” *Applied Sciences*, vol. 8, no. 8, p. 1368, 2018.

[2] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, “Cloud container technologies: a state-of-the-art review,” *IEEE Transactions on Cloud Computing*, 2017.

[3] H. Lee, K. Satyam, and G. Fox, “Evaluation of production serverless computing environments,” in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 442–450.

[4] V. Singh and S. K. Peddoju, “Container-based microservice architecture for cloud applications,” in *2017 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2017, pp. 847–852.

[5] “Docker,” https://www.docker.com/, 2020, [Online; accessed April-1-2020].

[6] H. Zhu and I. Bayley, “If docker is the answer, what is the question?” in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2018, pp. 152–163.

[7] “Get started, part 1: Orientation and setup — docker documentation,” https://docs.docker.com/get-started/, 2020, [Online; accessed May-1-2020].

[8] 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 *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 323–333.

[9] G. Schermann, S. Zumberi, and J. Cito, “Structured information on state and evolution of dockerfiles on github,” in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 26–29.

[10] B. Tak, H. Kim, S. Suneja, C. Isci, and P. Kudva, “Security analysis of container images using cloud analytics framework,” in *International Conference on Web Services*. Springer, 2018, pp. 116–133.

[11] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, “A dataset of dockerfiles,” in *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. IEEE, 2020.

[12] ——, “Learning from, understanding, and supporting devops artifacts for docker,” in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[13] T. Xu and D. Marinov, “Mining container image repositories for software configuration and beyond,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 2018, pp. 49–52.

[14] “Docker hub quickstart,” https://docs.docker.com/docker-hub/, 2020, [Online; accessed April-8-2020].

[15] “Docker development best practices,” https://docs.docker.com/develop/dev-best-practices/, 2020, [Online; accessed May-6-2020].

[16] “Intro guide to dockerfile best practices,” https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/, 2020, [Online; accessed May-6-2020].

[17] “Images: Tagging vs digests,” https://success.docker.com/article/images-tagging-vs-digests, 2020, [Online; accessed May-6-2020].

[18] “What’s wrong with the docker :latest tag?” https://vsupalov.com/docker-latest-tag/, 2019, [Online; accessed May-2-2020].

[19] “Image tag best practices - azure container registry — microsoft docs,” https://docs.microsoft.com/en-us/azure/container-registry/container-registry-image-tag-version, 2019, [Online; accessed April-24-2020].

[20] B. Adams, “Co-evolution of source code and the build system,” in *IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2009, pp. 461–464.

[21] S. McIntosh, B. Adams, and A. E. Hassan, “The evolution of ANT build systems,” in *7th IEEE Working Conference on Mining Software Repositories (MSR)*, May 2010, pp. 42–51.

[22] C. Lin, S. Nadi, and H. Khazaei, “Artifact repository,” https://github.com/linncy/icsme2020-docker-study, [Online; accessed August-24-2020].

[23] “Docker hub,” https://hub.docker.com/, 2020, [Online; accessed May-3-2020].

[24] “Set up automated builds — docker documentation,” https://docs.docker.com/docker-hub/builds/, 2020, [Online; accessed April-9-2020].

[25] R. Shu, X. Gu, and W. Enck, “A study of security vulnerabilities on docker hub,” in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 269–280.

[26] “Best practices for writing dockerfiles,” https://docs.docker.com/develop/develop-images/dockerfile_best-practices, 2020, [Online; accessed May-4-2020].

[27] K. Morris, *Infrastructure as code: managing servers in the cloud*. ” O’Reilly Media, Inc.”, 2016.

[28] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, “The making of cloud applications: An empirical study on software development for the cloud,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 393–403.

[29] Y. Jiang and B. Adams, “Co-evolution of infrastructure and source code: An empirical study,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 45–55.

[30] T. Sharma, M. Fragkoulis, and D. Spinellis, “Does your configuration code smell?” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 189–200.

[31] A. Rahman, C. Parnin, and L. Williams, “The seven sins: Security smells in infrastructure as code scripts,” in *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[32] “Http api v2 — docker documentation,” https://docs.docker.com/registry/spec/api/, 2020, [Online; accessed May-1-2020].

[33] “Docker hub search engine,” https://hub.docker.com/search/, 2020, [Online; accessed: May-1-2020].

[34] “Github api,” https://api.github.com/, 2020, [Online; accessed May-1-2020].

[35] “Bitbucket api,” https://developer.atlassian.com/bitbucket/api/2/reference/, 2020, [Online; accessed May-1-2020].

[36] “The top programming languages 2019,” https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019, 2019, [Online; accessed April-4-2020].

[37] “Ubuntu release cycle — ubuntu,” https://ubuntu.com/about/release-cycle, 2019, [Online; accessed June-1-2019].

[38] “Dockerfile reference,” https://docs.docker.com/engine/reference/builder/, 2020, [Online; accessed May-9-2020].

[39] “Haskell dockerfile linter,” https://github.com/hadolint/hadolint, 2020, [Online; accessed May-4-2020].

[40] “About security alerts for vulnerable dependencies,” https://help.github.com/en/github/managing-security-vulnerabilities/about-security-alerts-for-vulnerable-dependencies#alerts-and-automated-security-updates-for-vulnerable-dependencies, 2020, [Online; accessed May-16-2020].