Should you Upgrade Official Docker Hub Images in Production Environments?

Sara Gholami[1], Hamzeh Khazaei[2], Cor-Paul Bezemer[1]

***Abstract*—Docker, one of the most popular software containerization technologies, allows a user to deploy Docker images to create and run containers. While Docker images facilitate the deployment and in-place upgrading of an application in a production environment by replacing its container with one based on a newer image, many dependencies could change at once during such an image upgrade, which can potentially be a source of risk. In this paper, we study the official Docker images on Docker Hub and explore how packages are changing in these images. We found that the number of package changes varies across different types of applications and that often the changing packages are utility packages. Our study takes a first important look at potential risks when doing an in-place upgrade of a Docker image.**

***Index Terms*—Dependency upgrades, Downgrades, Docker, Docker Hub, Containerization**

I. Introduction

Containerization is a popular approach to deploy software systems [6]. One of the enabling technologies for containerization is Docker, an open-source framework to deploy containers in different computing environments [2]. Docker containers are composed of an image that encapsulates software code and all its required package and library dependencies [6]. As a result, deploying Docker containers into a production environment and applying in-place upgrades by replacing them with containers created from newer images is easy. However, with every upgrade of a Docker image, many packages could change at once, which could e.g., result in reduced performance or security of the application.

Several related work studied the Docker images available on Docker Hub from a security point of view [13], [16], [17]. For instance, Shu et al. [13] studied over 300,000 Docker images for the spread of vulnerabilities from one image to another image that uses it. They found that images inherit security vulnerabilities from their parent image. Similarly, Zerouali et al. [16] showed that vulnerabilities in *npm* packages might impact Docker images.

Besides the potential security risks in Docker images, there is another potential yet unexplored risk: the risk of changing many components of a functional system at once. The risk of package changes was studied in different environments and languages such as *Node.js*, and *Java* [8], [9], [12], [15]. These studies show that package changes can lead to broken functionality, poor performance, or security vulnerabilities in the applications that depend on the packages. As a result, although applying in-place upgrades on Docker images is easy, it can put the whole system at risk through issues that are caused by internal packages.

We study the package changes in official Docker images (images that are reviewed by the Docker team) in the Docker Hub registry. We focus on the native (operating system), *Node*, and *Python* packages and investigate which types of applications tend to have more package changes.

Our study is the first to show how frequent changes are happening in the packages used by different types of applications on Docker Hub. This important new view on the Docker ecosystem helps to raise awareness with developers of containerized software systems of the necessity of being cautious when upgrading Docker images in a production environment.

II. Docker and Docker Hub

Docker [5] is a container virtualization technology [1] that puts together several kernel-level technologies such as LXC and Cgroups to facilitate the deployment and use of containers. Docker provides interfaces to create and deploy containers. Docker containers are lightweight, packaged applications that can run on different computing environments without modification. Docker relies on two major components: the Docker Engine, which is the virtualization technology, and Docker Hub, a service for sharing Docker images [2].

Docker containers are created from Docker images by executing the `docker run` command. Docker Hub is where Docker images are stored by default, which means that by default, `docker push` uploads an image to Docker Hub and `docker pull` downloads an image from Docker Hub. To run a container, first, an image is pulled from Docker Hub by executing the `docker pull` command, and then by executing the `docker run` command, the image is used to create a container. Each image is created based on a DockerFile, a text file consisting of a series of commands to create an image. In the first line of a DockerFile, the operating system is specified, such as *Ubuntu*, meaning that the packages required for *Ubuntu* are added to the final image. Based on the application, different languages can be added to the DockerFile, such as *Python* and *Node.js*, which means that packages required for these languages will be added to the final image. In general, the used packages in an image are either packages that are native to a Linux distribution such as *debian*, *arch*, or *alpine*, or packages that are installed by popular package managers such as *PyPy*, *npm*, or *CRAN* [16].

Docker Hub [4] is Docker's default registry for finding and sharing container images. Currently, there are over 3

[1]Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Canada. E-mail: {sgholami, bezemer}@ualberta.ca

[2]Department of Electrical Engineering & Computer Science, York University, Toronto, Canada. E-mail: hkh@yorku.ca

million repositories in Docker Hub. Each repository is a collection of images, which allows users to share container images with other users, such as their team members or customers. Images in a repository are identified by unique user-identified tags. The following four types of repositories are available on Docker Hub:

1) *Community* repositories are maintained and delivered by community developers, including all users with a Docker Hub account. As a result, there is no guarantee on security, maintainability, or following best practices for development in these repositories. More than 99% of the Docker Hub repositories (over 3.3M) are community repositories.
2) *Verified* repositories are published and maintained by verified third-party publishers such as IBM or Microsoft [10]. There are 339 verified repositories on Docker Hub.
3) *Official* repositories are reviewed and published by a team that is sponsored by Docker Inc. Docker community members can contribute to developing the official images. Images in the official repositories exemplify DockerFile best practices and ensure that security updates are applied in a timely manner. There are 160 official repositories.
4) *Certified* repositories are a special type of verified repository that are built following best practices, tested and validated against the Docker Enterprise Edition platform and APIs, passed security requirements, and are collaboratively supported [10]. There exist 51 certified repositories on Docker Hub.

III. Methodology

Figure 1 displays the steps of our methodology for studying package changes in official Docker Hub images. We detail each step below.

Step 1: Selecting Repositories: We selected the Docker Hub official repositories as these 160 repositories are considerably more popular than the other types of repositories. Our analysis of the repositories shows that the official repositories have a median pull count of 10 million, while the community repositories have a median pull count of 45. Likewise, the median star count is 271 for the official repositories and 0 for the community repositories. In addition, as Docker Inc. sponsors a team to verify and publish the official repositories' content, users can be more confident about the credibility of the images in the official repositories. Therefore, we focused our study on the official repositories. We excluded the *Scratch* and *OpenSUSE* repositories as they did not contain images. Hence, we study 158 repositories.

Step 2: Collecting Image Tags: We retrieved the list of available tags for each repository using the code provided by source{d} [14] with a few modifications to retrieve all available tags. We collected a list of almost 37K tags from the official repositories.

Step 3: Collecting Packages and Latest Update Dates: We downloaded the images and analyzed their packages using the code provided by source{d} [14]. We focused on the native (used by the operating system), *Node* (for *Node.js* applications), and *Python* packages. All studied repositories contained native packages, six contained *Node* packages, six contained *Python* packages, and two contained both *Node* and *Python* packages. In addition, we collected the latest update date for each image by executing the docker inspect command.

Step 4: Identifying Package Changes: In the last step, we study the package changes when upgrading an image. We split the images in each repository into versioning branches, as images in different branches might use different packages, and comparing them would not be insightful. Figure 2 shows an example timeline of how branches could evolve in a repository. In this example repository, there is a branch with the *alpine* ending, which indicates those images are using *alpine*, a *Linux* distribution. As the branches may progress independently, we should not compare images from different branches. For example, we did not compare *1.1-alpine* to *1.1.1* as they are from different branches. After identifying the branches, we sorted the images in each branch based on their latest update date. However, there were cases in which several images were updated on the same day. In these cases, we manually sorted the images based on the versioning specified in the tags. As tag names do not follow any naming convention, we could not automate this process. Some repositories used the release date as their version number (e.g., *20200415*), and some used semantic versioning to indicate major, minor, and patch releases (e.g., *1.13.2*).

In addition, the tags in the *TomEE*, *NeuroDebian*, *ROS*, *BuildPack-Deps*, and *AdoptOpenJDK* repositories were not clearly dividable into branches. For example, in the *ROS* repository, all of the tags are names, such as *lunar-perception-stretch* and *melodic-perception*. Therefore, we excluded these five repositories from our analysis.

Finally, we compared the packages in each image with its adjacent image in the same branch to identify major, minor, or patch upgrades and downgrades. To determine if a change is an upgrade or downgrade, we compared the numbers and characters in the package versioning. For instance, a version change from *1.3.0* to *2.0* or from *1.3-a* to *1.3-b* is an upgrade. A change from version *2.1.0* to *2.0* or from *3.3-b* to *3.3-a* is a downgrade. In addition, we needed to determine if a change is major, minor, or patch. Based on the semantic versioning definition [11], major changes make incompatible API changes, minor changes add functionality in a backwards compatible manner, and patch changes make backwards compatible bug fixes. To identify each type of change, we separated the numbers in the version tags. If the first digits were different, then the change is major. If the second digits were different, it is a minor change. Otherwise, it is a patch change. For example, version *1.2.0* to *2.0* is a major change, while a change from version *1.2.0* to *1.3.1* is a minor change, and a version change from *1.2.0* to *1.2.1* is a patch change.

There are 14 categories of official repositories on Docker Hub (*Analytics, Application Frameworks, Application Infrastructure, Application Services, Base Images, Databases, DevOps Tools, Featured Images, Messaging Services, Monitoring,*

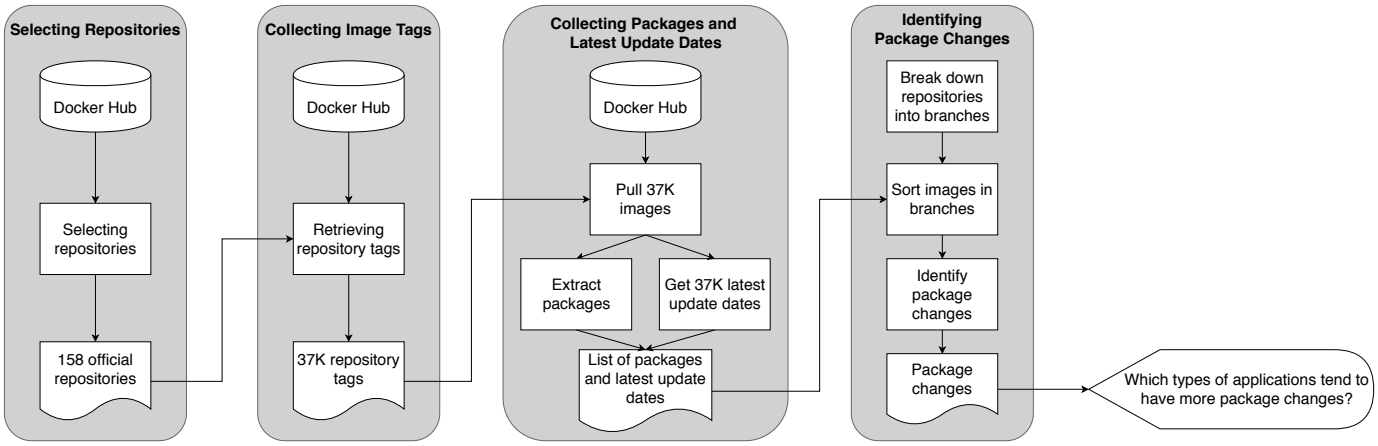

Fig. 1: Overview of our data collection process for our study on package changes in `Docker Hub` repositories

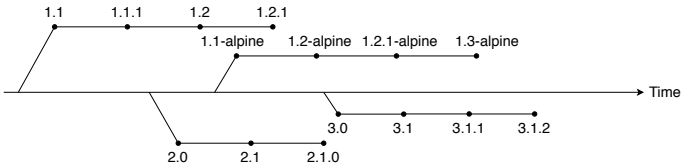

Fig. 2: An example of branches in a repository over time

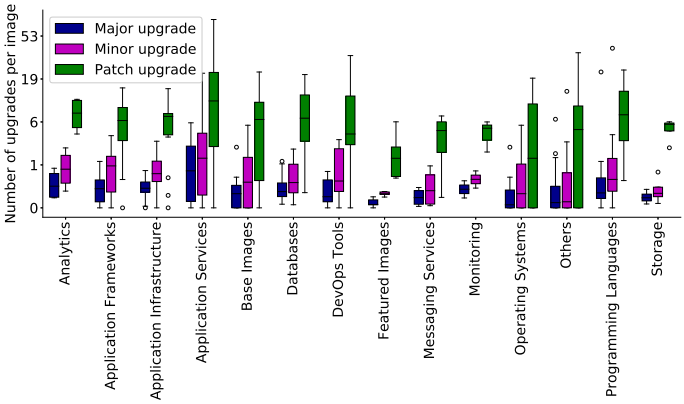

Fig. 3: Median number of upgrades in each category

Operating Systems, Programming Languages, and *Storage*). From the 153 studied official repositories, 115 belong to one or two of these categories. We categorized 38 official repositories that did not belong to any category as *Others*. We used these categories to compare the package changes in repositories of different categories.

IV. Results

There is a median of 8.6 upgrades per image across official `Docker` images. Figure 3 displays the distribution of major, minor, and patch upgrades in images of different categories. The *Application Services* applications have a median number of 1.4, 2.2, and 11.1 major, minor, and patch upgrades, respectively, which are the highest medians across categories. More specifically, in the *Application Services* category, the *ZNC* application has the highest number of upgrades (6.2 major upgrades, 21.9 minor upgrades, and 79.2 patch upgrades). The *Analytics* category has the second-highest median number of major (0.6) and patch (8.1) upgrades, and the third-highest median number of minor (1.5) upgrades. Afterward, the *Programming Languages*, *Application Infrastructure*, and *Databases* categories have the next highest median number of patch upgrades.

There is a median of 2.1 downgrades per image across official `Docker` images. Figure 4 illustrates the distribution of major, minor, and patch downgrades per image across different categories. The *Analytics* applications with 0.4, 0.8, and 3.8 have the highest median number of major, minor, and patch downgrades. The *Application Infrastructure*, *Application Services*, and *Programming Languages* categories have the second-highest median number of package downgrades in major, minor, and patch changes, respectively.

Images of *Analytics* applications are the least stable. The official images specify a median of up to 36 third-party packages. Figure 5 shows the median number of packages per image specified in each category. As can be seen, the *Operating Systems* and *Base Images* categories have the lowest median number of packages, which is why these applications also have the lowest median number of package changes in both upgrades and downgrades. The images in the *Operating Systems* and *Base Images* applications tend not to add many additional packages and provide the base operating system in an image. Although the images in the *Application Services* category have one of the highest median numbers of packages changes, the median number of packages used in these applications is not the highest. In contrast, applications in the *Analytics* category have the highest median number of package changes and total packages used in the images. This finding suggest that images for the *Analytics* applications are less stable than images for other types of applications.

The packages that are changed the most often are common utility packages. There are over 9K different packages used in the official `Docker` images. The ones with the most changes are utility packages such as *tzdata, base-files, libsystemd0, libudev1*, and *openssl*. In many cases, when upgrading a system, we do not want to upgrade utility packages

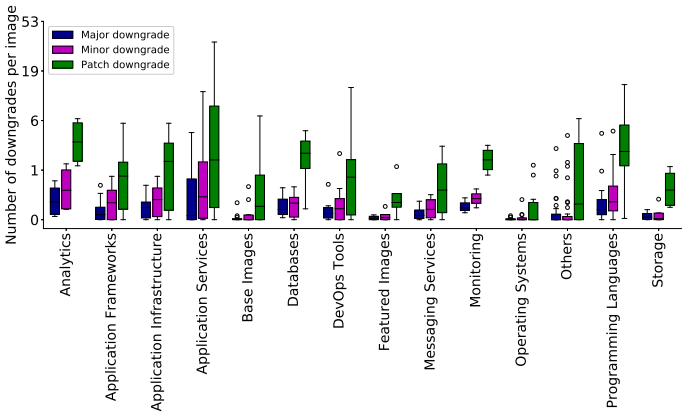

Fig. 4: Median number of downgrades in each category

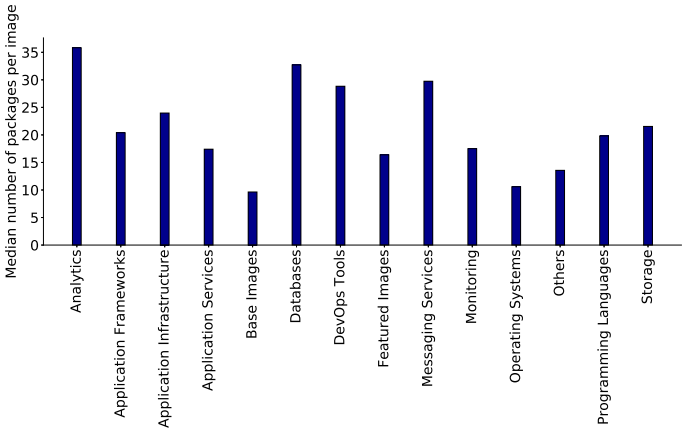

Fig. 5: Median number packages in each category

unless absolutely necessary, as such upgrades might cause incompatibilities. In addition, newer versions of these utility packages may contain bugs. Therefore, practitioners need to carefully check the packages which are changing in an image upgrade and consider the consequences on their system.

> ***Summary:*** *Practitioners need to be cautious when doing in-place upgrades of images from the official* `Docker Hub` *repositories as in all studied applications, many packages are changing.*

V. Related Work

Studies on Docker Hub: Most of the existing work on `Docker Hub` focuses on the security aspect of `Docker` images [13], [16], [17]. In addition, Ibrahim et al. [7] studied the similarity of `Docker` images on `Docker Hub` across repositories, to guide users when selecting a `Docker` image. Their study differs from ours as we focus on image upgrades. Zhao et al. [18] studied duplication of files in images on `Docker Hub` from the point of view of reducing the overall storage requirements of `Docker Hub`.

Package Updates: Prior studies on package dependencies focused on several environments (e.g., *Maven*) and languages (e.g., *Java* and *Node.js*). Cogo et al. [3] studied downgrades in *npm*. Kerzazi et al. [8] studied botched releases in an application for 1.5 years. Botched releases are releases that cause abnormal behaviors such as poor performance or crashes. Kerzazi et al. show that 22.5% of the releases are botched, which significantly affects depending systems. Raemaekers et al. [12] conducted a study on version changes of the jar files in *Maven* repository where about one-third of the major changes and one-third of the minor changes had at least one breaking change. Breaking changes are vital as they can have a significant impact on the client's software system and lead to compilation errors and crashes. In a study by Mezzetti et al. [9] on *Node.js* libraries, they found that 5% of the packages have been affected by breaking changes due to a minor or patch update in their dependencies. Xavier et al. [15] studied breaking changes in updates of 317 *Java* libraries, where 14.8% of changes caused incompatibilities with previous versions.

VI. Threats to Validity

Internal Validity: To sort the images in each repository, we first separated the images into possible branches. This process has been done manually as there is no concept of branch defined on `Docker Hub`. We did not include repositories in our study when we were not sure about the branches. Future studies should investigate automated approaches for identifying branches from version numbers. In addition, in our study we assume that the packages follow the semantic versioning principle.

External Validity: Future studies should investigate whether our findings hold for non-official `Docker` images. Also, while we found native packages in all of the images, only a few images used *Node* (6) and *Python* (6) packages. Future studies should analyze changes in `Docker` images for other types of packages (such as *R* packages).

VII. Conclusion

In this paper, we studied the official `Docker Hub` repositories and analyzed over 37K images in these repositories for their native (operating system), *Node*, and *Python* packages. Our study shows that a median of 8.6 packages are upgraded during an image update, which is high compared to a traditional update scenario in which only the target application is updated. Our study takes the first important step to studying package changes in `Docker` images. We encourage future studies to further investigate and quantify the risk with doing in-place image updates. To answer the question in the paper title: Should you upgrade official `Docker Hub` images in production environments? Probably not without thoroughly testing them first, or at least before investigating which packages are changing together with the main application.

References

[1] C. Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[2] T. Bui. Analysis of Docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[3] F. R. Cogo, G. A. Oliva, and A. E. Hassan. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, pages 1–15, 2019.

[4] Docker Inc. Docker hub. https://hub.docker.com/. Accessed: 2020-09-28.

[5] Docker Inc. Empowering app development for developers — docker. https://www.docker.com/. Accessed: 2020-09-28.

[6] IBM Cloud Education. Containerization. https://www.ibm.com/cloud/learn/containerization. Accessed: 2020-05-08.

[7] M. H. Ibrahim, M. Sayagh, and A. E. Hassan. Too many images on DockerHub! how different are images for the same system? *Empirical Software Engineering*, 25:4250–4281, Sep 2020.

[8] N. Kerzazi and B. Adams. Botched releases: Do we need to roll back? Empirical study on a commercial web app. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 574–583. IEEE, 2016.

[9] G. Mezzetti, A. Moller, and M. T. Torp. Type regression testing to detect breaking changes in Node.js libraries. In *Proceedings of the 32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[10] J. Morgan. Introducing the new Docker Hub. https://www.docker.com/blog/the-new-docker-hub/. Accessed: 2020-04-17.

[11] T. Preston-Werner. Semantic versioning 2.0.0. https://semver.org/. Accessed: 2020-04-24.

[12] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

[13] R. Shu, X. Gu, and W. Enck. A study of security vulnerabilities on Docker Hub. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy*, pages 269–280. ACM, 2017.

[14] source{d}. sourced datasets. https://github.com/src-d/datasets. Accessed: 2020-09-28.

[15] L. Xavier, A. Brito, A. Hora, and M. T. Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE, 2017.

[16] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona. On the impact of outdated and vulnerable JavaScript packages in Docker images. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623. IEEE, 2019.

[17] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona. On the relation between outdated Docker containers, severity vulnerabilities, and bugs. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–501. IEEE, 2019.

[18] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt. Large-scale analysis of the Docker Hub dataset. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, 2019.