DOCKERFINDER: Multi-attribute search of Docker images

Antonio Brogi, Davide Neri, and Jacopo Soldani
Department of Computer Science, University of Pisa, Italy
{name.surname}@di.unipi.it

***Abstract*—Docker containers run from Docker images, which can be distributed through so-called Docker registries. The currently available support for searching images in registries is however limited. Available registries (e.g., Docker Hub) only permit searching images "by name", i.e. by specifying a term occurring in the image name, in the image description or in the name of the user that created such image.**

In this paper we try to enhance the support for discovering Docker images by introducing DOCKERFINDER, a microservice-based prototype that permits searching for images based on multiple attributes, e.g., image name, image size, or supported software distributions. DOCKERFINDER crawls images from a remote Docker registry, it automatically analyses such images to produce multi-attribute descriptions to be stored in a local repository, and it permits searching for images by querying the local repository.

***Index Terms*—Docker, image search, microservices.**

I. INTRODUCTION

DevOps [11] practices aim at easing the collaboration and communication of software developers (Dev) and operators (Ops). Correspondingly, new technologies are emerging with the objective of supporting developers in building, testing, deploying and managing their software [13].

A notable example of such emerging technologies is Docker [3], a platform for building, shipping, and running applications, together with their dependencies, in lightweight virtual environments (called *containers*). Docker containers run from Docker *images*, which are the read-only templates used to create them.

A Docker image permits packaging a software component (e.g., the source code of an application, or its binaries) together with all software dependencies needed to run it. Images hence play a crucial role within Docker, and to ease images distribution Docker permits sharing them through so-called Docker *registries* (e.g., Docker Hub [5], which is the official registry for storing and retrieving Docker images).

As Docker registries may contain a huge amount of heterogeneous images [6], powerful search methods are needed to ease the search of images within registries. For instance, suppose that we wish to deploy an application component, and that such component requires certain software distributions to run (e.g., `python 2.7` and `java 1.8`). We would need a way to search for Docker images from which to run containers that actually support such software distributions.

Unfortunately, the current support for searching Docker images is limited, as Docker registries only permit looking for images "by name", i.e. by specifying a term, which is then exploited to return all images where such term occurs in the name, in the description or in the name of the user that built the image. As a consequence, users cannot specify more complex queries, e.g., by imposing requirements on the software distributions that an image must support (such as in the aforementioned example), on its size, or on "how much" it is recommended by the Docker community.

The main objective of this article is to try to overcome the aforementioned limitations, by proposing an enhanced discovery of Docker images. More precisely, we aim at permitting to search Docker images not only "by name", but also based on the software distributions they support, their size, and their popularity (viz., the amounts of pulls and of stars gained within Docker community).

Currently available search approaches are based on metadata that is either manually specified by the image developers (e.g., names and natural language descriptions of images), or automatically extracted from images by treating them as "black-boxes" (e.g., size of images, and "how much" they are recommended by the Docker community). Our approach instead relies on the very idea of automatically extracting the runtime features of images (e.g., the software distributions they support). To extract such features, we start containers from Docker images, and we directly check which features are actually supported in the containers' runtime environments.

To present our approach and to demonstrate its feasibility, we hereafter present the design and the implementation of DOCKERFINDER. DOCKERFINDER is a prototype that permits searching Docker images based on multiple attributes (e.g., image name, image size, supported software distributions). DOCKERFINDER crawls images from a Docker registry, and it automatically analyses them to produce multi-attribute descriptions. DOCKERFINDER stores the obtained image descriptions in a local repository, and it permits searching for images by submitting multi-attribute queries both through a remotely accessible API and through a graphical user interface.

DOCKERFINDER is designed by adopting the microservice-based architectural style [10], which is proven to improve scalability, manageability, and integrability of software systems [9]. The DOCKERFINDER prototype is then implemented and shipped as a multi-container Docker application [4].

It is worth noting that DOCKERFINDER is not intended to be an alternative to existing Docker registries, but it is rather a

novel approach for improving their image discovery capabilities. The inspection approach of DOCKERFINDER can indeed be directly exploited within Docker registries (e.g., Docker Hub [5]) to enrich the description of the images they contain.

The rest of the paper is organised as follows. Sect. II illustrates the architecture of DOCKERFINDER. Sect. III introduces the DOCKERFINDER prototype. Sects. IV and V discuss related work and draw some concluding remarks.

II. ARCHITECTURE OF DOCKERFINDER

Given a remote registry of Docker images, the objective of DOCKERFINDER is to permit searching such images not only "by name", but also based on additional information (e.g., the software distributions they support, their size, and their popularity). In this perspective, DOCKERFINDER is designed to provide the following three main functionalities:

- *Analysis* → DOCKERFINDER pulls and analyses each image in the registry it is connected to. The analysis of each image consists in retrieving all the metadata already available in the registry, and in running a container to automatically extract its runtime features (e.g., the software distributions it support). All collected information is used to build the multi-attribute description of an image.
- *Storage* → DOCKERFINDER stores all produced image descriptions in a local repository.
- *Discovery* → DOCKERFINDER allows users to search for images. Users can indeed submit multi-attribute queries to DOCKERFINDER, which are then evaluated with respect to the image descriptions stored in its local repository.

We choose to implement the above functionalities as a suite of interacting microservices [9], as the microservice-based architectural style is proven to improve scalability, manageability, and integrability of software systems [9], [10].

We now detail the microservices in the architecture of DOCKERFINDER, which is depicted[1] in Fig.1. We separately discuss the microservices in the *analysis* group (Sect. II-A), those in the *storage* group (Sects. II-B), and those in the *discovery* group (Sect. II-C).

A. Microservices in the analysis group

As illustrated in Fig. 1, the *analysis* functionality is carried on by a *Crawler*, a *Message Broker*, multiple *Scanner*s, and a *Checker*.

Crawler. The *Crawler* crawls the Docker images to be analysed from the remote Docker registry. More precisely, the *Crawler* crawls the *names* of the images to be analysed, and it sends such names to the *Message Broker* (as it is the task of *Scanner*s to pull and concretely analyse images).

The *Crawler* can also implement filtering policies on the image names that it crawls. For example, the *Crawler* can crawl only images with a particular tag (e.g., *latest*), or it can avoid to re-send the names of images whose description is already available and up-to-date in the local repository.

[1]The figure displays all microservices forming DOCKERFINDER (as rectangles) and the interactions among them (as arcs connecting rectangles). The microservices forming DOCKERFINDER are partitioned in three groups (i.e., *analysis*, *storage*, and *discovery*), which corresponds to the three main functionalities carried on by DOCKERFINDER itself.

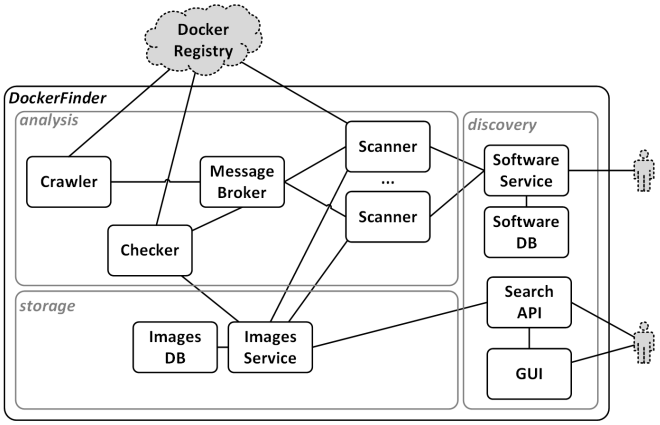


Fig. 1. Microservice-based architecture of DOCKERFINDER.

Message Broker. A message broker is an intermediary service whose purpose is to take incoming messages from one or multiple sources, to process such messages, and to route them to one or more destinations [14]. The DOCKERFINDER's *Message Broker* receives the names of the images to be analysed (from the *Crawler* or from the *Checker*), it stores them into a queue, and it permits *Scanner*s to retrieve them.

The main advantage of using the *Message Broker* is decoupling the *Crawler* and the *Checker* from the multiple *Scanner*s. Indeed, once the *Crawler* (or the *Checker*) has sent an image name to the *Message Broker*, the *Crawler* can continue its crawling process, being confident that the *Message Broker* will retain the message until a *Scanner* retrieves it.

Scanner. The *Scanner* retrieves the name of the images from the *Message Broker*, and it analyses and builds descriptions for the corresponding images. More precisely, the *Scanner* continuously works as follows:

1) It retrieves an image name from the *Message Broker*.
2) It pulls the corresponding image and downloads its metadata from the registry.
3) It extracts the software distributions supported by the pulled image. To do so, it runs a container from the image, and it exploits the information given by the *Software Service* (see Sect. II-C) to determine which software distributions are supported within the running container[2].
4) It generates the image description by exploiting the retrieved metadata and the extracted software distributions.

[2]Given an image `i` to be analysed, the *Scanner* creates a non-stopping container by running `docker run i ping 127.0.0.1`, and it stores the returned container `id`. Then, for each software distribution `s = <command, options, output>` retrieved from the *Software Service*, the *Scanner* executes `docker exec id command options`, and it parses the returned output to check whether such output is compatible with the (regular expression defined in) `output`. If this is the case, the image is marked as supporting `s`. Once the presence of all software distributions have been checked, the *Scanner* stops the container by running `docker stop id`.

5) It sends the produced description to the *Images Service*.

Checker. The main purpose of the *Checker* is to enforce eventual consistency between the image descriptions in the local repository and the images contained in the remote Docker registry. The latter is dynamic, as each image it contains can be updated (e.g., changing the software distributions it supports) or deleted by image developers at any time. The *Checker* is hence in charge of (i) forcing a new analysis of images whose description is out-of-date[3], and of (ii) deleting the description of images that are no longer available within the remote Docker registry. To do so, the *Checker* periodically[4] checks the up-to-dateness of image descriptions as follows:

1) It retrieves all image descriptions stored in the local repository (from the *Images Service*).
2) For each image description, it checks whether (i) such description is out-of-date, or whether (ii) the corresponding image has been deleted from the remote Docker registry. In the case of (i), the *Checker* forces a new analysis of the image by sending its name to the *Message Broker* (to add it to the queue of images to be analysed by *Scanner*s). In the case of (ii), the *Checker* asks to the *Images Service* to delete the image description.

B. Microservices in the storage group

DOCKERFINDER stores all image descriptions produced by the *Scanner*s into a local repository, and it makes them accessible to the other microservices in DOCKERFINDER. To accomplish such a *storage* functionality, DOCKERFINDER relies on two microservices (see Fig. 1):

Images Database. The *Images Database* contains the local repository for storing image descriptions.

Images Service. The *Images Service* is a RESTful service that permits searching, adding, deleting, and updating image descriptions into the *Images Database*. The *Images Service* is not exposed outside of DOCKERFINDER, as it can only be consumed by the microservices in DOCKERFINDER.

C. Microservices in the discovery group

The sub-functionalities of the microservices in the *discovery* group are twofold: On the one hand, (i) they permit configuring the list of software distributions to be searched (by *Scanner*s) within images. On the other hand, (ii) they permit searching for images by submitting queries to DOCKERFINDER.

The microservices that (i) permit configuring the list of software distributions to be searched within images are the *Software Database* and the *Software Service*. Instead, the microservices that (ii) permit submitting queries and searching for images are the *Search API* and *GUI*.

Software Database. The *Software Database* contains the repository of software distributions whose support has to be searched within an image. A software distribution `s` is specified as a triple `<command, options, output>`, which states the `command` and `options` to be executed in a container run from a Docker image, and the expected `output` if `s` is actually supported by such image.

Software Service. The *Software Service* is a RESTful service that permits adding, updating, and deleting the triples representing software distributions to the *Software Database*. The *Software Service* is publicly available outside of DOCKERFINDER, which means that end-users just have to invoke the exposed RESTful API to update the list of software distributions to be searched within images.

Search API. The *Search API* is a remotely accessible API that acts as a proxy for the searching capabilities of the *Images Service*[5]. More precisely, users can submit queries to the *Search API*, which then adapts and forwards the request to the *Images Service*. Once the *Images Service* returns the image descriptions satisfying the submitted query, the *Search API* forwards such descriptions to the invoker.

GUI. The *GUI* is a web-based graphical user interface that allows users to build queries, which can then be submitted to DOCKERFINDER (through the *Search API*). The image descriptions satisfying the query (i.e., those returned by the *Search API*) are then visualised by the *GUI* itself.

III. DOCKERFINDER PROTOTYPE

We hereby introduce the open source DOCKERFINDER prototype[6], by first illustrating how DOCKERFINDER has been implemented as a multi-container Docker application [4] (Sect. III-A), and then by showing two alternative deployment solutions for such an application (Sect. III-B).

A. Implementation of the microservice-based architecture

DOCKERFINDER is implemented as a multi-container Docker application [4], i.e. each microservice is implemented and shipped in its own Docker container. Fig. 2 displays such a multi-container Docker application by representing each Docker container as a box labelled with the name of the microservice it implements, and with the logo of the official Docker image used to ship such a microservice. Fig. 2 also illustrates the communication protocols exploited by the microservices to interact each other (viz., HTTP, AMQP, or mongodb), and the Docker registry from which to retrieve the images to be analysed (viz., the Docker Hub [5]).

We now separately discuss the implementation of the microservices in the *analysis*, *storage*, and *discovery* groups.

[3]An image description stored in the local repository is *out-of-date* if it has been generated before the last update of the corresponding image in the remote Docker registry.

[4]The periodicity of the *Checker* can be configured to trade-off up-to-dateness of image descriptions and performances of DOCKERFINDER.

[5]The reason of having such a kind of proxy is to avoid exposing all functionalities of the *Images Service*, which would also permit users to add, update, or delete the image descriptions contained in the local repository (hence potentially hampering the eventual consistency between such repository and the remote Docker registry).

[6]We released the source code of DOCKERFINDER under an Apache License, and we made it publicly available on GitHub at https://github.com/di-unipi-socc/DockerFinder. A running instance of DOCKERFINDER can be accessed at http://black.di.unipi.it/dockerfinder.

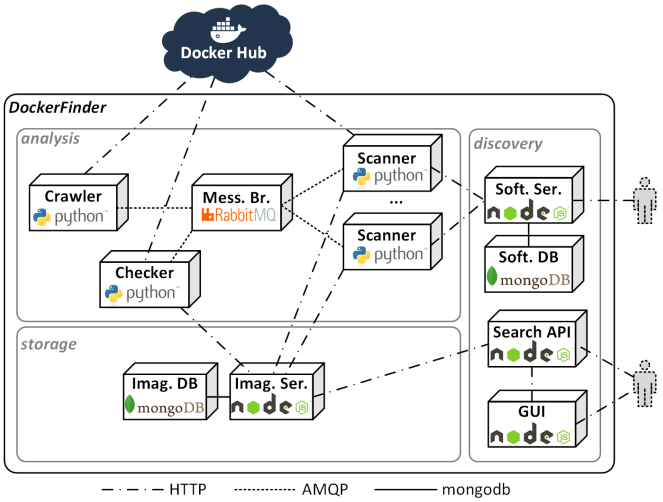


Fig. 2. DOCKERFINDER as a multi-container Docker application.

Analysis. The *Message Broker* is implemented by directly exploiting the official Docker image for RabbitMQ (https://hub.docker.com/_/rabbitmq/).

The *Crawler*, the *Checker*, and the *Scanner*s are instead implemented as Python modules, which are shipped in Docker containers based on the official Docker image for Python (https://hub.docker.com/_/python/). All modules exploit the Python library *requests*[7] for interacting with the HTTP APIs of the Docker Hub and of the *Software Service*, and the Python library *pika*[8] for communicating (via AMQP) with the *Message Broker*. The *Scanner* module also exploits the Python library *docker-py*[9] for creating containers from the images to be analysed, and for managing such containers.

Storage. The *Images Database* is implemented as a NoSQL database hosted on a MongoDB container (https://hub.docker.com/_/mongo/).

The *Images Service* is a RESTful API implemented in JavaScript, which is shipped in a container based on the official Docker image for NodeJS (https://hub.docker.com/_/node/). The *Images Service* API provides the HTTP methods for retrieving, adding, updating, and deleting image descriptions. To do so, it exploits the JavaScript libraries *express*[10] and *mongoose*[11] to run a web server and to interact (via mongodb) with the *Images Database*, respectively.

Discovery. The *Software Database* is implemented as a NoSQL database hosted on a MongoDB container.

The *Software Service* is a RESTful API implemented in JavaScript, which offers the HTTP methods for retrieving, adding, updating, and deleting the triples modelling the software distributions to be searched within containers by *Scanner*s. Similarly to the *Images Service*, the *Software Service* exploits the libraries *express* and *mongoose*, and it is shipped in a container based on the official image for NodeJS.

The *Search API* and the *GUI* are also implemented in JavaScript, and shipped in Docker containers based on the official image for NodeJS. The *Search API* is an HTTP API providing the endpoint `/search`, which permits submitting multi-attribute queries for searching for images (e.g., by invoking `GET /search?python=3.4&pulls_gt=20`, we look for all images having `python 3.4` installed and whose number of `pulls` is greater than `20`). The *GUI* is instead a web application, which permits building the queries to be submitted to the *Search API* in a graphical environment. Similarly to the *Images Service* and the *Software Service*, both the *Search API* and the *GUI* are implemented by exploiting the JavaScript library *express* to run a web server. The *GUI* is also exploiting the JavaScript library *AngularJS*[12] for implementing the graphical interface.

B. How to deploy DOCKERFINDER

DOCKERFINDER is a multi-container Docker application, and it can be deployed in two different configurations, depending on whether the target infrastructure is a single host or a cluster of multiple hosts.

Single host deployment. Docker Compose [4] permits deploying a multi-container application on a single host, if such application is equipped with a *docker-compose.yml* file that describes the application deployment. DOCKERFINDER is equipped with its own *docker-compose.yml* file, and it can hence be deployed on any host supporting Docker Compose.

Multi-host deployment. Docker Swarm [8] permits defining clusters of hosts where to schedule the containers forming a multi-container application. DOCKERFINDER is equipped with two scripts, called *init_all.sh* and *start_all.sh*, which exploit Docker Swarm to initialise and schedule the containers forming DOCKERFINDER on a cluster of multiple hosts. The default configuration is for a cluster of three virtual machines: Two virtual machines are in charge of running 8 *Scanner*s each, while the third virtual machine is running the other containers in the architecture of DOCKERFINDER.

IV. RELATED WORK

Due to space limitations, we hereby discuss only solutions whose aim is to improve the image search capabilities currently offered by Docker.

The closest solution to DOCKERFINDER is probably the Docker Store [7]. Docker Store is a web-based application that extends the image search capabilities provided by the Docker Hub. Despite it still permits searching for images "by name" (i.e., it permits specifying a term to be matched within the name, description, or username associated to an image), it also permits filtering the returned images by category (e.g., application framework images, database images, programming languages images). Docker Store differs from DOCKERFINDER

[7]http://docs.python-requests.org/.
[8]http://pika.readthedocs.io/.
[9]https://docker-py.readthedocs.io/.
[10]http://expressjs.com/.
[11]http://mongoosejs.com/.
[12]https://angular.io/.

mainly because of two reasons. First, Docker Store allows to filter only trusted and validated images, while DOCKERFINDER permits applying filters to all images available in a Docker registry. Second, as Docker Store only allows to search images "by name" and then to filter them by category, it is not possible to distinguish, for instance, whether an image supports Java or Python, since all images supporting such languages fall in the same category (viz., the category of images supporting programming languages). DOCKERFINDER does not suffer from the same limitation, as it permits explicitly searching for images supporting either Java or Python, or both.

JFrog's Artifactory [12] is an Universal Artefact Repository working as a single access point to software packages created by any language or technology (i.e. including Docker). JFrog users can search for Docker images by their name, tag or image digest. Users can also assign custom properties to images, which can then be exploited to specify and resolve queries. JFrog differs from DOCKERFINDER since it requires users to manually assign properties to images, while DOCKERFINDER automatically produces image descriptions by retrieving their metadata from the remote Docker registry, and by analysing (containers run from) images to extract their runtime features.

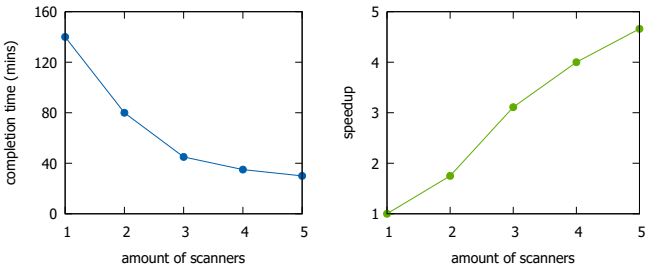
Finally, it is worth noting that our work shares with Wettinger et al. [15] the general objective of contributing to ease the discovery of DevOps "knowledge" (which includes Docker images). [15] proposes a collaborative approach to store DevOps knowledge in a shared, taxonomy-based knowledge base. More precisely, [15] proposes to build the knowledge-base in a semi-automated way, by (automatically) crawling heterogeneous artefacts from different sources, and by requiring DevOps experts to share their knowledge and (manually) associate metadata to the artefacts in the knowledge-base. DOCKERFINDER instead focuses only on container images, and it builds the description of such images in a fully-automated way.

V. CONCLUSIONS

The current support for searching Docker images is limited, as users can only look for images in Docker registries "by name" (i.e., by specifying a term that should occur in name, description, or username associated to an image).

In this paper, we tried to overcome the aforementioned limitations by introducing DOCKERFINDER, a microservice-based prototype that permits searching for Docker images not only "by name", but also imposing requirements on other attributes (e.g., the software distributions they support, their size, or their popularity in the Docker community). DOCKERFINDER crawls images from a remote Docker registry, it automatically analyses such images to produce descriptions to be stored in a local repository, and it permits searching for images by querying the local repository.

It is worth observing that DOCKERFINDER is not thought to replace Docker registries, but to permit improving their search capabilities. The ideal situation would be to integrate the capabilities of DOCKERFINDER within existing Docker registries (e.g., Docker Hub). This is precisely one of the reasons why DOCKERFINDER is designed following the microservice-based architectural style [10], which is proven to simplify system integration [9].



(a) (b)

Fig. 3. Time performances registered for analysing a set of 100 images randomly sampled from the Docker Hub, where each image was analysed by *Scanner*s by checking the availability of 16 different software distributions. In both plots, the x-axes represent the amount of *Scanner*s actually running in the running instance of DOCKERFINDER. The y-axes instead represent the (a) completion time and the (b) corresponding speed-up.

By running DOCKERFINDER, we discovered that the most time consuming task is that of *Scanner*s, which have to spend time in downloading images and in analysing them to produce their description. However, the analysis of images is independent one another, and it can hence be easily scaled out to improve the time performances of DOCKERFINDER (as shown[13] in Fig. 3). Given the fact that DOCKERFINDER is a multi-container Docker application (implementing a microservice-based architecture), scaling *Scanner*s just corresponds to manually increasing/decreasing the amount of corresponding Docker containers actually running. As part of our immediate future work, we plan to provide DOCKERFINDER with auto-scaling capabilities.

Four other interesting extensions of DOCKERFINDER are: allowing it to simultaneously crawl images from multiple Docker registries, including user authentication to permit offering DOCKERFINDER as a multi-tenant service, enforcing security policies in the analysis of Docker images (e.g., DockerPolicyModules [1]), and improving the image caching policies of DOCKERFINDER. Maintaining an image in a local cache would permit lowering the time needed for downloading the images extending it. Since it is practically unfeasible to maintain locally all images available in the Docker Hub [6], we need a smart caching policy to decide which images to maintain and which images to destroy.

Last, but not least, we view DOCKERFINDER as a first step towards the definition of a more general framework allowing users to specify arbitrary criteria for analysing container images (not only limited to Docker, but also including other container technologies — e.g., rkt [2]). The development of such a framework is in the scope of our future work, as we believe that it would open new research paths. For instance, re-

[13]The results displayed in Fig. 3 have been obtained by running DOCKERFINDER on a Ubuntu 16.04 LTS workstation having a AMD A8-5600K APU (3.6 GHz) and 4 GBs of RAM.

searchers could start generating datasets of customised image descriptions, which could then be used to extract knowledge by applying data mining approaches, or to experiment and validate research solutions.

ACKNOWLEDGMENTS

This work has been partly supported by the project *Through the fog* (PRA_2016_64) funded by the University of Pisa.

REFERENCES

[1] Enrico Bacis, Simone Mutti, Steven Capelli, and Stefano Paraboschi. DockerPolicyModules: Mandatory access control for docker containers. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 749–750, 2015.

[2] CoreOS. rkt. https://coreos.com/blog/rocket/.

[3] Docker Inc. Docker. https://www.docker.com/. Last accessed: January 10th, 2017.

[4] Docker Inc. Docker compose. https://docs.docker.com/compose/. Last accessed: January 10th, 2017.

[5] Docker Inc. Docker hub. https://hub.docker.com/. Last accessed: January 10th, 2017.

[6] Docker Inc. Docker hub hits 5 billion pulls. https://blog.docker.com/2016/08/docker-hub-hits-5-billion-pulls/. Last accessed: January 10th, 2017.

[7] Docker Inc. Docker store. https://store.docker.com/.

[8] Docker Inc. Docker swarm. https://docs.docker.com/swarm/. Last accessed: January 10th, 2017.

[9] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *CoRR*, abs/1606.04036, 2016.

[10] Martin Fowler and James Lewis. Microservices. ThoughtWorks, https://www.thoughtworks.com/insights/blog/microservices-nutshell, 2016.

[11] Michael Httermann. *DevOps for Developers*. Apress, Berkely, CA, USA, 1st edition, 2012.

[12] JFrog Ltd. Docker: Secure Clustered HA Docker Registries With A Universal Artifact Repository. https://www.jfrog.com/support-service/whitepapers/docker/. Last accessed: January 10th, 2017.

[13] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure devops. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 202–211. IEEE Computer Society, 2016.

[14] Stephen James Paul Todd. Message broker apparatus, method and computer program product, 2003. US Patent 6,510,429.

[15] Johannes Wettinger, Vasilios Andrikopoulos, and Frank Leymann. Automated capturing and systematic usage of devops knowledge for cloud applications. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 60–65, March 2015.