SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2018; **48**:8,1461–1474
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.2583

A microservice-based architecture for (customisable) analyses of Docker images

Antonio Brogi, Davide Neri, Jacopo Soldani

Department of Computer Science, University of Pisa, Italy. name.surname@di.unipi.it

SUMMARY

We introduce DOCKERANALYSER, a microservice-based tool that permits building customised analysers of Docker images. The architecture of DOCKERANALYSER is designed to crawl Docker images from a remote Docker registry, to analyse each image by running an analysis function, and to store the results into a local database. Users can build their own image analysers by instantiating DOCKERANALYSER with a custom analysis function and by configuring the architecture. More precisely, the steps needed to obtain new analysers are: (i) replacing the analysis function used to analyse crawled Docker images, (ii) setting the policy for crawling Docker images, and (iii) setting the scalability options for obtaining a scalable architecture. In this paper, we also present two different use cases, i.e., two different analysers of Docker images created by instantiating DOCKERANALYSER with two different analysis functions and configuration options. The two use cases show that DOCKERANALYSER decreases the effort required to obtain new analysers versus building them from scratch.

Received . . .

KEY WORDS: Docker; Microservices; Container

1. INTRODUCTION

Container-based virtualisation [23, 24] has gained significant acceptance, because it provides a lightweight solution for running multiple isolated user-space instances (called *containers*). Such instances are particularly suited to package, deploy and manage complex, multi-component applications [3]. Developers can bundle application components along with the dependencies they need to run in isolated containers and execute them on top of a container run time (e.g., Docker [10], Rkt [7], Dynos [20]). Compared to previous existing virtualisation approaches, like virtual machines, the use of containers features faster start-up times and less overhead [22].

The current de-facto standard technology for container-based virtualization is Docker [29, 9], a platform for building, shipping, and running applications inside portable 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 together with all the software dependencies needed to run it (e.g., libraries, binaries). In addition, Docker provides the ability to distribute and search (images of) containers that were created by other developers through Docker registries. Given that any developer can create and distribute its own created images through Docker registries, other users have at their disposal plentiful repositories of heterogeneous, ready-to-use images. In this scenario, public registries (such as the official Docker Hub [12]) are playing a central role in the distribution of images [13].

However, images stored in Docker registries are described by fixed attributes (e.g., name, description, owner of the image), and this makes it difficult for users to analyse and select the images satisfying their needs. Also, needs may differ from user to user, depending on the actual exploitation of Docker images they wish to carry out. For instance, a developer may want to deploy her application on a Docker image supporting precise software distributions (e.g., *Python 2.7* and *Java 1.8*), an end-user may want to assign custom tags to her images in order to ease their retrieval, or a data scientist may wish to analyse images to discover interesting, recurring patterns.

Currently, a support for performing analyses on large set of Docker images is missing. Users are required to manually check whether an image satisfies their needs by looking at the attributes provided by the Docker registry or on the image features by running it in a container.

In this paper, we present DOCKERANALYSER, a tool that permits building customised analysers of Docker images. Users can create their own Docker image analysers by simply instantiating DOCKERANALYSER with a user-defined analysis function that produces descriptions of Docker images. The analysis function can be any Python code that, given the name of a Docker image, scans such image to extract some metadata that are used to generate the description of the image. DOCKERANALYSER is designed to provide a scalable architecture for running the analysis function provided by the users on large set of Docker images in a fully automated way. Users are only required to provide the analysis function, while DOCKERANALYSER provides the other functionalities for crawling Docker images from a Docker registry, running the analysis function on each image, storing the results of the analysis function in a local storage, and allowing to query the storage through a RESTful API.

To illustrate this, we implemented two different analysers of Docker images, namely DOCKERFINDER and DOCKERGRAPH.

- DOCKERFINDER collects the software distributions supported by an image, and it permits searching for images supporting such software distributions. For instance, if a developer wishes to package her application into a Docker image satisfying certain software requirements (e.g. *Python 2.7* and *Java 1.8*), she can query DOCKERFINDER and select the image that best satisfies such requirements.
- DOCKERGRAPH creates a directed graph whose nodes are names of (repositories of) Docker images, and whose arcs connect each image i to its *parent* image (viz., the image that has been used as the basis to create i), if any. Many applications can take advantage of the graph created by DOCKERGRAPH. For instance, if a parent image is affected by a security flaw, DOCKERGRAPH can be used for retrieving all the images that are built starting from such image.

We deployed both DOCKERGRAPH and DOCKERFINDER as multi-container Docker applications where the microservices of the analysers run inside Docker containers.

We chose to implement DOCKERANALYSER as a suite of interacting microservices mainly because of the configurability properties of microservice-based architectures [17, 28]. For instance, replaceability in a microservice-based architecture allows replacing a microservice with another offering the same interface, without affecting any of the other microservices composing the architecture [28]. In DOCKERANALYSER, replaceability allows obtaining different analysers by just changing the actual implementation of the microservice running the analysis function. We illustrate this by showing how DOCKERFINDER and DOCKERGRAPH are built by changing the implementation of such microservice.

This paper is an extended version of [5]. [5] describes a tool that analyses Docker images by executing a fixed analysis function. DOCKERANALYSER is a generalisation of that in [5], and such generalisation permits customising the analysis function executed by the architecture in order to create different analysers of Docker images.

The rest of the paper is organised as follows. Sect. 2 describes the microservice-based architecture of DOCKERANALYSER. Sect. 3 introduces the DOCKERANALYSER tool. Sect. 4 presents two use cases of analysers of Docker images (DOCKERFINDER and DOCKERGRAPH) obtained by customising the analysis function of DOCKERANALYSER. Sect. 5 discusses related work. Sect. 6 draws some conclusions.

2. DOCKERANALYSER ARCHITECTURE

The objective of DOCKERANALYSER (Fig. 1) is to permit building analysers of Docker images. A new analyser of Docker images can be created by instantiating DOCKERANALYSER with a different analysis function (contained in the *deploy package*). We implemented DOCKERANALYSER as a suite of interacting microservices.

- *Analysis*. DOCKERANALYSER crawls and analyses each image contained in the Docker registry it is connected to. The analysis of the images is performed by running the analysis function provided by the user.
- *Storage*. DOCKERANALYSER stores all produced image descriptions into a local storage. The storage is then made accessible to external users through a RESTful API.

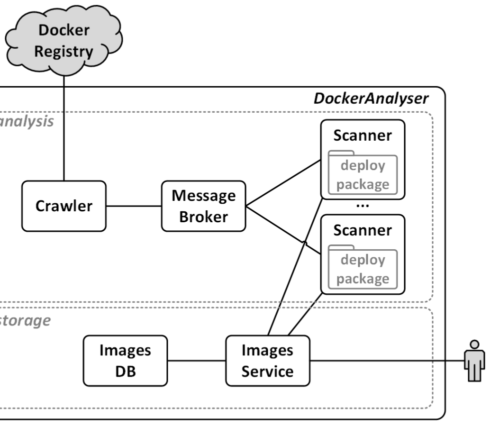

Figure 1. Microservice-based architecture of DOCKERANALYSER.

We now detail the microservices composing the architecture of DOCKERANALYSER (Fig. 1). We separately discuss the microservices in the *analysis* group (Sect. 2.1) and those in the *storage* (Sect. 2.2) group.

2.1. Microservices in the Analysis group

As illustrated in Fig. 1, the *analysis* is carried out by a *Crawler*, a *Message Broker*, and (one or more) *Scanner*s.

Crawler. The *Crawler* crawls the Docker images to be analysed from a remote Docker registry. More precisely, the *Crawler* crawls the *names* of the images from the registry, and it passes such names to the *Message Broker*. The *Crawler* can be configured by the users to implement two different crawling policies: randomly or sequentially. The former permits crawling a random sample of images, while the latter permits crawling all the images sequentially. In both cases, the total number of images to be crawled can be configured.

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 [6]. The *Message Broker* of DOCKERANALYSER receives the names of the images to be analysed (from the *Crawler*), it stores them into a messages queue, and it permits the *Scanner*s to retrieve them. The goal of the *Message Broker* is to decouple the *Crawler* from the *Scanner*s.

Scanner. The *Scanner* retrieves the name of the images from the *Message Broker*, and for each name received it runs the analysis function. More precisely, given a user-defined function *analysis*, each *Scanner* continuously works as follows:

1. It retrieves an image name *i* from the *Message Broker*.

2. It runs the analysis function *analysis* on the image name *i* producing a description *descr*=analysis*(i)*.
3. It sends the generated description *descr* to the *Images Service* that stores the description into the local storage.

The description *descr* sent to the *Images Service* is a JSON object containing the information obtained by running the analysis function on the image. It is worth noting that the *Scanner*, depending on the analysis function executed, can be the most time consuming service in the architecture. For example, if the function *analysis* requires downloading all layers of a Docker image locally then it can require up to minutes to download a single image. In order to decrease the time needed to analyse images, the number of *Scanner* microservices can be increased by exploiting the scalability property of microservice-based architectures (see Sect.4.1 for a concrete example of analyser exploiting such scalability to reduce the time to analyse images).

2.2. *Microservices in the* storage *group*

DOCKERANALYSER stores all image descriptions produced by the *Scanner*s into a local storage. The images descriptions stored in the local storage are made accessible through a RESTful API. To accomplish such a *storage* functionality, DOCKERANALYSER relies on a microservice composed by the *Image Service* and *Image Database* (Fig. 1).

Images Database. The *Images Database* is the local repository where the image descriptions are stored. Given that different analysis functions can produce different image descriptions, the *Images Database* has been implemented as a NoSQL database without a fixed model.

Images Service. The *Images Service* is a RESTful service that permits adding, deleting, updating, and searching image descriptions inside the *Images Database*. The *Images Service* interface is used both by other microservices in DOCKERANALYSER (for adding, deleting, updating images descriptions) and by external users (for submitting queries to the local repository).

3. DOCKERANALYSER

We hereby illustrate the implementation of DOCKERANALYSER[*] (Sect. 3.1) and we then show the steps needed to obtain different analysers of Docker images (Sects. 3.2 and 3.3).

3.1. *Implementation of* DOCKERANALYSER

The microservice-based architecture of DOCKERANALYSER has been implemented as a multi-container Docker application, where each microservice is implemented and shipped in its own Docker container. Fig. 2 illustrates 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 microservice. Fig. 2 shows also the communication protocol exploited by the microservices to interact each other (viz., HTTP, AMQP), and the Docker registry from which to retrieve the images to be analysed (viz., the Docker Hub). We now separately discuss the implementation of the microservices in the *analysis* and *storage* groups.

Analysis. The *Message Broker* is implemented by directly exploiting the official Docker image for RabbitMQ[†]. The *Crawler* and the *Scanner* are instead implemented as Python modules, which are shipped in Docker containers based on the official Docker image for Python[‡]. Both modules exploit the Python library *pika* [30] for communicating (via AMQP) with the *Message Broker*. *Crawler* uses also the Python library *requests* [31] for interacting with the Docker Hub REST API. The *Scanner*

[*]The source code of DOCKERANALYSER is available on GitHub https://github.com/di-unipi-socc/DockerAnalyser.
[†]RabbitMQ Docker image https://hub.docker.com/_/rabbitmq/
[‡]Python Docker image https://hub.docker.com/_/python/

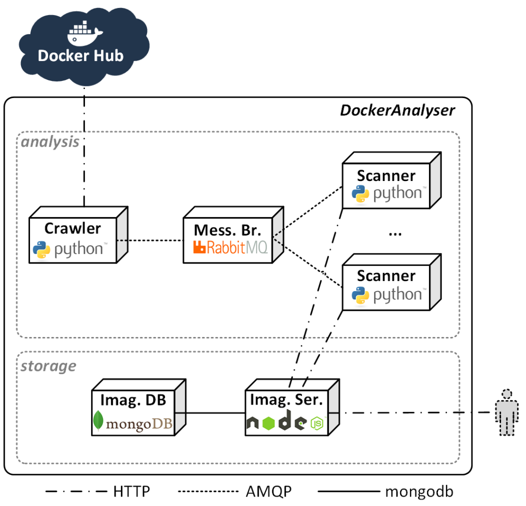

Figure 2. DOCKERANALYSER as a multi-container Docker application.

module is configured to import and run the user-defined analysis function. By default, the analysis function is a *void* function that given an image to be analysed it sends the same image to the *Images Service*. The steps needed to execute a custom analysis function in DOCKERANALYSER are listed in Sect. 3.2.

Storage. The *Images Database* is implemented as a NoSQL database hosted on a MongoDB container[§]. The *Images Service* is a RESTful API implemented in JavaScript, which is shipped in a container based on the official Docker image for NodeJS[¶]. The *Images Service* API provides the HTTP methods for adding, updating, deleting, and searching image descriptions. To do so, it exploits the JavaScript framework *express* [16] to run a web server and *mongoose* [27] sto interact (via mongodb) with the *Images Database*. The *Images Service* API returns the image descriptions as JSON documents.

3.2. *How to create new Docker image analysers*

A user can instantiate DOCKERANALYSER in order to obtain new Docker image analysers. The steps needed for obtaining a new analyser consist of (i) replacing the analysis function, (ii) selecting the crawling policies of the *Crawler* microservice, and (iii) setting the scaling options of the *Scanner* microservice.

The analysis function is replaced by instantiating the *Scanner* microservice of DOCKERANALYSER with a user-defined analysis function. More precisely, the steps needed to instantiate DOCKER-ANALYSER with a customised analysis function are the following:

1. Clone the GitHub repository of DOCKERANALYSER locally.
2. Create a folder `F` (that represents the *deploy package* – Fig. 1) and inside the created folder create the following files:
   (a) The `analysis.py` file that contains the code of the custom analysis function,
   (b) The `requirements.txt` file that contains the Python library dependencies[||],
   (c) Any other file needed by the analysis function (e.g., configuration files)

[§]MongoDB Docker image https://hub.docker.com/_/mongo/
[¶]NodeJS Docker image https://hub.docker.com/_/node/
[||]The file `requirements.txt` is empty if the implemented function does not have dependencies.

3. Build the *Scanner* Docker image with Docker Compose [11] by running the command `docker-compose build --build-arg DEPLOY_PACKAGE_PATH=<F> scanner`, (where `F` is the name of the folder created at step 2).

It is worth noting that step 3 builds the Docker image of the *Scanner* with the customised analysis function (contained in the `analysis.py`) that replaces old *Scanner* code. The option `--build-arg DEPLOY_PACKAGE_PATH=<F>` at step 3 copies the folder `F` (containing the custom `analysis.py` file, the `requirements.txt` file and any other files needed to the analysis) into the (old) Docker image of the *Scanner* Docker image hence allowing to create a new image running the new analysis function. The new image of the *Scanner* is indeed built by importing the custom `analysis.py` function and by installing all dependencies listed in the file `requirements.txt` (if any).

```python
1 def analysis(image_json, context):
2     logger = context['logger']
3     client_images = context['images']
4     # return True or False
```

Listing 1. Signature of the function defined in the `analysis.py` file.

The `analysis.py` file stored in the deploy package `F` contains the code of the custom analysis function (written in Python) and it must follow the signature illustrated in Listing 1.

- The `image` parameter is a JSON object containing the `name` of the image to be analysed along with other basic fields taken from the registry (some of the most important fields of the JSON object are `is_automated`, `is_official`, `star_count`, and `pull_count`).
- The `context` parameter is a dictionary containing the objects `images` and `logger` (lines 2, 3). The `images` object can be used for interacting with the *Images Service* API. More precisely, `images` offers the methods `get_image(name)`, `post_image(json)`, `put_image(json)`, `delete_images(id)` for getting, adding, updating, and deleting an image into the *Images Database*, respectively. The `logger` is the standard *logging.Logger* class of Python and it provides a set of methods for logging the actions during the execution of the code (e.g., `info()`, `warning()`, `error()`, `critical()`, `log()`.
- The return code is a boolean value. `True` must be returned by the function if the image has been processed correctly. `False` must be returned for discarding and deleting the image from the *Message Broker*.

Sect. 4 presents two examples of analysis functions that we used to create two different analysers of Docker images.

The second step required to obtain a custom analyser is to set the crawling policy of the *Crawler* microservice for crawling the images from the Docker Hub. The crawling options of the *Crawler* can be found in the `docker-compose.yml` file in the `crawler` service definition. The available configuration options are the following:

`--random` By setting `--random=True` the *Crawler* crawls the images from the Docker registry by randomly selecting them, otherwise it crawls images sequentially.

`--policy` By setting `--policy=stars_first` the *Crawler* crawls the images starting from those with a higher number of stars. Otherwise, by setting `--policy=pulls_first` it crawls first the images with more number of pulls.

`--min-stars` This option permits setting the minimum number of stars (`--min-stars=<integer>`) that an image must have in order to be crawled. All the images with a number of stars less than `--min-stars` are not crawled.

`--min-pulls` This option permits setting the minimum number of pulls (`--min-pulls=<integer>`) that an image must have in order to be crawled. All the images with a number of pulls less than `--min-pulls` are not crawled.

`--only-official` If this option is set, then only the official images stored into the Docker registry are crawled.

`--only-automated` If this option is set, then only the images that are automatically created from a GitHub repository are crawled.

Finally, the user can configure the scaling options of the architecture by setting the number of replicas of the *Scanner* microservice. Running more *Scanner*s in parallel may reduce the time needed to analyse the crawled images. The `deploy` option of the `scanner` in the *docker-compose.yml* file permits specifying the number of parallel *Scanner*s to be started. Listing 2 shows an example of a configuration that starts 10 *Scanner*s in parallel when the analyser is started.

```
scanner:
  ...
  deploy:
    mode: replicated
    replicas: 10
```

Listing 2. An example of configuration of the *Scanner* microservice.

In addition, the *Scanner* can be scaled up or down at run time by using the command `docker-compose scale [SERVICE=NUM...]`. For example, the command `docker-compose scale scanner=5` updates the number of *Scanner* replicas to 5.

3.3. *How to deploy* DOCKERANALYSER

DOCKERANALYSER is a multi-container Docker application which can be deployed using the Docker platform. It can be deployed in two different configurations, depending on whether the target infrastructure is a single host or a cluster of multiple hosts. The *single host deployment* configuration runs all the containers of DOCKERANALYSER in a single node while the *multi host deployment* runs the containers in a cluster of distributed nodes. While former is suitable for running simple and low load analyser, the latter is recommended whenever the analyser requires higher amount of physical resources (e.g., network traffic or storage space) because it permits distributing the load on multiple machines rather than just one.

Single host deployment. Docker Compose [11] permits deploying a multi-container application on a single host if such application is equipped with a *docker-compose.yml* that describes the application deployment. DOCKERANALYSER is equipped with its own *docker-compose.yml* file, and it can hence be deployed on any host supporting Docker Compose. In order to start a newly created analyser, users should submit the command `docker-compose up`.

Multiple host deployment. Docker Swarm [15] permits defining a cluster of Docker engines (called a *swarm*) where to schedule the containers forming a multi-container application. DOCKERANALYSER is equipped with a shell scripts (called *start_swarm.sh*) that allows to start the analyser in a swarm. The script assumes that the user has already configured the swarm where the analyser will be actually executed (as the script itself will have to be executed within the Docker engine managing the swarm). The *start_swarm.sh* script takes as input the name of the deploy package and the name to be assigned to the analyser. The script first creates the *Scanner* image with the deploy package, and it then runs the analyser into the swarm. By default, the script distributes the containers of the analyser among all nodes of the swarm.

4. USE CASES

In this section, we illustrate how different analysers of Docker images can be created by instantiating DOCKERANALYSER with different user-defined analysis functions. In particular, we present DOCKERFINDER and DOCKERGRAPH, two use cases that run different analysis functions. DOCKERFINDER analyses an image by running it in a container and checking whether the image provides a list of software distributions. DOCKERGRAPH, instead, creates a graph of Docker images where each node is a name of the repository of an image and where every node is connected to its parent image. The use cases are obtained by replacing the scanner microservice of DOCKERANALYSER (that consist of replacing the *Scanner* Docker image) while the other microservices in the architecture remain untouched. As presented in Sect. 3.2 replacing the scanner microservice corresponds to building a new *Scanner* Docker image starting from a user-defined deploy package folder (containing the `analysis.py` file, the `requirements.txt` file, and any other files needed by the analysis function).

4.1. DOCKERFINDER

Docker images stored in Docker Hub provide virtually almost any software distributions (e.g., libraries, programming languages, frameworks) to the users. However, the current support for searching such images based on the software distributions they support is missing. Users may want to deploy an application component in a Docker image and that such application requires some specific versions of software distributions (e.g., *Python 2,7*, *Java 1.8*). DOCKERFINDER permits searching for Docker images based on the versions of software distributions they support.

The *deploy-package* folder of DOCKERFINDER contains three files: the `analysis.py` (Listing 3), the `requirements.txt` file that contains only the `docker==2.2.1` python library dependency (used by the analysis function for interacting with Docker daemon), and the `software.json` JSON (Listing 4) file containing the list of software distributions to be searched in each image. The JSON file contains a list of triples, where each triple is composed of the name of the software distribution, the command to be executed in order to know the version of the software, and a regular expression used to search the matching version (if it exists).

The analysis function of DOCKERFINDER is detailed in Listing 3. Lines 1-4 import the Python libraries `json`, `docker`, `re`, and `os` used by the function[**]. Line 6 creates the Docker client object exploited for interacting with Docker daemon. Line 8 defines the `analysis` function that takes as input the image to be analysed and the context. Lines 12-14 pull the image locally using the docker client, then create and start an infinite sleeping container. Lines 16-20 open `software.json` file contained in the deploy package folder and for each software distribution (line 18) takes the command (e.g., *python version*) to be executed and run the command into the already running container. Line 19 the `output` variable that contains the result of execution of the command inside the container. Lines 20-26 use the regular expression to search the version of the software in the output variable (if it exists). Lines 27 adds the software distribution found (if any) in the JSON that will then sent to the images server. Line 28 uses the `client_images.post_image(json)` to post the JSON object containing the results of the analysis into the *Images Service*. Line 29-30 stop the sleeping container and remove it. Line 31 removes also the Docker image analysed. Both the container and the image are removed for freeing storage space.

Users can search Docker images based on the software distributions they support calling the RESTful API of the *Images Service*. The parameters of the *Images Service* API are obtained looking at the fields contained in the JSON object that describe the images analysed. For example, in order to retrieve the Docker images supporting both *Java* and *Python* users can query the *Images Service* with the `GET api/images?python=2.7&java=1.8` method. DOCKERFINDER can be exploited by other tools for obtaining the list of Docker images that satisfy the software

[**]The analysis function can import (in addition to the library present into the `requirements.txt`) any of the standard library provided by Python (e.g., `json`, `re`, `os`).

```
import json
import docker
import re
import os

client_docker= docker.DockerClient(base_url="unix://var/run/docker.sock")

def analysis(image_json, context):
    logger = context['logger']
    client_images = context['images']
    try:
        image = client_docker.images.pull(images_json['name'])
        container = client_docker.containers.create(images_json['name'],
    entrypoint="sleep  infinity")
        container.start()
        softwares = {}
        with open(os.path.join(os.path.dirname(__file__), 'softwares.json')) as
     softwares_json:
            software = json.load(softwares_json)
            for sw in software:
                output = container.exec_run(cmd=sw['cmd']).decode()
                match = re.search(sw['regex'], output)
                if match:
                    version = match.group(0)
                    softwares[sw['name']] = match.group(0)
                else:
                    logger.debug("[{0}] NOT found in ".format(sw['name']))

        images_json['softwares'] = softwares
        client_images.post_image(images_json)
        container.stop(timeout=2)
        container.remove()
        client_docker.images.remove(images_json['name'], force=True)
    except docker.errors.ImageNotFound as e:
        logger.exception("{} image not found".format(images_json['name']))
        return False
    return True
```

Listing 3. `analysis.py` function of DOCKERFINDER

```
[{
  "name": "python",
  "cmd": "python --version",
  "regex": "[0-9]+[.][0-9]*[.0-9]*"
 }, {
  "name": "java",
  "cmd": "java -version",
  "regex": "[0-9]+[.][0-9]*[.0-9]*"
 }
]
```

Listing 4. Some of the software distributions listed in the `software.json` file.

distributions required by an application component that needs to be deployed in Docker containers (e.g., TOSKERISER [4]).

By running DOCKERFINDER, we discovered that the most time consuming task is that of *Scanner*, which have to spend time in downloading images and in analysing them to produce their descriptions. Images can be scanned independently and *Scanner* can hence be easily scaled out to improve the time performances of DOCKERFINDER (as shown[††] in Fig. 3). Exploiting the scalability

[††]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.

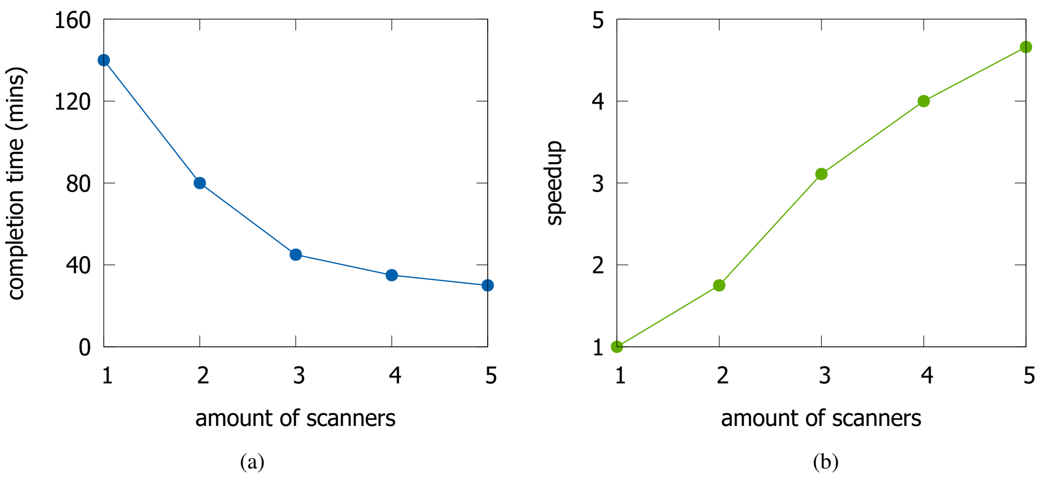

Figure 3. Time performances registered for analysing a set of 100 images randomly sampled from the Docker Hub, where each image was analysed by *Scanner*s by checking the availability of 16 different software distributions. In both plots, the x-axes represent the amount of replicas of *Scanner*s actually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

property of microservice-based architecture, and given the fact that DockerFinder is a multi-container Docker application scaling *Scanner*s just corresponds to manually increasing/decreasing the amount of corresponding Docker containers running.

4.2. DockerGraph

Docker permits reusing an already existing image for building other images. An image reused by another image is called parent image. Most of the Docker images stored in Docker Hub are built by reusing already existing images. However, the support for knowing the parent relationship occurring between images is missing. Knowing which are the images that are more used by other images or knowing the images use a single parent image can be exploited for many applications. For example, if a parent image is affected by a security flaw, having the graph of all the images that reused the affected image as parent image can be useful for patching the flaw in such images. DockerGraph can be also used by other tools that require to implement a smart caching policy of images. For instance, the graph can be exploited for maintaining the images locally that are more used as parent image without deleting them.

DockerGraph constructs a directed graph of images where the nodes are the repository names of images and a link from an image `s` to an image `p` is added if the image `p` is the parent image of *s*. DockerGraph retrieves the repository name of the parent image by looking at the *FROM* option in the Dockerfile used to create an image.

The deploy package folder of DockerGraph is composed by the `analysis.py` file and an empty `requirements.txt` because the analysis function requires no external libraries. The `analysis` function of DockerGraph is shown in Listing 5. Lines 1-2 import the *requests* and *re* Python libraries used to interact with the GitHub API and for handling regular expression, respectively. Lines 4-24 defines the `analysis` function of DockerGraph. Lines 5-6 gets the `logger` and the `client_images` objects. Line 10 checks whether the image has been already analysed by calling the `is_new()` method of the `client_images` object provided by the context. If the image has not been already analysed, line 13 calls the `get_dockerfile(repo)` method that retrieve the image's Dockerfile stored into Docker Hub (if it is present). Line 14 calls `extract_FROM(dockerfile)` method that use a regular expression for extracting the *FROM* option present into the Dockerfile. It returns a couple of strings where the first is the repository name and the second is the tag of the parent image. Lines 15 adds the repository name (`from_repo`) and

```
1  import requests
2  import re
3
4  def analysis(image_json, context):
5      logger = context['logger']
6      client_images = context['images']
7
8      repo = image_json["repo_name"]
9      logger.info("Received image to be analysed: {} ".format(repo))
10     if client_images.is_new(repo):
11         node_image = {'name': repo}
12         try:
13             dockerfile = get_dockerfile(repo)
14             from_repo, from_tag = extract_FROM(dockerfile)
15             node_image['from_repo'] = from_repo
16             node_image['from_tag'] = from_tag
17             client_images.post_image(node_image)
18         except ValueError as e:
19             logger.error(str(e))
20             return False
21         return True
22     else:
23         logger.info("{}  already present into local database ".format(repo))
24         return False
25
26 def extract_FROM(dockerfile):
27     search = re.search('FROM ([^\s]+)', dockerfile)
28     if search:
29         from_image = search.group(1)
30         if ":" in from_image:
31             from_repo, from_tag = from_image.split(":")
32         else:
33             from_repo = from_image
34             from_tag = None
35         return from_repo, from_tag
36     else:
37         raise ValueError("FROM value not found in DockerFile")
38
39
40 def get_dockerfile(repo_name):
41
42     docker_url = "https://hub.docker.com/v2/repositories/{}/dockerfile/"
43     try:
44         response = requests.get(docker_url.format(repo_name))
45         dockerfile = response.json()['contents']
46         return dockerfile
47     except ConnectionError as e:
48         raise e
```

Listing 5. `analysis.py` function of DOCKERGRAPH

the tag (`from_tag`) of the parent image name Python dictionary describing the image. In line 17 the `client_images.post_image(JSON)` method is called to add the `node_image` dictionary to the *Images Service*.

DOCKERGRAPH has been executed[‡‡] to crawl sequentially the repository stored in Docker Hub. At the time of executing DOCKERGRAPH, Docker Hub contained approximately 600000 repositories. The graph constructed by DOCKERGRAPH counts 87570 repository names because DOCKERGRAPH discarded all the repositories in Docker Hub whose Dockerfiles are not present or badly formatted. Fig. 4 shows only the top 10 images used as parent images by other images. The most used image is *ubuntu* with 15208 images using it as parent image, while *nginx* is used as parent image by 1697 images.

[‡‡]DOCKERFINDER has been executed on a Ubuntu 16.04 LTS workstation having a AMD A8-5600K APU (3.6 GHz) and 4 GBs of RAM.

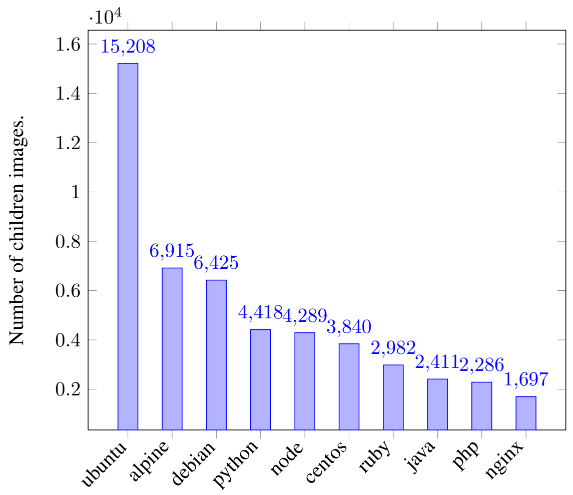

Figure 4. Top ten Docker images used as parent images.

It is worth nothing that the image descriptions obtained by an analyser are returned as raw data (i.e., JSON objects). It is left to the users to post-process the raw data and to visualise it (like in the chart in Fig. 4) using data visualisation tools. The data visualisation of the obtained images descriptions is outside of the scope of this work.

4.3. *Discussion*

In this section, we discuss the advantages of using DOCKERANALYSER versus building Docker images analysers from scratch. We evaluated the usefulness of DOCKERANALYSER by considering (i) the number of functionalities to be developed, (ii) the reusability of the code, and (iii) the time required to obtain new analysers starting from already existing analysers.

DOCKERANALYSER reduces the number of functionalities to be developed with respect to build the analyser from scratch. Table I reports four of the main functionalities that an analyser should support: *Image crawling* is how the images are crawled from a Docker registry, *Image analysis* is how the images are analysed, *Storage of results* is how the results are stored, and *Scalable architecture* is how to implement a scalable architecture. As shown by Table I, DOCKERANALYSER considerably simplifies the building of analysers by requiring to only account for the *Image analysis*, as the others functionalities are provided by the architecture. *Image crawling* is carried out by the *Crawler*, *Storage of results* is provided by the local storage database, and the *Scalable architecture* is permitted by scaling the *Scanner* microservice. Instead, building an analyser from scratch would require to develop all the four functionalities.

	Using DockerAnalyser	From Scratch
Image crawling	no	yes
Image analysis	yes	yes
Storage of results	no	yes
Scalable architecture	no	yes

Table I. Functionalities to implement during the design of new Docker images analysers.

We also evaluated the reusability of DOCKERANALYSER by considering both the number of reusable components and the amount of reusable code of the architecture. DOCKERANALYSER is composed by five components (*Crawler*, *Message Broker*, *Scanner*, *Images Service*, and *Images Database*). A new analyser is obtained from DOCKERANALYSER by only replacing the *Scanner* component, hence reusing the other four components. We also evaluated the amount of reusable code with the following metric (taken from [18]):

$$R = \frac{\texttt{lines of reused code}}{\texttt{total lines of code}}$$

We exploited such metric for evaluating the percentage of reused code for both the analysers we developed, viz., DOCKERFINDER (R_{df}) and DOCKERGRAPH (R_{dg}). For both DOCKERFINDER and DOCKERGRAPH the reused code is around 94%:

$$R_{df} = \frac{loc_{da}}{loc_{df}} = \frac{1861}{1971} = 0.944$$
$$R_{dg} = \frac{loc_{da}}{loc_{dg}} = \frac{1861}{1976} = 0.941$$

Notice that, in both formulas, the value of `lines of reused code` is the amount loc_{da} of lines of code of DOCKERANALYSER (which are all included also in both the analysers we developed). The values of `total lines of code` are instead the amounts loc_{df} and loc_{dg} of all lines of code of DOCKERFINDER and DOCKERGRAPH, respectively.

Finally, DOCKERANALYSER can reduce the time required to obtain new analysers starting from already existing analysers. For example, by reusing the code of DOCKERFINDER, a user may create a new analyser by only modifying the files `analysis.py` and `software.json` that we provided. She can customise the commands executed by DOCKERFINDER by modifying the `software.json` file or she can modify the lines 15-25 of Listing 3 in order to execute a different type of analysis on the images.

5. RELATED WORK

MicroBadger [26] is an on-line service that shows the contents of public Docker images, including metadata and layer information. Using MicroBadger a user can also add personalized metadata to images in order to retrieve them successively. MicroBadger differs from DOCKERANALYSER because it only permits to assign metadata to images but it does not provide a way to run customised analysis of Docker images.

Another approach that allows assigning custom properties to Docker images is JFrog [21]. JFrog's Artifactory is a universal Artefact Repository working as a single access point to software packages including Docker. JFrog can search Docker images by their name, tag or digest. Users can also assign custom properties to images, which can then be exploited to specify and resolve queries. JFrog differs from DOCKERANALYSER since permits only to assign manually custom metadata to images. DOCKERANALYSER architecture fully automates the process of assigning properties to the images based on what they feature.

Works in [33, 19, 2, 1] are frameworks that follow the serverless architecture [32] for running custom functions. The serverless functions are functions written in any language that are mapped to event triggers (e.g., HTTP requests) and scaled when needed.

Snafu [33], or Snake Functions, is a modular system to host, execute and manage language-level functions offered as stateless microservices to diverse external triggers. Functions can be executed in-memory/in-process, through external interpreters (Python 2, Java), and dynamically allocated Docker containers.

OpenLambda [19] is an Apache-licensed serverless computing project, written in Go and based on Linux containers. One of the goals of OpenLambda is to enable exploration of new approaches to serverless computing.

kubeless [2] is a Kubernetes-native serverless framework. Kubeless permits creating functions and run them on a in-cluster controller that watches and launches the functions on-demand.

IronFunctions [1] is an open source serverless platform. It can run any languages as functions and it supports AWS lambda format. Prerequisites: Docker 1.12 or later installed and running

The common characteristic of DOCKERANALYSER and serverless architectures is that both approaches allow users to provide only an analysis function while the architecture is responsible to run, scale, and manage the execution of such function. DOCKERANALYSER differs from serverless architecture because it provides also an internal storage where the description of the images produced by the analysis function are stored. The previous approaches of serverless architecture, instead, do not provide any support for storing the results of the functions.

Our work shares with Wettinger et al. [34] the general objective of contributing to ease the discovery of DevOps "knowledge" (which includes Docker images). [34] proposes a collaborative approach to store DevOps knowledge in a shared, taxonomy-based knowledge base. More precisely, [34] 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. DOCKER-ANALYSER instead focuses only on container images, and it permits building analyser that creates description of such images in a fully-automated way.

Finally, is worth noting that there exist solutions that try resolve the problems addressed by the two use cases presented in Sect. 4.

Docker Store [14] is a repository containing trusted and verified Docker images. Similar to Docker Hub, Docker store offers a search web-based interface that returns the images that match the image name, description, or the publisher name. In addition, Docker Store permits limiting the results by category (e.g., programming languages, base images, Operating System). With Docker Store it is not possible to distinguish, for instance, whether an image support a software distribution (e.g., Python, Java) since all images supporting such languages fall in the same category. DOCKERFINDER does not suffer of the same limitation, as it permits explicitly searching for images supporting either Java or Python, or both.

ImageLayers [25] is an on-line service that analyses Docker images stored in Docker Hub and shows the layers that compose them and layers that are shared by multiple images. While *ImageLayers* considers the layers composing an image, DOCKERGRAPH instead considers the parent image of an image. DOCKERGRAPH permits analysing all the images contained in a Docker Registry and constructs a graph of images. *ImageLayers* permits only analysing the layers a single image at the time and returns a flat description of a single image.

6. CONCLUSIONS

Docker images are stored in Docker Registries that allow to add, remove, distribute, and search such images. Images stored inside Docker registries are described by fixed attributes (e.g., name, description, owner of the image), which may not be enough to permit users to select the images satisfying their needs. Currently, users are required to manually download the images from remote registries and look for images satisfying the desired functionalities.

In order to solve the aforementioned problem, we presented DOCKERANALYSER a tool to build customised analysers of Docker images in a fully automated way. Users are required to provide only the analysis function and any other files needed by the analysis function, whilst DOCKERANALYSER disposes of the functionalities for crawling the images from a Docker registry, running the provided analysis on every image, storing the results of the analysis in a local storage, and searching the obtained results.

We believe that the actual value of DOCKERANALYSER is that it can be exploited by users (e.g., researchers, developers and data miners) interested in building their own analysers of Docker images. Users are only required to provide the analysis function, in the form of a Python function that, given the name of a Docker image, scans such image to extract some metadata for generating the description of the image.

We identified three main classes of analysers that can be obtained from DOCKERANALYSER, namely:

1. Analysers that execute commands inside Docker images for extracting features,
2. analysers that inspect the source code of Docker images, and
3. analysers that scan the compiled/binaries version of Docker images.

In the paper, we have shown a concrete example of analyser for class (1) and a concrete example for class (2). For (1) we presented DOCKERFINDER, an analyser that extracts the versions of the software supported by the images. For (2) we presented DOCKERGRAPH that analyses the source code stored in the GitHub repository (the Dockerfile of the image) in order to construct a graph of *parent* images. The development of an analyser in class (3), and (more generally) the development of other analysers and the identification of other classes of analysers that can be defined is left for future work.

The use cases also showed that the choice of implementing DOCKERANALYSER with a microservice-based architecture eases building customisable and scalable analysers. We indeed experimented the benefits of the scalability and replaceability properties of microservice-based architectures. In particular, replaceability allows obtaining DOCKERFINDER and DOCKERGRAPH by only replacing the Docker image of the scanner microservice. Instead, by exploiting the scalability property, we scaled the number of *Scanner* microservices of DOCKERFINDER in order to reduce the time needed to analyse the images.

We believe that DOCKERANALYSER can also take advantage of the extensibility property of microservice-based architectures that permit adding new microservices. For instance, DOCKER-ANALYSER can be extended with a *checker* microservice (such as in [5]) that maintains the consistency of the images stored in the local storage of DOCKERFINDER and those in Docker Hub.

As part of our future work we want to build DOCKERANALYSER as a web-based service where users can upload the *deploy package* folder `F` through a GUI and the web-based service creates the analyser with the provided *deploy package*, starts the analyser, and visualises the obtained images descriptions in a dashboard (e.g., with customisable charts, like that in Fig. 4).

In addition, we plan to extend DOCKERANALYSER in such a way, that (i) it permits analysing other container-based technologies (such as [7, 20]), and (ii) it permits specifying the analysis function in other programming languages (e.g., Java, Go, Bash), Finally, we plan also to implement other analysers of Docker images. For instance, a security analyser can be built by using one of the existing static analyser of Docker images (such as [8]) in order to analyse the images stored in Docker Hub discovering the images affected by security flaws.

REFERENCES

1. IronFunctions. https://github.com/iron-io/functions. Last accessed: February 20th, 2018.
2. kubeless. https://github.com/kubeless/kubeless. Last accessed: February 20th, 2018.
3. Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.
4. Antonio Brogi, Davide Neri, Luca Rinaldi, and Jacopo Soldani. From (incomplete) TOSCA specifications to running applications, with docker. In Antonio Cerone and Marco Roveri, editors, *Software Engineering and Formal Methods*, volume 10729 of *Lecture Notes in Computer Science*, pages 491–506. Springer International Publishing, 2018.
5. Antonio Brogi, Davide Neri, and Jacopo Soldani. Dockerfinder: Multi-attribute search of docker images. In *2017 IEEE International Conference on Cloud Engineering, IC2E 2017, Vancouver, BC, Canada, April 4-7, 2017*, pages 273–278. IEEE, 2017.
6. Maurizio Cavallari and Francesco Tornieri. Information systems architecture and organization in the era of microservices. In Rita Lamboglia, Andrea Cardoni, Renata Paola Dameri, and Daniela Mancini, editors, *Network, Smart and Open*, volume 24 of *Lecture Notes in Information Systems and Organisation*, pages 165–177, Cham, 2018. Springer International Publishing.
7. CoreOS. Rkt. https://coreos.com/rkt/. Last accessed: February 20th, 2018.
8. CoreOS. Vulnerability static analysis for containers. https://github.com/coreos/clair. Last accessed: February 20th, 2018.
9. DataDog. Eight surprising facts about docker adoption. https://www.datadoghq.com/docker-adoption/. Last accessed: February 20th, 2018.
10. Docker Inc. Docker. https://www.docker.com/. Last accessed: February 20th, 2018.

11. Docker Inc. Docker Compose. https://docs.docker.com/compose/overview/. Last accessed: February 20th, 2018.
12. Docker Inc. Docker Hub. https://hub.docker.com/. Last accessed: February 20th, 2018.
13. Docker Inc. Docker Hub hits 5 billion pulls. https://blog.docker.com/2016/08/docker-hub-hits-5-billion-pulls. Last accessed: February 20th, 2018.
14. Docker Inc. Docker Store. https://store.docker.com/. Last accessed: February 20th, 2018.
15. Docker Inc. Docker Swarm. https://docs.docker.com/swarm/. Last accessed: February 20th, 2018.
16. Express. Fast, unopinionated, minimalist web framework for node.js. http://expressjs.com/. Last accessed: February 20th, 2018.
17. Martin Fowler and James Lewis. Microservices. ThoughtWorks, https://martinfowler.com/articles/microservices.html. Last accessed: February 20th, 2018.
18. William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Comput. Surv.*, 28(2):415–435, June 1996.
19. Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.
20. Heroku. Dynos and the Dyno Manager. https://devcenter.heroku.com/articles/dynos. Last accessed: February 20th, 2018.
21. JFrog Ltd. Docker: Secure Clustered HA Docker Registries With A Universal Artifact Repository. https://www.jfrog.com/support-service/whitepapers/docker/. Last accessed: February 20th, 2018.
22. Ann Mary Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346, March 2015.
23. Zhanibek Kozhirbayev and Richard O. Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175 – 182, 2017.
24. Zheng Li, Maria Kihl, Qinghua Lu, and Jens A. Andersson. Performance overhead comparison between hypervisor and container based virtualization. In Leonard Barolli, Makoto Takizawa, Tomoya Enokido, Hui-Huang Hsu, and Chi-Yi Lin, editors, *31st IEEE International Conference on Advanced Information Networking and Applications, AINA 2017, Taipei, Taiwan, March 27-29, 2017*, pages 955–962. IEEE Computer Society, 2017.
25. Microscaling System. Image layers. https://imagelayers.io/. Last accessed: February 20th, 2018.
26. Microscaling System. Microbadger. https://microbadger.com/. Last accessed: February 20th, 2018.
27. Mongoose. Elegant mongodb object modelling for node.js. http://mongoosejs.com/. Last accessed: February 20th, 2018.
28. Sam Newman. *Building microservices*. O'Reilly Media, Inc., 2015.
29. Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 2017. *In press*, DOI: 10.1109/TCC.2017.2702586.
30. Pika. Introduction to pika. https://pika.readthedocs.io. Last accessed: February 20th, 2018.
31. Requests. Requests: Http for humans. http://docs.python-requests.org/. Last accessed: February 20th, 2018.
32. Mike Roberts. Serverless Architectures. https://martinfowler.com/articles/serverless.html. Last accessed: February 20th, 2018.
33. Josef Spillner. Snafu: Function-as-a-service (faas) runtime design and implementation. *CoRR*, abs/1703.07562, 2017.
34. 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.